OPTIMIZING THE HYPERPARAMETERS OF ML (MLP, SVM, NAIVE BAYES) IN PREDICTING BREAST CANCER

AIM

The project task is to analyze cell images and classify tumors as either cancerous or non-cancerous based on the features computed from the digitized images of a breast mass. The features are computed from the characteristics of the cell nuclei present in the image. The goal is to accurately determine if the tumor is malignant or benign.

Attribute Information(kaggle.com, n.d.):

id: ID number, diagnosis: The diagnosis of breast tissues (M = malignant, B = benign), radius_mean: mean of distances from center to points on the perimeter, texture_mean: standard deviation of gray-scale values, perimeter_mean: mean size of the core tumor, area_mean: area of the tumor, smoothness_mean: mean of local variation in radius lengths, compactness_mean: mean of perimeter^2 / area - 1.0, concavity_mean: mean of severity of concave portions of the contour, concave_points_mean: mean for number of concave portions of the contour, symmetry_mean, fractal_dimension_mean: mean for "coastline approximation" - 1, radius_se: standard error for the mean of distances from center to points on the perimeter, texture_se: standard error for standard deviation of gray-scale values, perimeter_se, area_se, smoothness_se: standard error for local variation in radius lengths, compactness_se: standard error for perimeter^2 / area - 1.0, concavity_se: standard error for severity of concave portions of the contour, concave_points_se: standard error for number of concave portions of the contour, symmetry_se, fractal_dimension_se: standard error for "coastline approximation" - 1, radius_worst: "worst" or largest mean value for mean of distances from center to points on the perimeter, texture_worst: "worst" or largest mean value for standard deviation of gray-scale values, perimeter_worst, area_worst, smoothness_worst: "worst" or largest mean value for local variation in radius lengths, compactness_worst: "worst" or largest mean value for perimeter^2 / area - 1.0, concavity_worst: "worst" or largest mean value for severity of concave portions of the contour, concave_points_worst: "worst" or largest mean value for number of concave portions of the contour, symmetry_worst, fractal_dimension_worst: "worst" or largest mean value for "coastline approximation" - 1

Importing Dependencies

In [1]:
```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from keras.models import Sequential
from keras.layers import Dense, Activation, Dropout
```

```
2023-06-29 17:37:57.514551: I tensorflow/core/platform/cpu_feature_guard.cc:182] This TensorFlow binary
is optimized to use available CPU instructions in performance-critical operations.
To enable the following instructions: AVX2 FMA, in other operations, rebuild TensorFlow with the approp
riate compiler flags.
```

Data Collection & Loading

In [2]:
```
df = pd.read_csv('/Users/oluwatoyineleja/Downloads/WISCONSIN.csv')
```

In [3]:
```
#print the first 5 rows of dataframe
df.head()
```

Out[3]:

	id	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_mean	pc
0	842302	M	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001	
1	842517	M	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869	
2	84300903	M	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974	
3	84348301	M	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414	
4	84358402	M	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980	

5 rows × 33 columns

In [4]:
```
#print last 5 rows of the dataframe
df.tail()
```

Out[4]:

	id	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_mean	pc
564	926424	M	21.56	22.39	142.00	1479.0	0.11100	0.11590	0.24390	
565	926682	M	20.13	28.25	131.20	1261.0	0.09780	0.10340	0.14400	
566	926954	M	16.60	28.08	108.30	858.1	0.08455	0.10230	0.09251	
567	927241	M	20.60	29.33	140.10	1265.0	0.11780	0.27700	0.35140	
568	92751	B	7.76	24.54	47.92	181.0	0.05263	0.04362	0.00000	

5 rows × 33 columns

In [5]:
```
#view number of rows and colums in the dataset
df.shape
```

Out[5]:
```
(569, 33)
```

In [6]:
```
#getting info about the data
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 569 entries, 0 to 568
Data columns (total 33 columns):
 #   Column                   Non-Null Count  Dtype  
---  ------                   --------------  -----  
 0   id                       569 non-null    int64  
 1   diagnosis                569 non-null    object 
 2   radius_mean              569 non-null    float64
 3   texture_mean             569 non-null    float64
 4   perimeter_mean           569 non-null    float64
 5   area_mean                569 non-null    float64
 6   smoothness_mean          569 non-null    float64
 7   compactness_mean         569 non-null    float64
 8   concavity_mean           569 non-null    float64
 9   concave points_mean      569 non-null    float64
 10  symmetry_mean            569 non-null    float64
 11  fractal_dimension_mean   569 non-null    float64
 12  radius_se                569 non-null    float64
 13  texture_se               569 non-null    float64
 14  perimeter_se             569 non-null    float64
 15  area_se                  569 non-null    float64
 16  smoothness_se            569 non-null    float64
 17  compactness_se           569 non-null    float64
 18  concavity_se             569 non-null    float64
 19  concave points_se        569 non-null    float64
 20  symmetry_se              569 non-null    float64
 21  fractal_dimension_se     569 non-null    float64
 22  radius_worst            569 non-null    float64
 23  texture_worst           569 non-null    float64
 24  perimeter_worst         569 non-null    float64
 25  area_worst              569 non-null    float64
 26  smoothness_worst        569 non-null    float64
 27  compactness_worst       569 non-null    float64
 28  concavity_worst         569 non-null    float64
 29  concave points_worst    569 non-null    float64
 30  symmetry_worst          569 non-null    float64
 31  fractal_dimension_worst 569 non-null    float64
 32  Unnamed: 32             0 non-null      float64
dtypes: float64(31), int64(1), object(1)
memory usage: 146.8+ KB
```

In [7]:
```
#statistical measures about the data
df.describe()
```

Out[7]:

	id	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_mean	poi
count	5.690000e+02	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	56
mean	3.037183e+07	14.127292	19.289649	91.969033	654.889104	0.096360	0.104341	0.088799	
std	1.250206e+08	3.524049	4.301036	24.298981	351.914129	0.014064	0.052813	0.079720	
min	8.670000e+03	6.981000	9.710000	43.790000	143.500000	0.052630	0.019380	0.000000	
25%	8.692180e+05	11.700000	16.170000	75.170000	420.300000	0.086370	0.064920	0.029560	
50%	9.060240e+05	13.370000	18.840000	86.240000	551.100000	0.095870	0.092630	0.061540	
75%	8.813129e+06	15.780000	21.800000	104.100000	782.700000	0.105300	0.130400	0.130700	
max	9.113205e+08	28.110000	39.280000	188.500000	2501.000000	0.163400	0.345400	0.426800	

8 rows × 32 columns

Preparing & Cleaning the Data

In [8]:
```
#checking for missing values
df.isnull().sum()
```

Out[8]:
```
id                         0
diagnosis                  0
radius_mean                0
texture_mean               0
perimeter_mean             0
area_mean                  0
smoothness_mean            0
compactness_mean           0
concavity_mean             0
concave points_mean        0
symmetry_mean              0
fractal_dimension_mean     0
radius_se                  0
texture_se                 0
perimeter_se               0
area_se                    0
smoothness_se              0
compactness_se             0
concavity_se               0
concave points_se          0
symmetry_se                0
fractal_dimension_se       0
radius_worst               0
texture_worst              0
perimeter_worst            0
area_worst                 0
smoothness_worst           0
compactness_worst          0
concavity_worst            0
```

```
concave points_worst         0
symmetry_worst               0
fractal_dimension_worst      0
Unnamed: 32                569
dtype: int64
```

Define Target and Input Variables

In [9]:
```
#Rename Diagnosis to "Label" to easily identify our target variable
df = df.rename(columns = {'diagnosis':'label'})
print(df.dtypes)
```

```
id                           int64
label                       object
radius_mean                float64
texture_mean               float64
perimeter_mean             float64
area_mean                  float64
smoothness_mean            float64
compactness_mean           float64
concavity_mean             float64
concave points_mean        float64
symmetry_mean              float64
fractal_dimension_mean     float64
radius_se                  float64
texture_se                 float64
perimeter_se               float64
area_se                    float64
smoothness_se              float64
compactness_se             float64
concavity_se               float64
concave points_se          float64
symmetry_se                float64
fractal_dimension_se       float64
```

```
radius_worst               float64
texture_worst              float64
perimeter_worst            float64
area_worst                 float64
smoothness_worst           float64
compactness_worst          float64
concavity_worst            float64
concave points_worst       float64
symmetry_worst             float64
fractal_dimension_worst    float64
Unnamed: 32                float64
dtype: object
```

Separating the features and target variables

In [10]:
```
#convert 'Target' categorical variable from [M]&[B] to [0]&[1] using Label Encoding

from sklearn.preprocessing import LabelEncoder

labelencoder = LabelEncoder()
Y = labelencoder.fit_transform(df["label"].values)
print("Label after encoding are: ", np.unique(Y))
```

```
Label after encoding are:  [0 1]
```

In [11]:
```
#dropping column "id" and target "label"
X = df.drop(labels=["label", "id"], axis=1)
```

In [12]:

```
#view input variable
print (X)
```

```
     radius_mean  texture_mean  perimeter_mean  area_mean  smoothness_mean  \
0          17.99         10.38          122.80     1001.0          0.11840   
1          20.57         17.77          132.90     1326.0          0.08474   
2          19.69         21.25          130.00     1203.0          0.10960   
3          11.42         20.38           77.58      386.1          0.14250   
4          20.29         14.34          135.10     1297.0          0.10030   
..           ...           ...             ...        ...              ...   
564        21.56         22.39          142.00     1479.0          0.11100   
565        20.13         28.25          131.20     1261.0          0.09780   
566        16.60         28.08          108.30      858.1          0.08455   
567        20.60         29.33          140.10     1265.0          0.11780   
568         7.76         24.54           47.92      181.0          0.05263   

     compactness_mean  concavity_mean  concave points_mean  symmetry_mean  \
0             0.27760         0.30010              0.14710         0.2419   
1             0.07864         0.08690              0.07017         0.1812   
2             0.15990         0.19740              0.12790         0.2069   
3             0.28390         0.24140              0.10520         0.2597   
4             0.13280         0.19800              0.10430         0.1809   
..                ...             ...                  ...            ...   
564           0.11590         0.24390              0.13890         0.1726   
565           0.10340         0.14400              0.09791         0.1752   
566           0.10230         0.09251              0.05302         0.1590   
567           0.27700         0.35140              0.15200         0.2397   
568           0.04362         0.00000              0.00000         0.1587   

     fractal_dimension_mean  ...  texture_worst  perimeter_worst  area_worst  \
0                   0.07871  ...          17.33           184.60      2019.0   
1                   0.05667  ...          23.41           158.80      1956.0   
2                   0.05999  ...          25.53           152.50      1709.0   
3                   0.09744  ...          26.50            98.87       567.7   
4                   0.05883  ...          16.67           152.20      1575.0   
```

```
..                ...  ...          ...              ...        ...
564           0.05623  ...        26.40           166.10     2027.0
565           0.05533  ...        38.25           155.00     1731.0
566           0.05648  ...        34.12           126.70     1124.0
567           0.07016  ...        39.42           184.60     1821.0
568           0.05884  ...        30.37            59.16      268.6

     smoothness_worst  compactness_worst  concavity_worst  \

0             0.16220            0.66560           0.7119
1             0.12380            0.18660           0.2416
2             0.14440            0.42450           0.4504
3             0.20980            0.86630           0.6869
4             0.13740            0.20500           0.4000
..                ...                ...              ...
564           0.14100            0.21130           0.4107
565           0.11660            0.19220           0.3215
566           0.11390            0.30940           0.3403
567           0.16500            0.86810           0.9387
568           0.08996            0.06444           0.0000

     concave points_worst  symmetry_worst  fractal_dimension_worst  \
0                  0.2654          0.4601                  0.11890
1                  0.1860          0.2750                  0.08902
2                  0.2430          0.3613                  0.08758
3                  0.2575          0.6638                  0.17300
4                  0.1625          0.2364                  0.07678
..                    ...             ...                      ...
564                0.2216          0.2060                  0.07115
565                0.1628          0.2572                  0.06637
566                0.1418          0.2218                  0.07820
567                0.2650          0.4087                  0.12400
568                0.0000          0.2871                  0.07039

     Unnamed: 32
0            NaN
```

```
1             NaN
2             NaN
3             NaN
4             NaN
..            ...
564           NaN
565           NaN
566           NaN

567           NaN
568           NaN

[569 rows x 31 columns]
```

In [13]:
```
#view target variable
print (Y)
```

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

In [14]:
```
#CHECKING THE DISTRIBUTION OF TARGET VARIABLE
df['label'].value_counts()
```

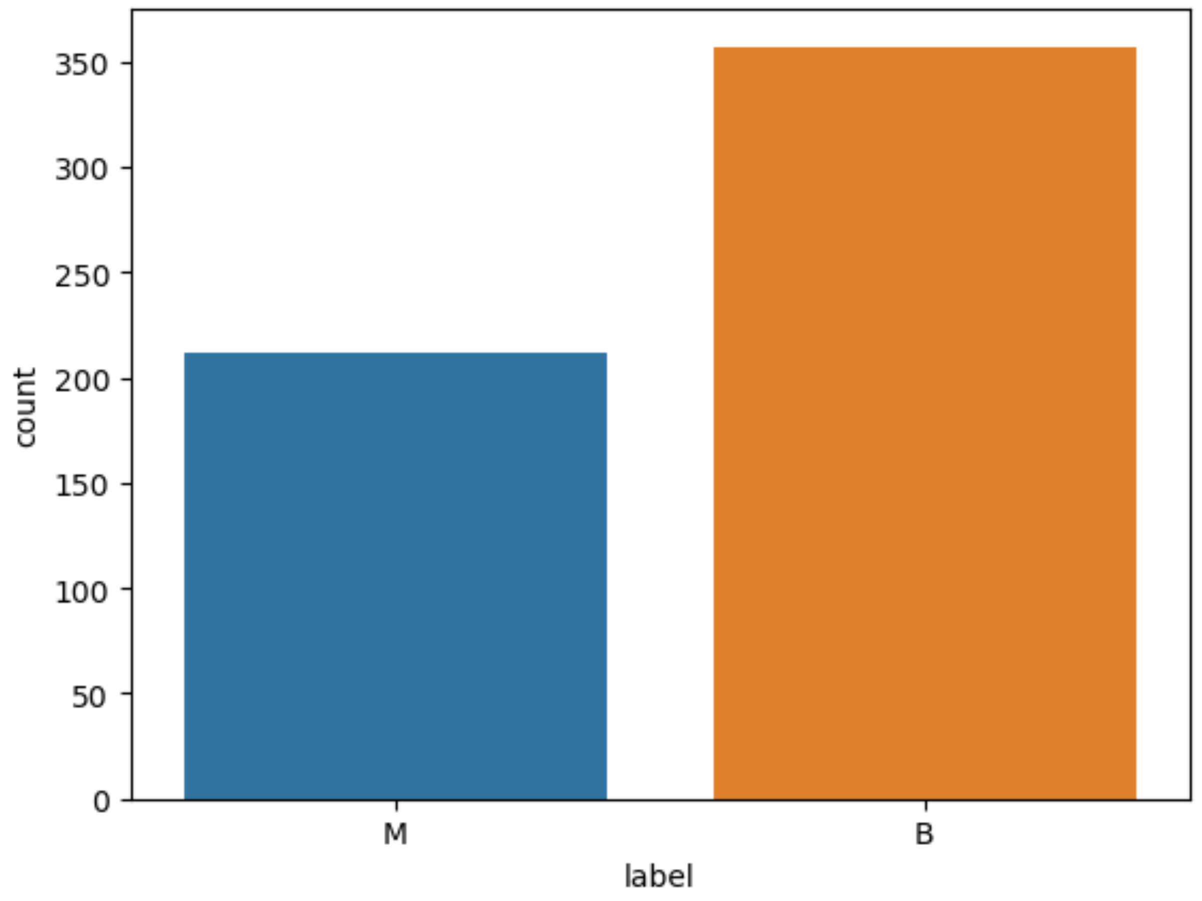
Out[14]:
```
B    357
M    212
Name: label, dtype: int64
```

Visualizing the Data

In this section we will develop some visualizations of the data in order to decide how to proceed with the multi-later perceptron model and machine learning algorithms.

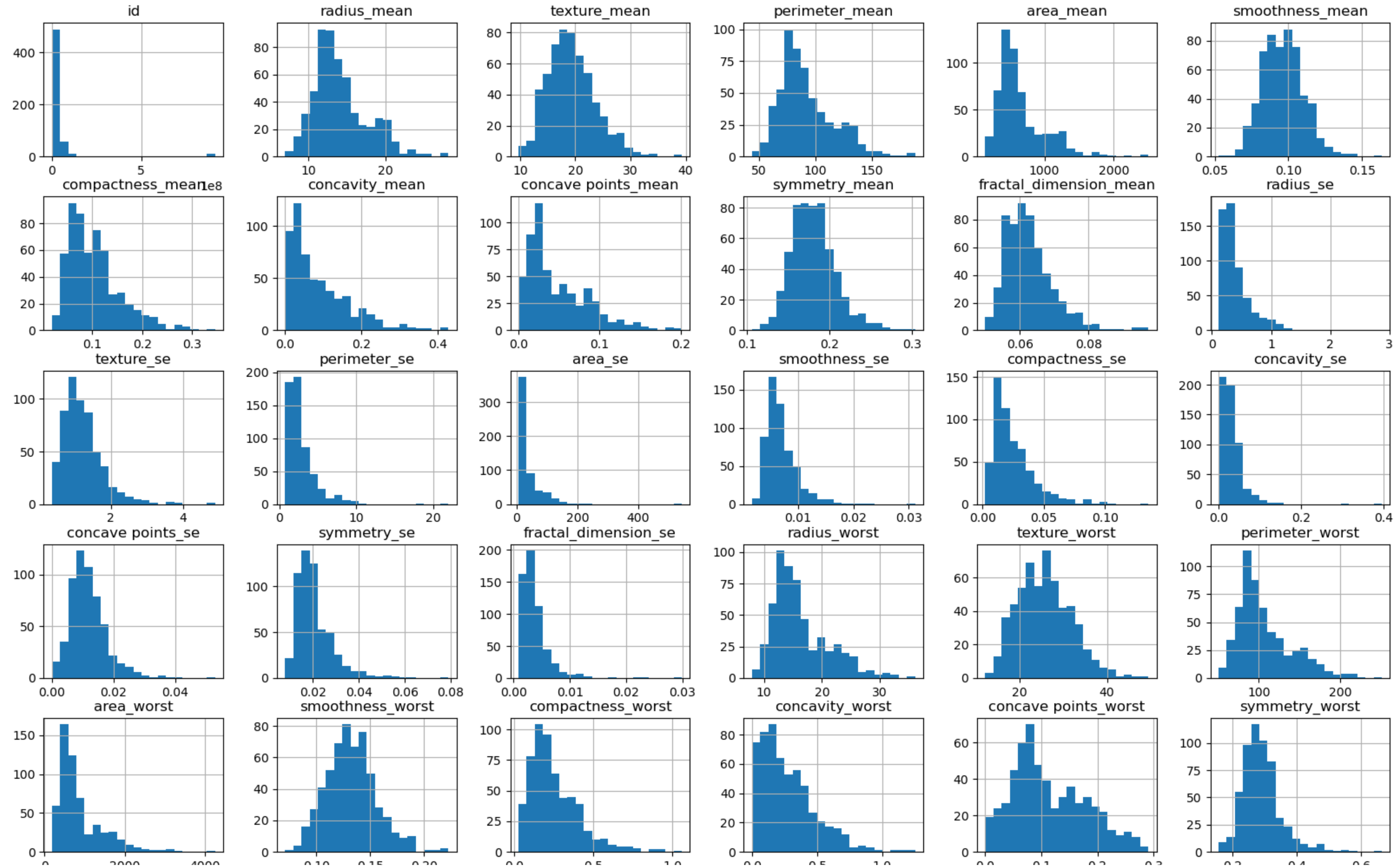
In [15]:
```
#countplot to view how many benign and malignant cases present in dataset
sns.countplot(data=df, x='label')
```

Out[15]:
```
<AxesSubplot:xlabel='label', ylabel='count'>
```



In [16]:
```
```

```
#visualizing the distribution of each feature
df.hist(bins=20, figsize=(20,15))
plt.show()
```



fractal_dimension_worst
Unnamed: 32

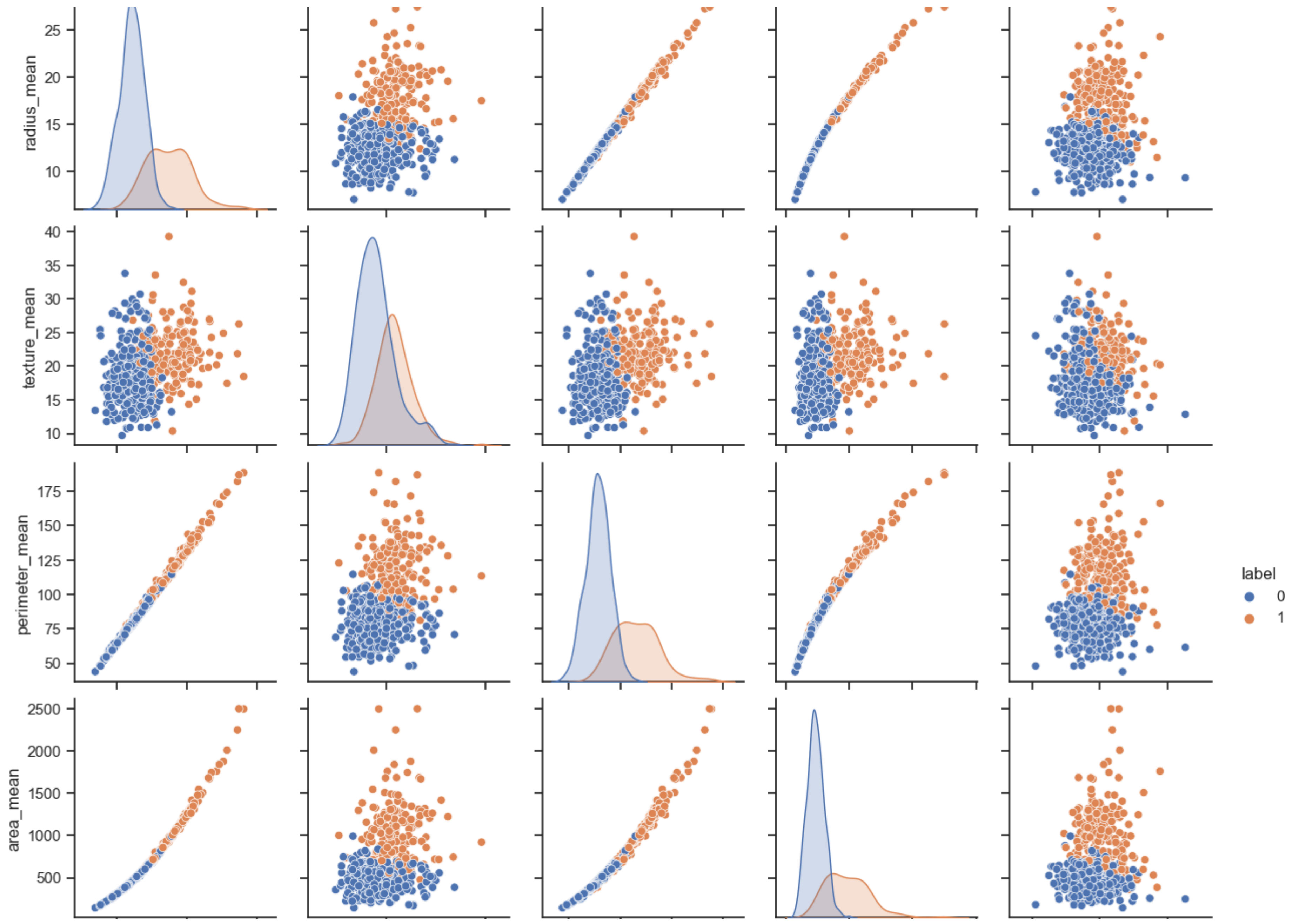
Group input variables

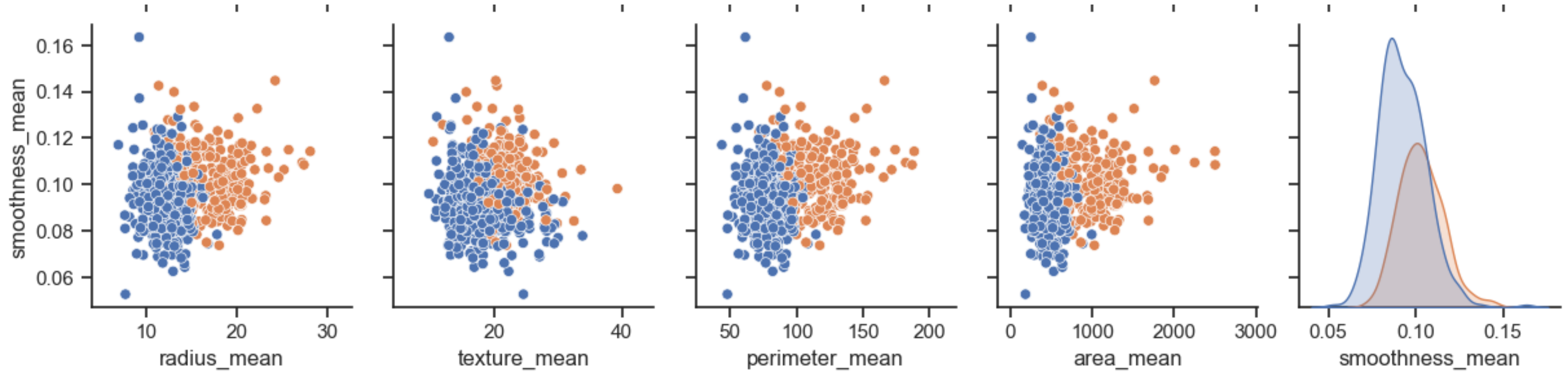
In [17]:
```
mean_features = list(df.columns[1:11])
se_features = list(df.columns[11:21])
worst_features = list(df.columns[21:31])
```

In [18]:
```
#define pairplot's x and y axis
X_pairplot = X[["radius_mean", "texture_mean", "perimeter_mean", "area_mean", "smoothness_mean"]]
X_pairplot["label"] = Y

# Create a pairplot of the data
sns.set(style="ticks")
sns.pairplot(X_pairplot, hue="label", vars=["radius_mean", "texture_mean", "perimeter_mean", "area_mean"
```

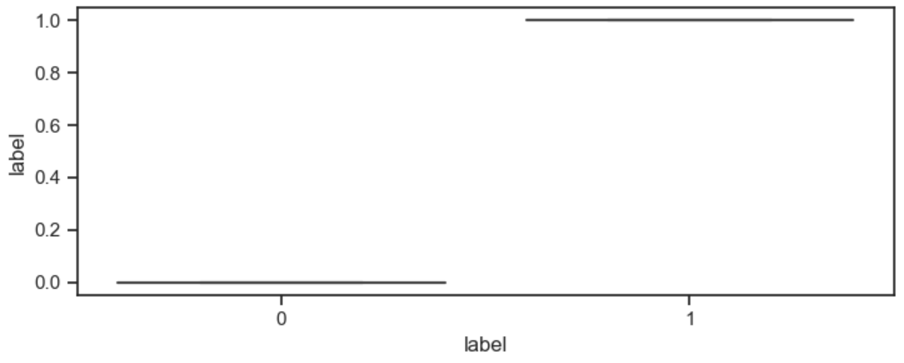
```
/var/folders/vt/j0bzfy6n54g9yf297psjl07w0000gn/T/ipykernel_7188/3861720049.py:3: SettingWithCopyWarning
:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.
html#returning-a-view-versus-a-copy (https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.h
tml#returning-a-view-versus-a-copy)
  X_pairplot["label"] = Y
```

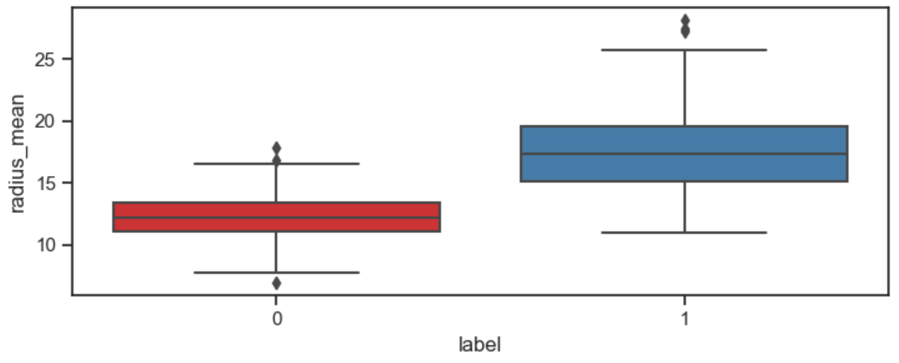
Out[18]:
```
<seaborn.axisgrid.PairGrid at 0x7ff14b854f70>
```

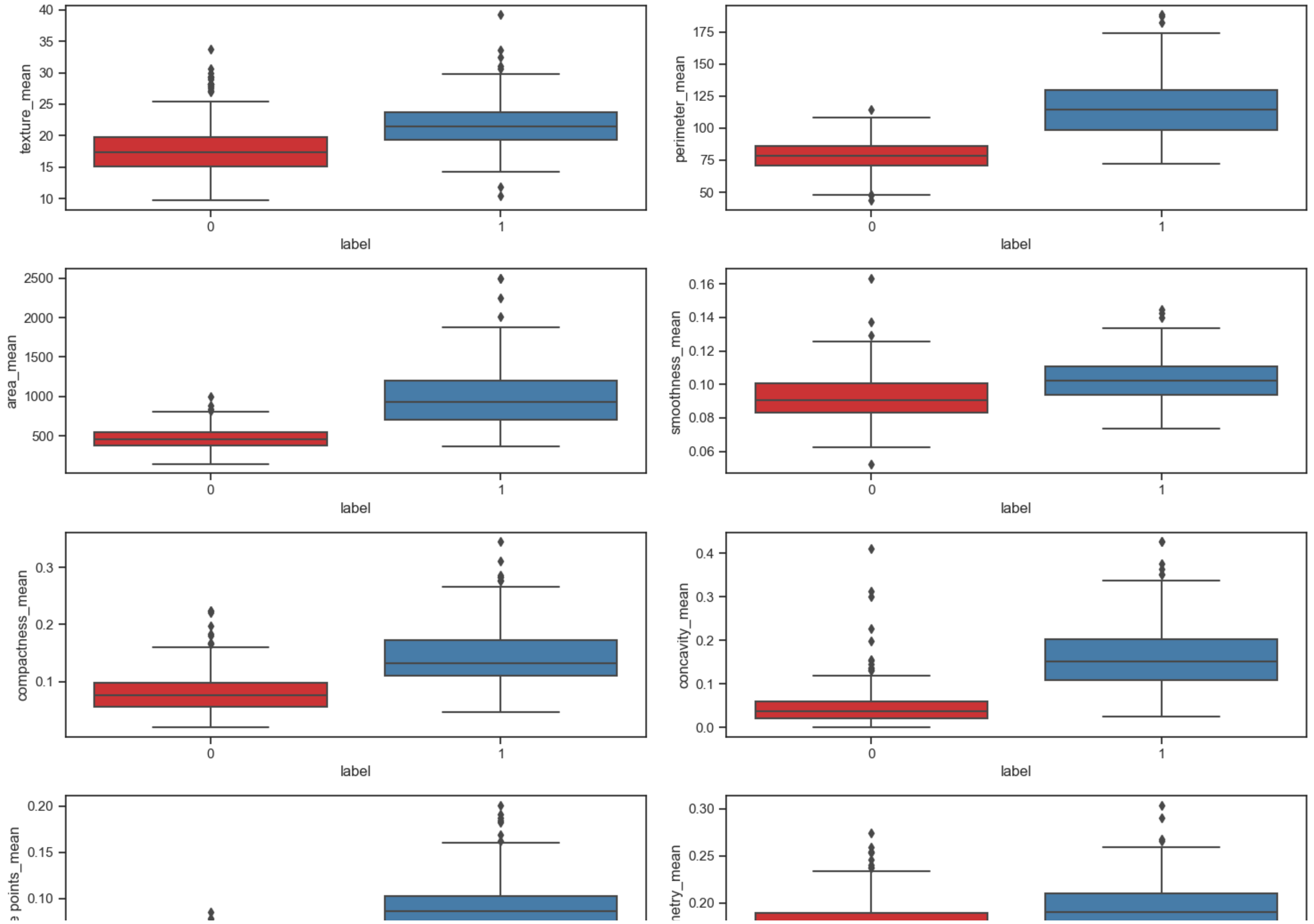

radius_mean
texture_mean
perimeter_mean
area_mean
label
0
1

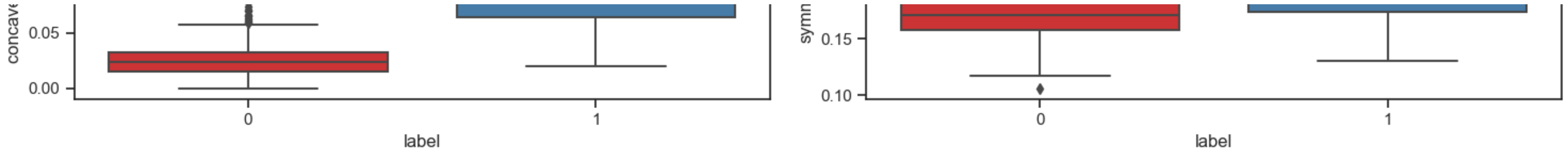


In [19]:
```
# Combine the target variable and features into a single dataframe
data = pd.concat([pd.DataFrame(Y, columns=["label"]), X], axis=1)
plt.figure(figsize=(15,15))
for i, feature in enumerate(mean_features):
    rows = int(len(mean_features)/2)

    plt.subplot(rows, 2, 1+i)

    sns.boxplot(x='label', y=feature, data=data, palette='Set1')

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

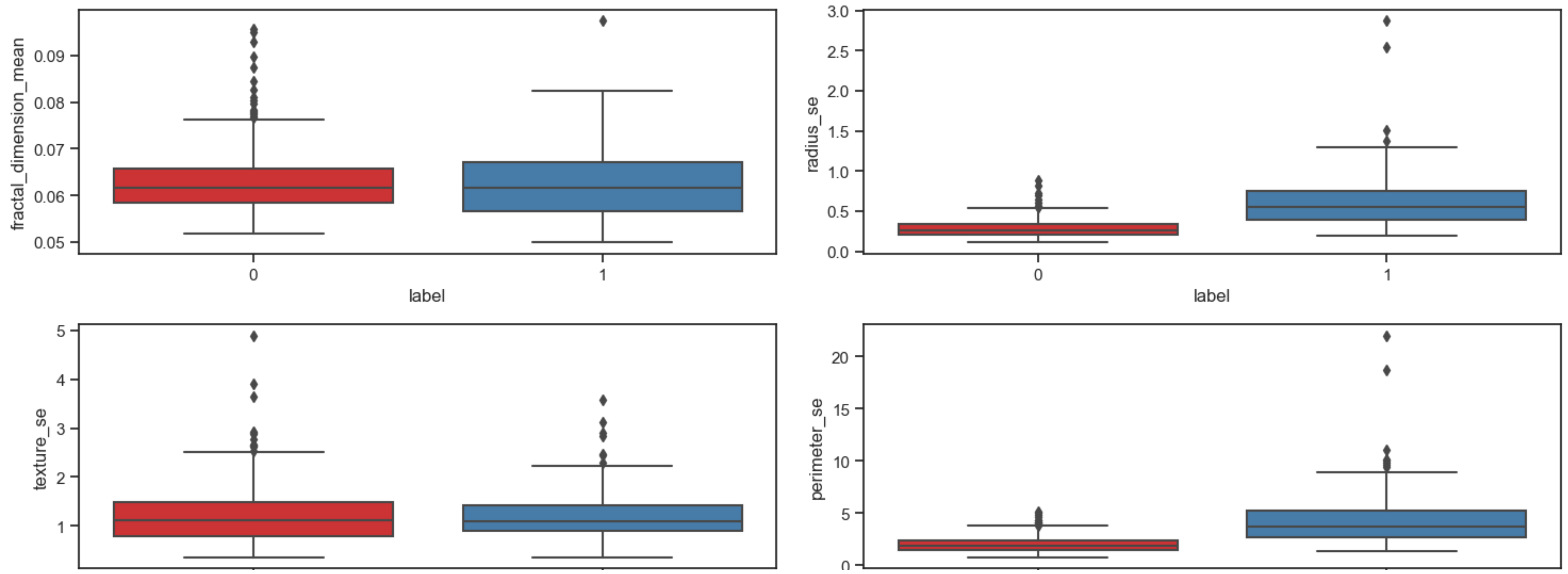


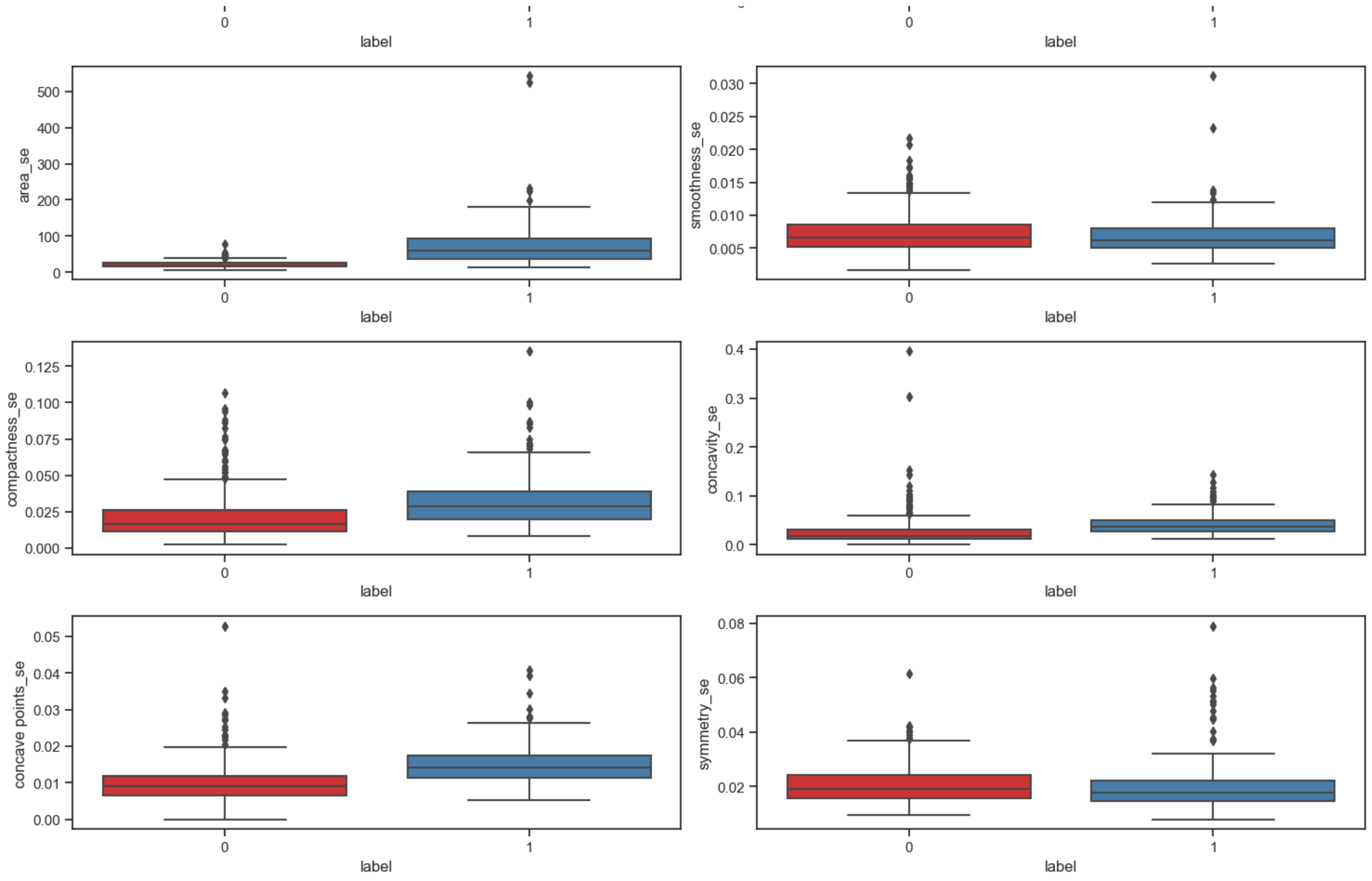



texture_mean
40
35
30
25
20
15
10
0
1
label
perimeter_mean
175
150
125
100
75
50
0
1
label
area_mean
2500
2000
1500
1000
500
0
1
label
smoothness_mean
0.16
0.14
0.12
0.10
0.08
0.06
0
1
label
compactness_mean
0.3
0.2
0.1
0
1
label
concavity_mean
0.4
0.3
0.2
0.1
0.0
0
1
label
points_mean
0.20
0.15
0.10
metry_mean
0.30
0.25
0.20



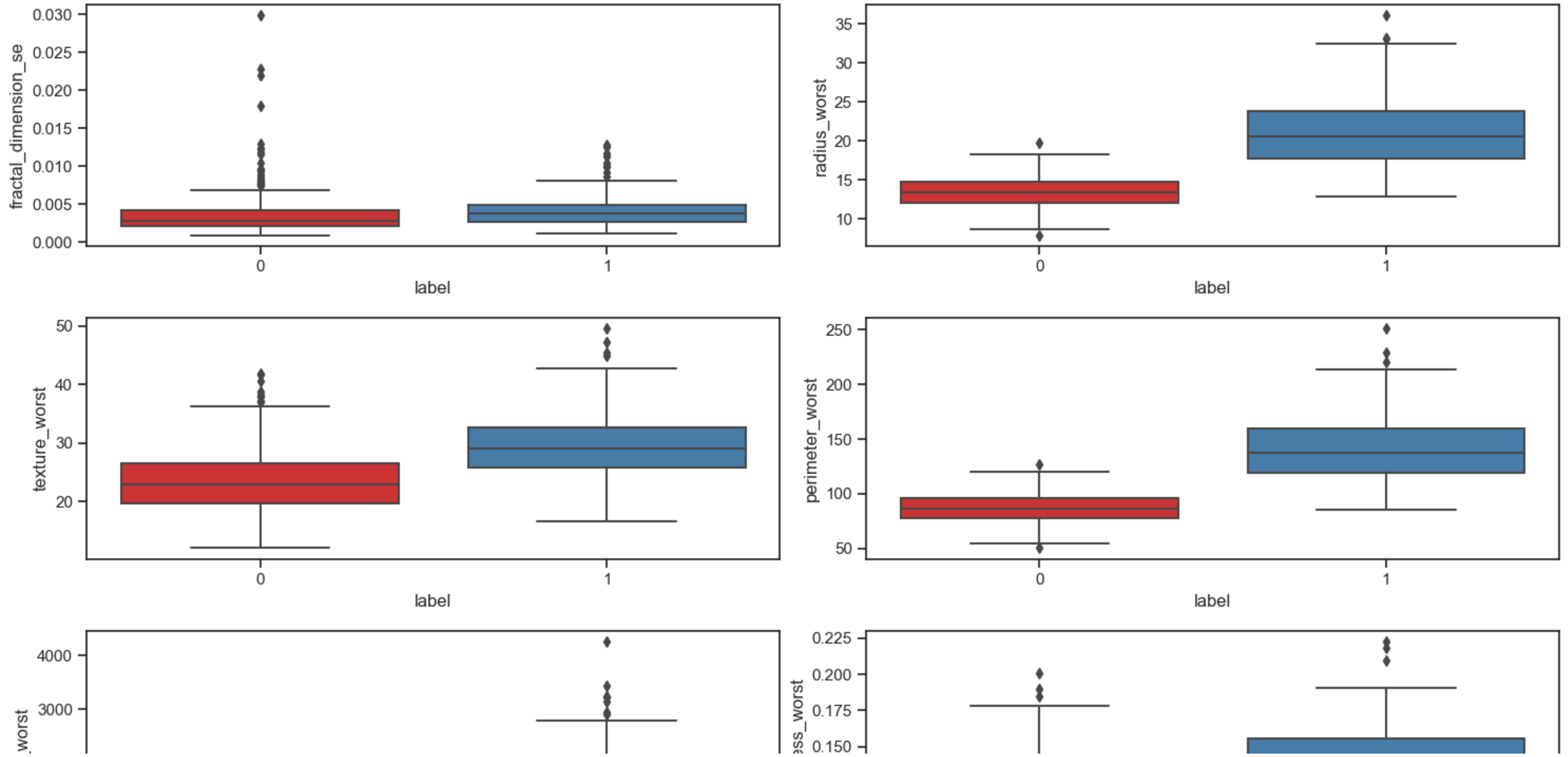
In [20]:
```
plt.figure(figsize=(15,15))
for i, feature in enumerate(se_features):
    rows = int(len(mean_features)/2)

    plt.subplot(rows, 2, 1+i)

    sns.boxplot(x='label', y=feature, data=data, palette='Set1')

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

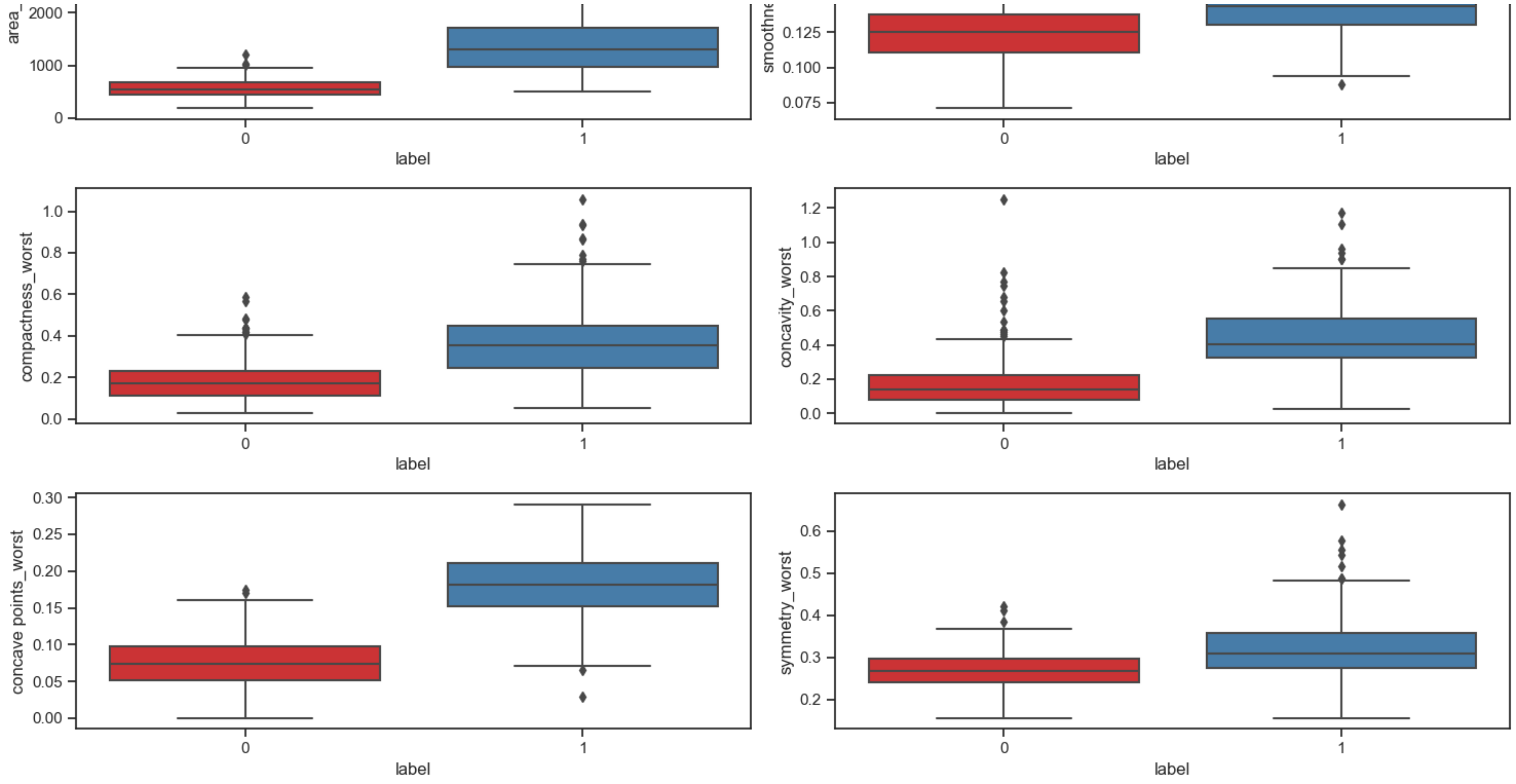



label
area_se
smoothness_se
compactness_se
concavity_se
concave points_se
symmetry_se
0
1

In [21]:

```
plt.figure(figsize=(15,15))
for i, feature in enumerate(worst_features):
    rows = int(len(mean_features)/2)

    plt.subplot(rows, 2, 1+i)

    sns.boxplot(x='label', y=feature, data=data, palette='Set1')

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





Explore Correlation

In [22]:
```
#correlation of all features
data.corr()
```

Out[22]:

label	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavit

label	1.000000	0.730029	0.415185	0.742636	0.708984	0.358560	0.596534	0
radius_mean	0.730029	1.000000	0.323782	0.997855	0.987357	0.170581	0.506124	0
texture_mean	0.415185	0.323782	1.000000	0.329533	0.321086	-0.023389	0.236702	0
perimeter_mean	0.742636	0.997855	0.329533	1.000000	0.986507	0.207278	0.556936	0
area_mean	0.708984	0.987357	0.321086	0.986507	1.000000	0.177028	0.498502	0
smoothness_mean	0.358560	0.170581	-0.023389	0.207278	0.177028	1.000000	0.659123	0
compactness_mean	0.596534	0.506124	0.236702	0.556936	0.498502	0.659123	1.000000	0
concavity_mean	0.696360	0.676764	0.302418	0.716136	0.685983	0.521984	0.883121	1
concave points_mean	0.776614	0.822529	0.293464	0.850977	0.823269	0.553695	0.831135	0
symmetry_mean	0.330499	0.147741	0.071401	0.183027	0.151293	0.557775	0.602641	0
fractal_dimension_mean	-0.012838	-0.311631	-0.076437	-0.261477	-0.283110	0.584792	0.565369	0
radius_se	0.567134	0.679090	0.275869	0.691765	0.732562	0.301467	0.497473	0
texture_se	-0.008303	-0.097317	0.386358	-0.086761	-0.066280	0.068406	0.046205	0
perimeter_se	0.556141	0.674172	0.281673	0.693135	0.726628	0.296092	0.548905	0
area_se	0.548236	0.735864	0.259845	0.744983	0.800086	0.246552	0.455653	0
smoothness_se	-0.067016	-0.222600	0.006614	-0.202694	-0.166777	0.332375	0.135299	0
compactness_se	0.292999	0.206000	0.191975	0.250744	0.212583	0.318943	0.738722	0
concavity_se	0.253730	0.194204	0.143293	0.228082	0.207660	0.248396	0.570517	0
concave points_se	0.408042	0.376169	0.163851	0.407217	0.372320	0.380676	0.642262	0
symmetry_se	-0.006522	-0.104321	0.009127	-0.081629	-0.072497	0.200774	0.229977	0
fractal_dimension_se	0.077972	-0.042641	0.054458	-0.005523	-0.019887	0.283607	0.507318	0

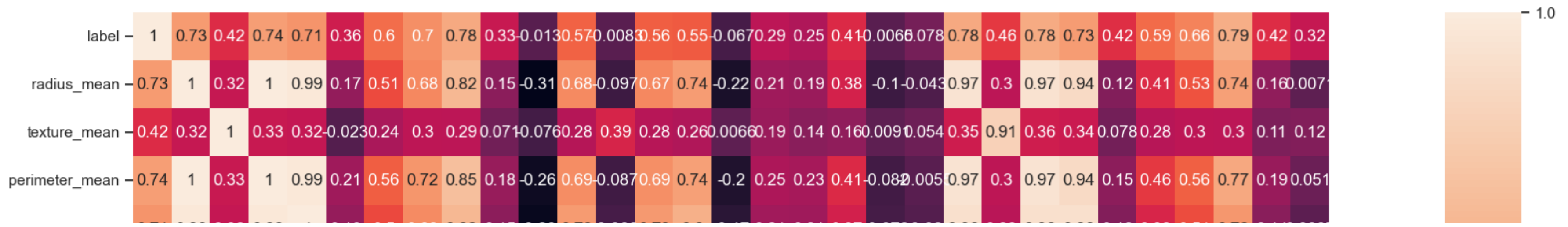
radius_worst	0.776454	0.969539	0.352573	0.969476	0.962746	0.213120	0.535315	0
texture_worst	0.456903	0.297008	0.912045	0.303038	0.287489	0.036072	0.248133	0
perimeter_worst	0.782914	0.965137	0.358040	0.970387	0.959120	0.238853	0.590210	0
area_worst	0.733825	0.941082	0.343546	0.941550	0.959213	0.206718	0.509604	0
smoothness_worst	0.421465	0.119616	0.077503	0.150549	0.123523	0.805324	0.565541	0
compactness_worst	0.590998	0.413463	0.277830	0.455774	0.390410	0.472468	0.865809	0
concavity_worst	0.659610	0.526911	0.301025	0.563879	0.512606	0.434926	0.816275	0
concave points_worst	0.793566	0.744214	0.295316	0.771241	0.722017	0.503053	0.815573	0
symmetry_worst	0.416294	0.163953	0.105008	0.189115	0.143570	0.394309	0.510223	0
fractal_dimension_worst	0.323872	0.007066	0.119205	0.051019	0.003738	0.499316	0.687382	0
Unnamed: 32	NaN	NaN	NaN	NaN	NaN	NaN	NaN	

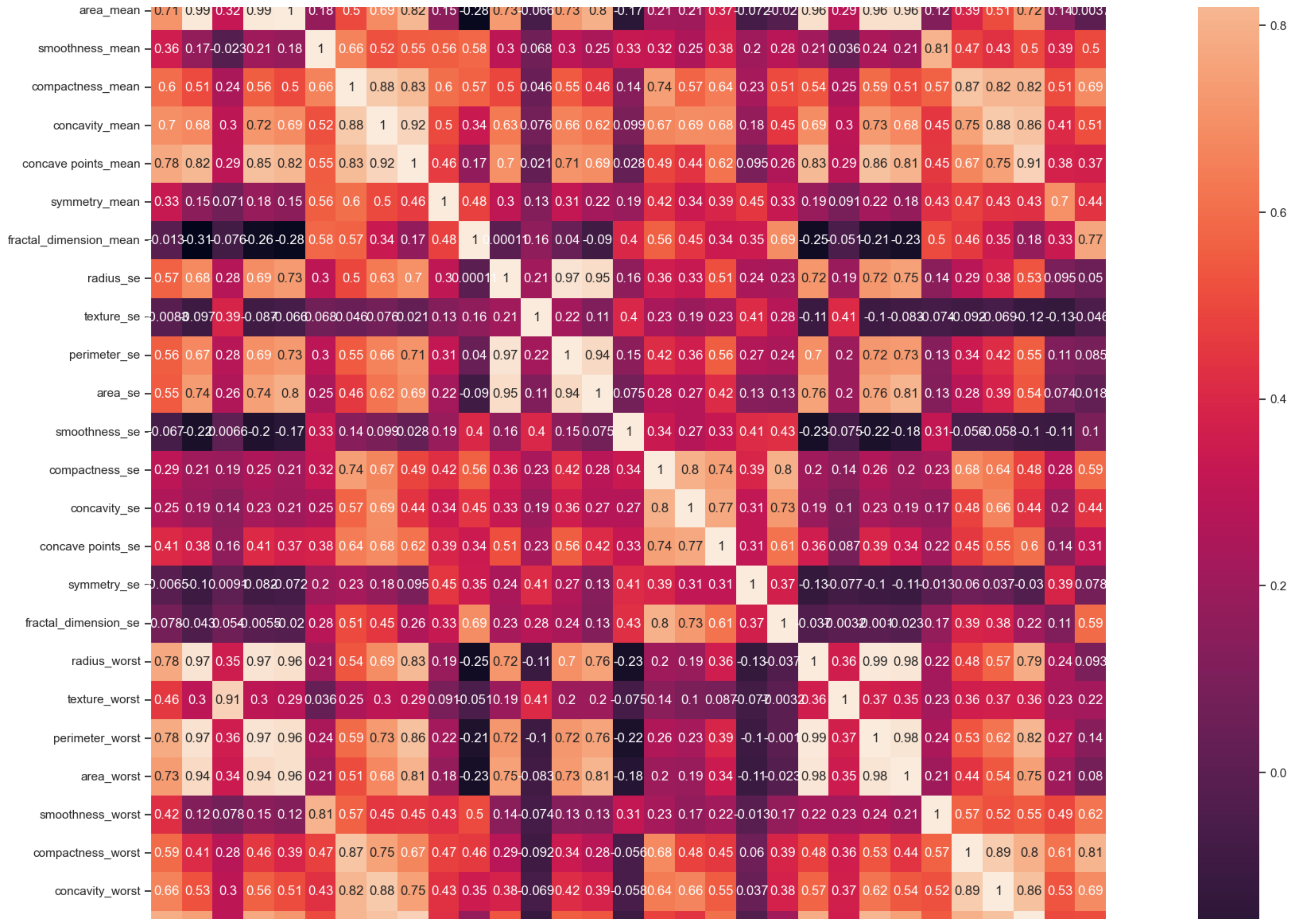
32 rows × 32 columns

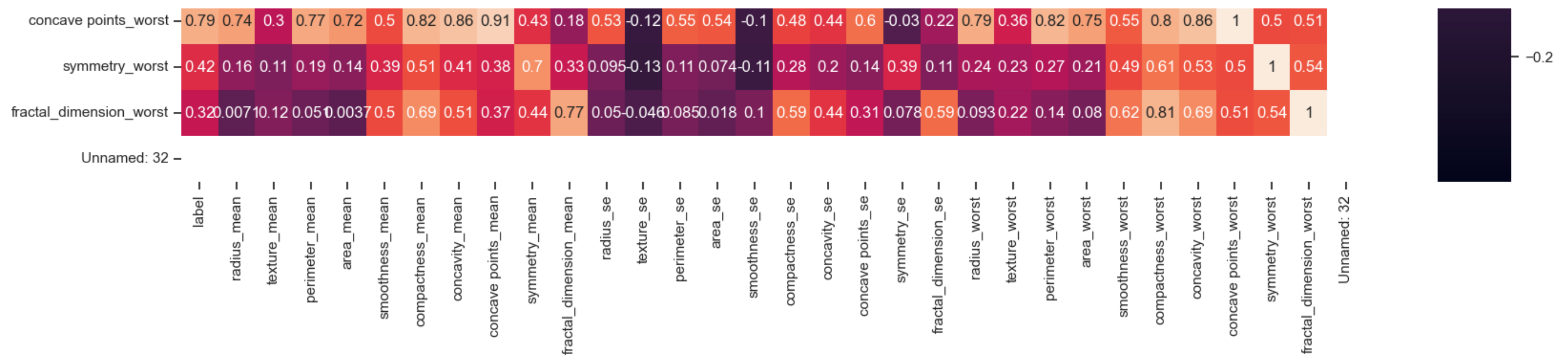
radius_mean, perimeter_mean,area_mean have a high correlation with malignant tumor

In [23]:
```
#correlation matrix of all features
plt.figure(figsize=(20,20))
sns.heatmap(data.corr(),annot=True)
```

Out[23]:
```
<AxesSubplot:>
```

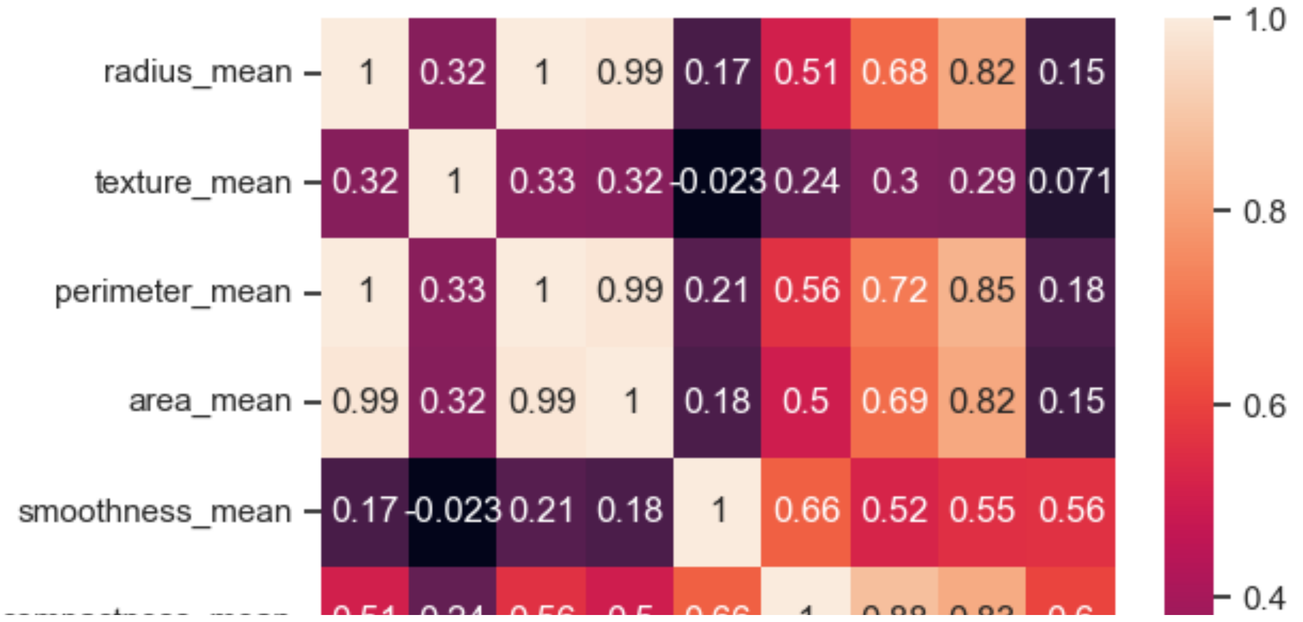


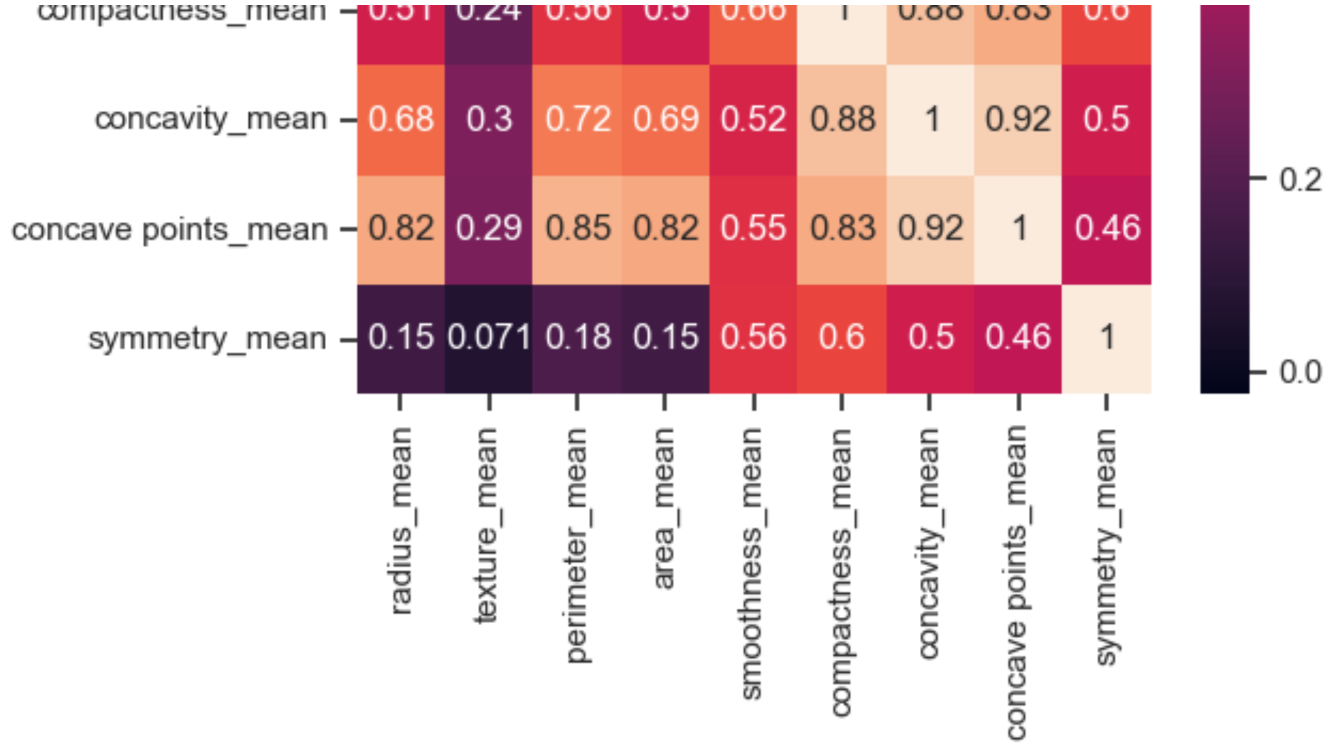

area_mean
smoothness_mean
compactness_mean
concavity_mean
concave points_mean
symmetry_mean
fractal_dimension_mean
radius_se
texture_se
perimeter_se
area_se
smoothness_se
compactness_se
concavity_se
concave points_se
symmetry_se
fractal_dimension_se
radius_worst
texture_worst
perimeter_worst
area_worst
smoothness_worst
compactness_worst
concavity_worst
0.8
0.6
0.4
0.2
0.0



In [24]:
```python
#correlation matrix of mean features
plt.figure(figsize=(6,6))
sns.heatmap(df[mean_features].corr(),annot=True)
```

Out[24]:
```
<AxesSubplot:>
```





Splitting data into training & test data.

In [25]:
```
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.25, random_state=42)
print("Shape of training data:", X_train.shape)
print("Shape of testing data:", X_test.shape)
```

```
Shape of training data: (426, 31)
Shape of testing data: (143, 31)
```

FEATURES SELECTION (ON 11,20,30 FEATURES)

1)Removing highly correlated features & Randomly selecting features:

one or more important highly correlated features are removed to reduce multicollinearity. However, this could lead to loss of important information and hence, lower accuracy.

1) Training the model with low and randomly correlated 11 features

In [26]:
```
Low_corr_vars = ['radius_mean','symmetry_mean', 'compactness_mean','texture_mean','compactness_worst',
                 'symmetry_se','concavity_mean','texture_worst','concavity_se',
                 'symmetry_worst','fractal_dimension_se']
```

In [27]:
```
X_train = X_train[Low_corr_vars]
X_test = X_test[Low_corr_vars]

print("Shape of X_train:", X_train.shape)
print("Shape of X_test:", X_test.shape)
```

```
Shape of X_train: (426, 11)
Shape of X_test: (143, 11)
```

Building the Neural Network

In [28]:
```
#setting up layers of Neural network
model = Sequential()
model.add(Dense(16, input_dim=11, activation= 'relu'))
model.add(Dropout(0.2))
model.add(Dense(1))
model.add(Activation('sigmoid'))

#Compiling the neural network
model.compile(loss='binary_crossentropy', optimizer='adam', metrics = ['accuracy'])

print(model.summary())
```

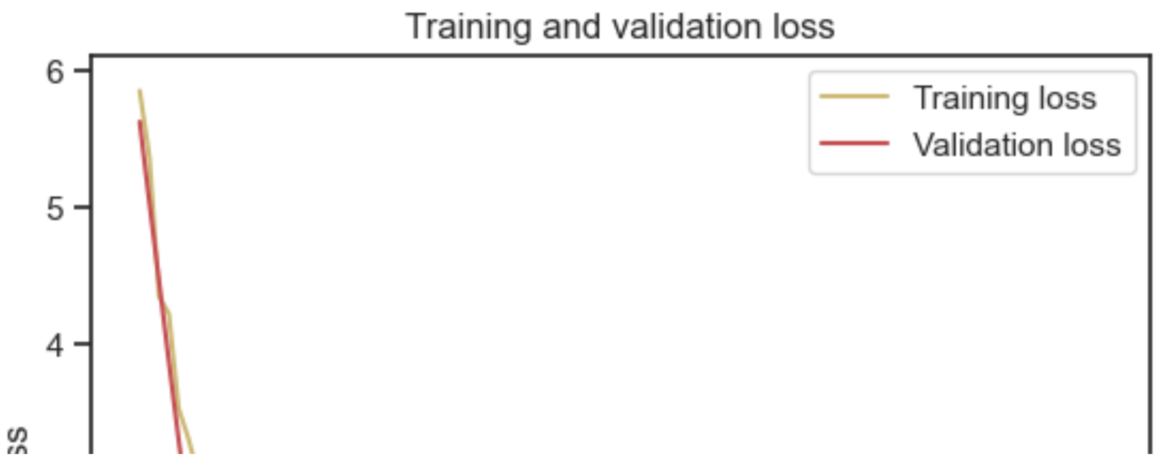
```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 dense (Dense)               (None, 16)                192       
                                                                 
 dropout (Dropout)           (None, 16)                0         
                                                                 
 dense_1 (Dense)             (None, 1)                 17        
                                                                 
 activation (Activation)     (None, 1)                 0         
                                                                 
=================================================================
Total params: 209
Trainable params: 209
Non-trainable params: 0
_________________________________________________________________
None
```

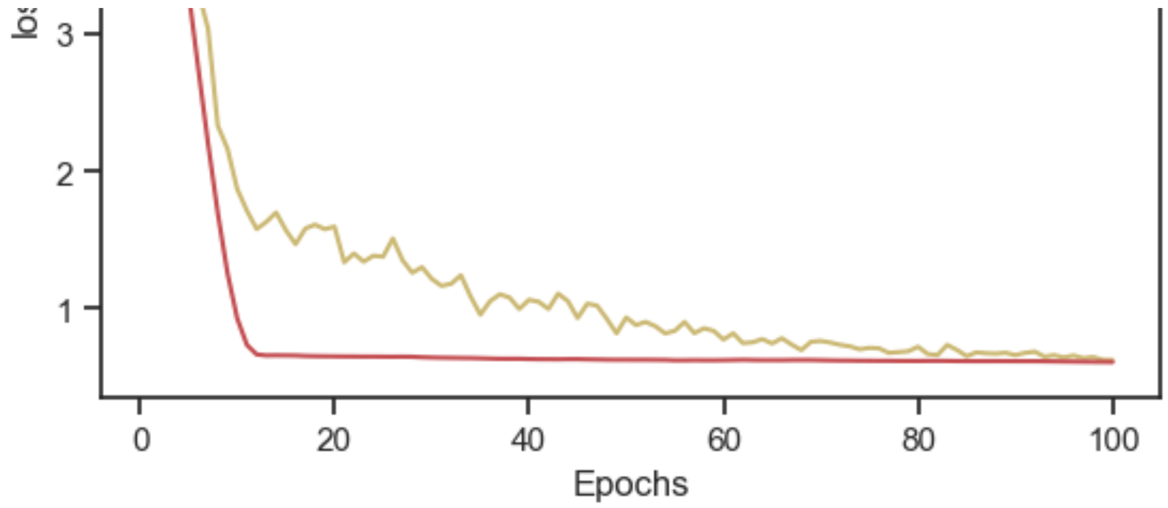
In [29]:
```
#training the neural network
history = model.fit(X_train, y_train, verbose=1, epochs=100, batch_size=64, validation_data=(X_test, y_t
```

```
7/7 [==============================] - 0s 10ms/step - loss: 1.3786 - accuracy: 0.5493 - val_loss: 0.650
3 - val_accuracy: 0.6364
Epoch 26/100
7/7 [==============================] - 0s 9ms/step - loss: 1.5111 - accuracy: 0.5376 - val_loss: 0.6498
- val_accuracy: 0.6364
Epoch 27/100
7/7 [==============================] - 0s 9ms/step - loss: 1.3523 - accuracy: 0.5516 - val_loss: 0.6503
- val_accuracy: 0.6364
Epoch 28/100
7/7 [==============================] - 0s 7ms/step - loss: 1.2617 - accuracy: 0.5657 - val_loss: 0.6495
- val_accuracy: 0.6364
Epoch 29/100
7/7 [==============================] - 0s 8ms/step - loss: 1.3021 - accuracy: 0.5376 - val_loss: 0.6462
- val_accuracy: 0.6364
Epoch 30/100
7/7 [==============================] - 0s 9ms/step - loss: 1.2171 - accuracy: 0.5516 - val_loss: 0.6442
- val_accuracy: 0.6294
Epoch 31/100
7/7 [==============================] - 0s 9ms/step - loss: 1.1661 - accuracy: 0.5751 - val_loss: 0.6428
- val_accuracy: 0.6364
```

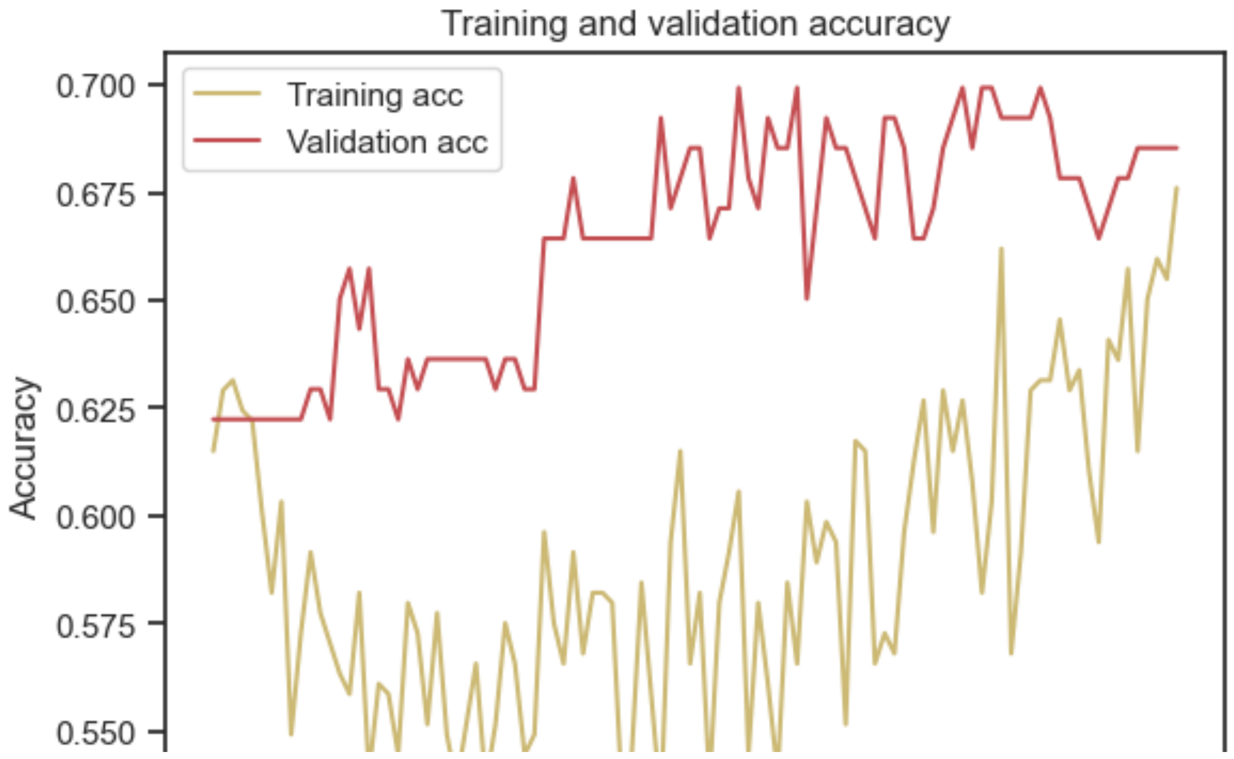
In [30]:

```
#plot training and validation accuracy and loss at each epoch
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(loss) + 1)
plt.plot(epochs, loss, 'y',label='Training loss')
plt.plot(epochs, val_loss, 'r', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('loss')
plt.legend()
plt.show()


acc = history.history['accuracy']
val_acc = history.history['val_accuracy']
plt.plot(epochs, acc, 'y',label='Training acc')
plt.plot(epochs, val_acc, 'r', label='Validation acc')
plt.title('Training and validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

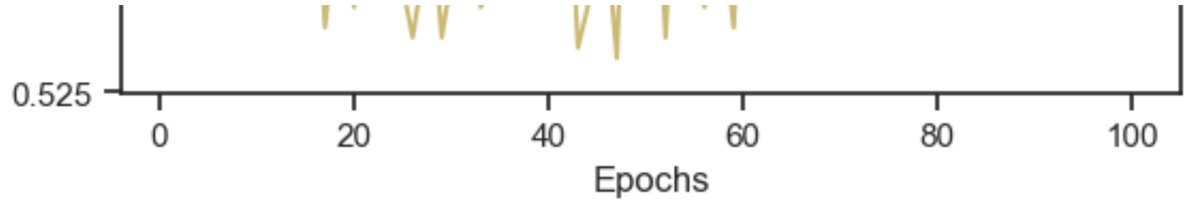




los
3
2
1
0
20
40
60
80
100
Epochs



Training and validation accuracy
Training acc
Validation acc
Accuracy
0.700
0.675
0.650
0.625
0.600
0.575
0.550

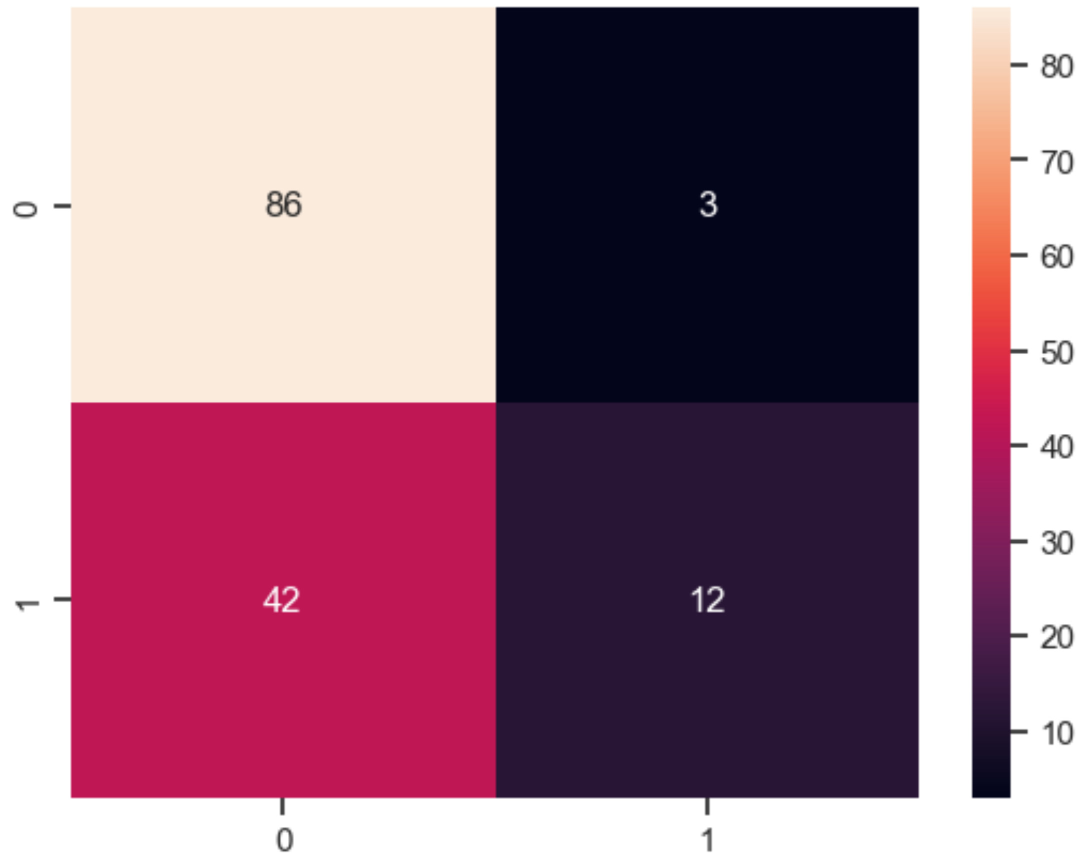


In [31]:
```
#predicting the test set results
y_pred = model.predict(X_test)


y_pred = (y_pred > 0.5)
```

```
5/5 [==============================] - 0s 2ms/step
```

In [32]:
```
#confusion matrix

from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)

sns.heatmap(cm, annot=True)
```

Out[32]:
```
<AxesSubplot:>
```



IMPACT OF LOW CORRELATED FEATURES

-The validation loss of a model is increased as these features might not be relevant to the target variable and can introduce noise to the model. -This can negatively impact the model's ability to generalize to new data, resulting in a higher validation loss. --Using low correlated features decreases the accuracy of the model, as these features may not be able to capture the important patterns and relationships in the data leading to the model performing poorly in predicting the target variable. -Using low correlated features led to an imbalanced confusion matrix, which skewed the model's classification towards one class with more false positives or false negatives. This can make it challenging to accurately classify the different classes and can result in a less balanced confusion matrix.

In [33]:
```
from sklearn import metrics
accuracy = metrics.accuracy_score(y_test, y_pred)
```

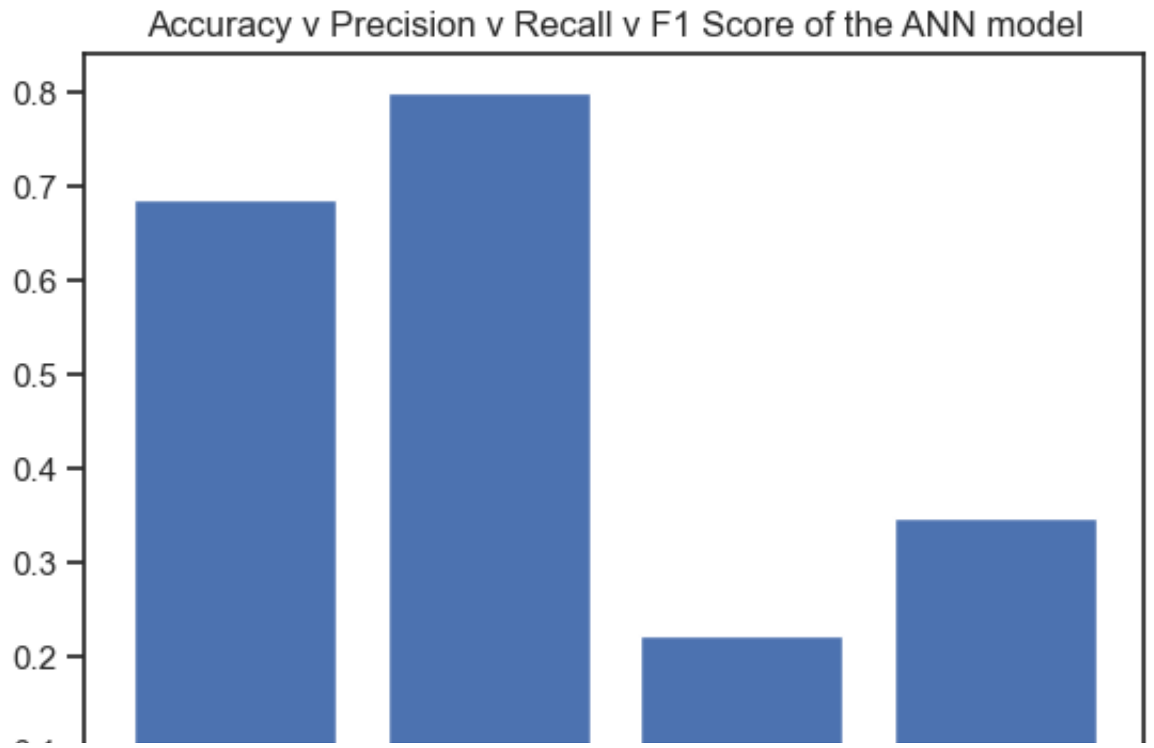
In [34]:
```
#Evaluation of model
from sklearn import metrics
accuracy=metrics.accuracy_score(y_test,y_pred)
print ("The accuracy is %.2f" % accuracy)

#print the classification report
c_report=metrics.classification_report(y_test, y_pred)
print (c_report)
```

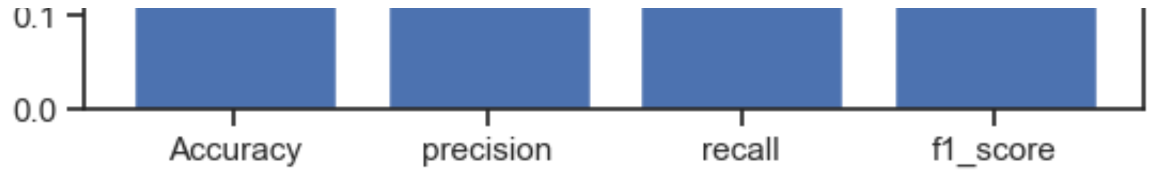
```
The accuracy is 0.69
              precision    recall  f1-score   support

           0       0.67      0.97      0.79        89
           1       0.80      0.22      0.35        54

    accuracy                           0.69       143
   macro avg       0.74      0.59      0.57       143
weighted avg       0.72      0.69      0.62       143
```

Visualizing the Evaluation Result with low correlated values

In [35]:

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
Accuracy = accuracy_score(y_test, y_pred)
# y_true is the true labels and y_pred is the corresponding predicted labels
tn, fp, fn, tp = confusion_matrix(y_test, y_pred).ravel()
precision = tp / (tp + fp)
recall = tp / (tp + fn)
f1_score = 2 * precision * recall / (precision + recall)
Eval_Metrics = [Accuracy, precision, recall, f1_score]
Metric_Names = ['Accuracy', 'precision', 'recall', 'f1_score']
Metrics_pos = np.arange(len(Metric_Names))
plt.bar(Metrics_pos, Eval_Metrics)
plt.xticks(Metrics_pos, Metric_Names)
plt.title('Accuracy v Precision v Recall v F1 Score of the ANN model')
plt.show()
```





IMPROVING THE MODEL

In [36]:
```
#Standardizing the dataset
from sklearn.preprocessing import MinMaxScaler
```

2) Training the model with highly correlated 20 features & Standardizing the dataset

In [37]:
```

High_corr_vars = ['radius_mean', 'perimeter_mean', 'area_mean', 'compactness_mean', 'concavity_mean',
                  'concave points_mean', 'radius_se', 'area_se','radius_worst', 'perimeter_worst',
                  'compactness_worst','smoothness_mean','symmetry_mean','perimeter_se','concavity_se',
                  'smoothness_worst','concavity_worst','area_worst','compactness_se','smoothness_mean']
x = df[High_corr_vars].values
```

In [38]:
```
scaler = MinMaxScaler()
scaler.fit(x)
x = scaler.transform(x)
print (x)
```

```
[[0.52103744 0.54598853 0.36373277 ... 0.45069799 0.35139844 0.59375282]
 [0.64314449 0.61578329 0.50159067 ... 0.43521431 0.08132304 0.28987993]
 [0.60149557 0.59574321 0.44941676 ... 0.37450845 0.2839547  0.51430893]
 ...
 [0.45525108 0.44578813 0.30311771 ... 0.23073142 0.26330099 0.28816467]
 [0.64456434 0.66553797 0.4757158  ... 0.402035   0.44557936 0.58833619]
 [0.03686876 0.02853984 0.01590668 ... 0.02049744 0.01808514 0.        ]]
```

In [39]:
```
#splitting to train and test set
X_train, X_test, y_train, y_test = train_test_split(x, Y, test_size=0.25, random_state=42)
```

In [40]:
```
print("Shape of X_train:", X_train.shape)
print("Shape of X_test:", X_test.shape)
```

```
Shape of X_train: (426, 20)
Shape of X_test: (143, 20)
```

Building the neural network with: -Number of layers: 2

-Neurons per layer: Input layer: 20 Hidden layer: 16 Output layer: 1

-Activation function: Hidden layer: 'relu' Output layer: 'sigmoid'

-Dropout rate: 0.2 for the dropout layer

-Loss function: 'binary_crossentropy'

-Optimizer: 'adam'

In [41]:
```
#setting up layers of Neural network
model = Sequential()
model.add(Dense(16, input_dim=20, activation= 'relu'))
model.add(Dropout(0.2))
model.add(Dense(1))
model.add(Activation('sigmoid'))

#Compiling the neural network
model.compile(loss='binary_crossentropy', optimizer='adam', metrics = ['accuracy'])

print(model.summary())
```

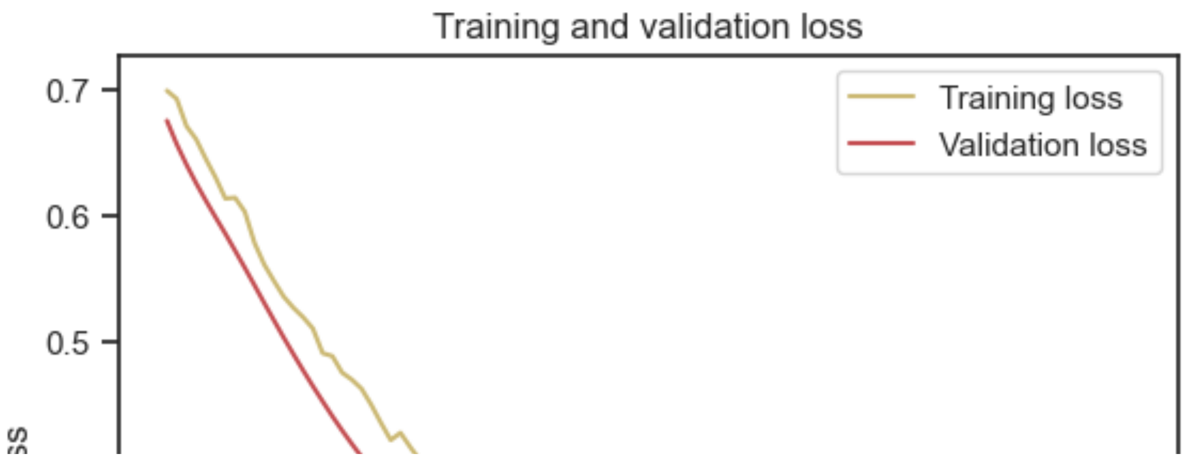
```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 dense_2 (Dense)             (None, 16)                336       
                                                                 
 dropout_1 (Dropout)         (None, 16)                0         
                                                                 
 dense_3 (Dense)             (None, 1)                 17        
                                                                 
 activation_1 (Activation)   (None, 1)                 0         
                                                                 
=================================================================
Total params: 353
Trainable params: 353
Non-trainable params: 0
_________________________________________________________________
None
```

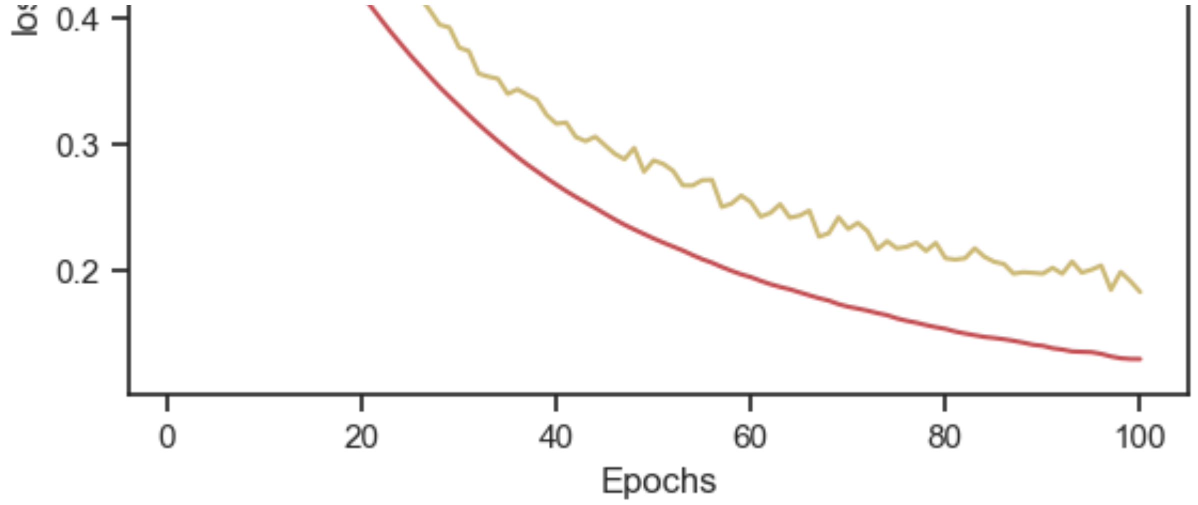
In [42]:
```
history = model.fit(X_train, y_train, verbose=1, epochs=100, batch_size=64, validation_data=(X_test, y_t
```

```
Epoch 29/100
7/7 [==============================] - 0s 10ms/step - loss: 0.3925 - accuracy: 0.8826 - val_loss: 0.337
6 - val_accuracy: 0.9371
Epoch 30/100
7/7 [==============================] - 0s 12ms/step - loss: 0.3764 - accuracy: 0.9108 - val_loss: 0.330
3 - val_accuracy: 0.9371
Epoch 31/100
7/7 [==============================] - 0s 10ms/step - loss: 0.3738 - accuracy: 0.9085 - val_loss: 0.322
9 - val_accuracy: 0.9371
Epoch 32/100
7/7 [==============================] - 0s 15ms/step - loss: 0.3559 - accuracy: 0.9131 - val_loss: 0.315
8 - val_accuracy: 0.9371
Epoch 33/100
7/7 [==============================] - 0s 13ms/step - loss: 0.3535 - accuracy: 0.9061 - val_loss: 0.309
0 - val_accuracy: 0.9371
Epoch 34/100
7/7 [==============================] - 0s 11ms/step - loss: 0.3518 - accuracy: 0.9014 - val_loss: 0.302
4 - val_accuracy: 0.9371
Epoch 35/100
7/7 [==============================] - 0s 13ms/step - loss: 0.3399 - accuracy: 0.9155 - val_loss: 0.296
```

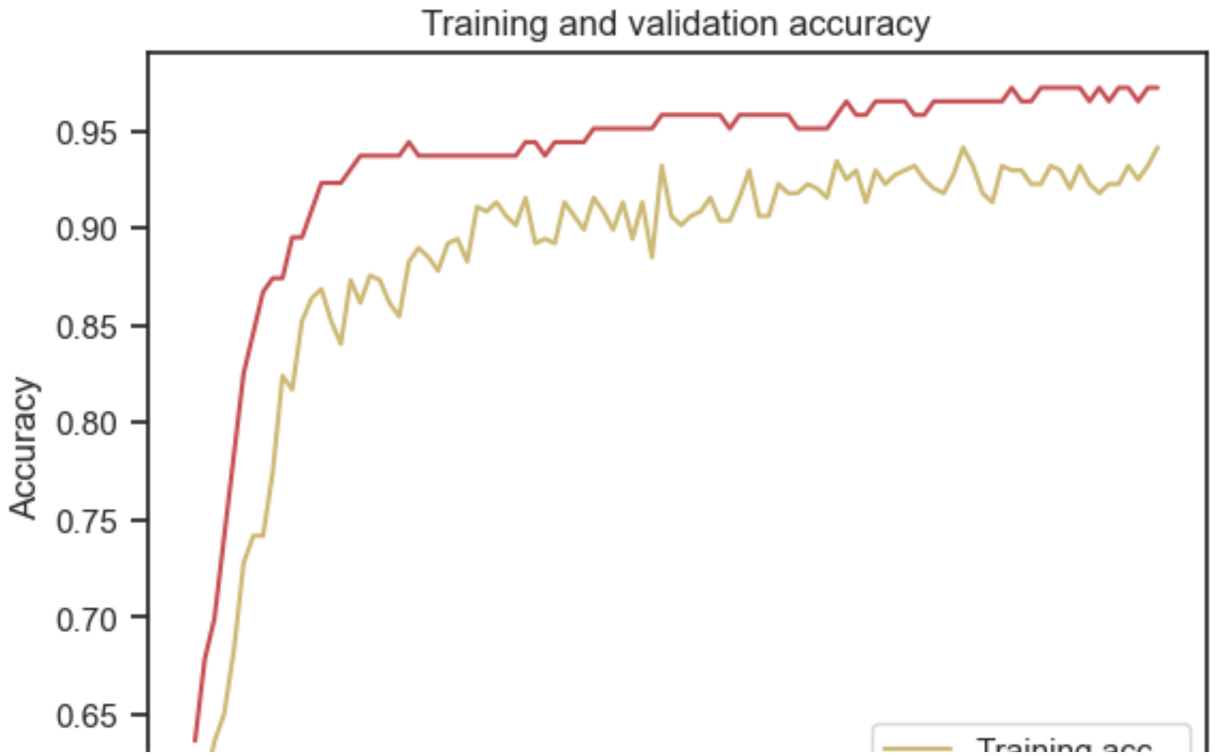
In [43]:

```
#plot training and validation accuracy and loss at each epoch
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(loss) + 1)
plt.plot(epochs, loss, 'y',label='Training loss')
plt.plot(epochs, val_loss, 'r', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('loss')
plt.legend()
plt.show()


acc = history.history['accuracy']
val_acc = history.history['val_accuracy']
plt.plot(epochs, acc, 'y',label='Training acc')
plt.plot(epochs, val_acc, 'r', label='Validation acc')
plt.title('Training and validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

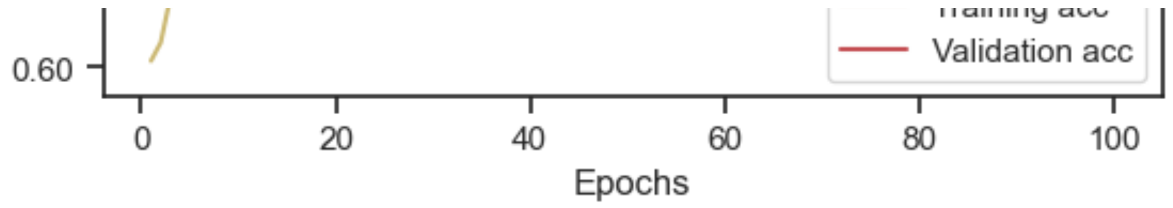




los
0.4
0.3
0.2
0
20
40
60
80
100
Epochs



Training and validation accuracy
Accuracy
0.95
0.90
0.85
0.80
0.75
0.70
0.65
Training acc

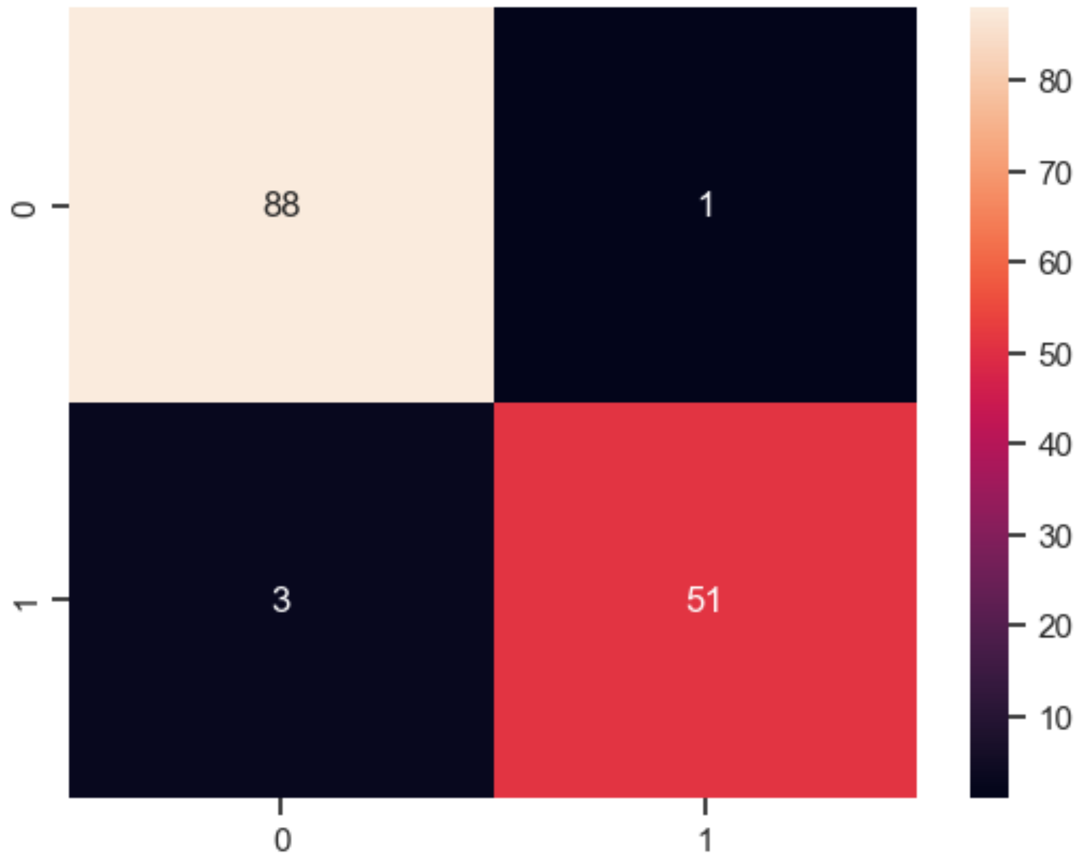


In [44]:
```
#predicting the test set results
y_pred = model.predict(X_test)


y_pred = (y_pred > 0.5)
```

```
5/5 [==============================] - 0s 2ms/step
```

In [45]:
```python
#confusion matrix

from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)

sns.heatmap(cm, annot=True)
```

Out[45]:
```
<AxesSubplot:>
```



In [46]:
```
accuracy = accuracy_score(y_test, y_pred)
print("Testing accuracy:", accuracy)
```

```
Testing accuracy: 0.972027972027972
```

In [47]:
```
training_accuracy = history.history['accuracy'][-1]
print("Training accuracy:", training_accuracy)
```

```
Training accuracy: 0.9413145780563354
```

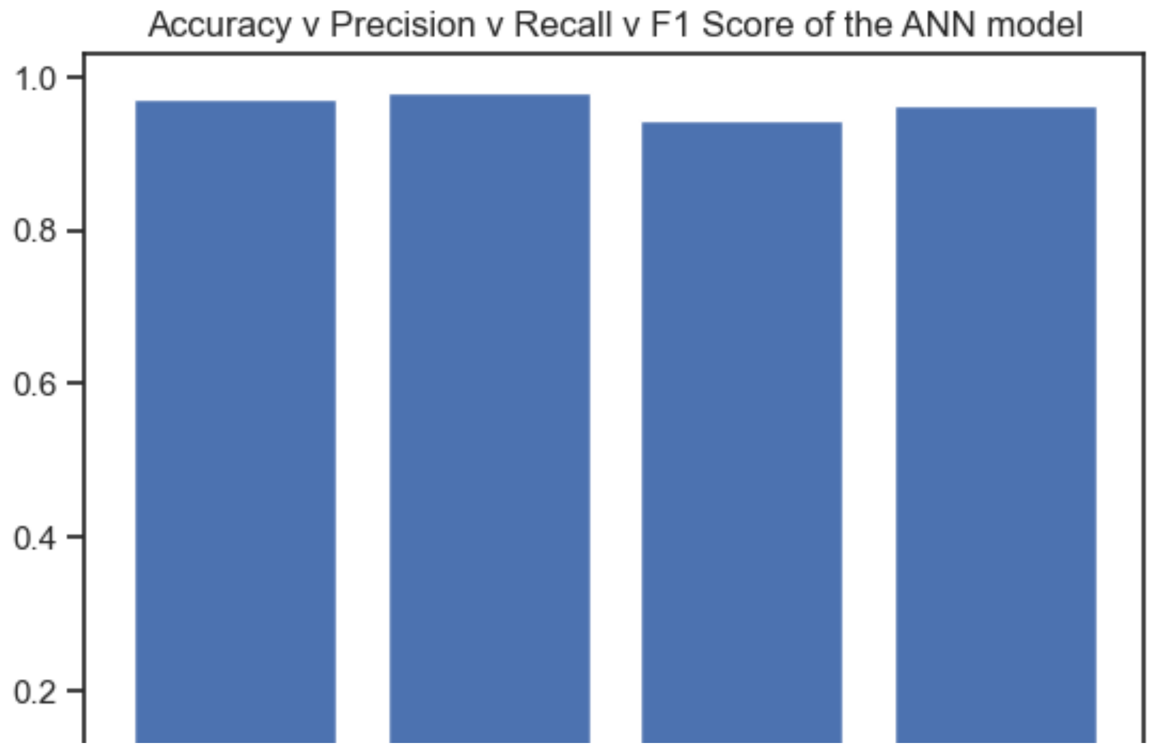
The testing data accuracy is higher than the accuracy on the training data, it suggests that the model is good at making predictions on new and unseen data, and not just the data that it was trained on. However, having a high testing accuracy is not a guarantee that the model is perfect or will work well in all situations. It's possible that the model may have been overfitted to the testing data or may perform poorly on data that it has not seen before. To ensure the model's reliability, I evaluated its performance using different evaluation metrics, rather than making any hasty conclusions about its performance.

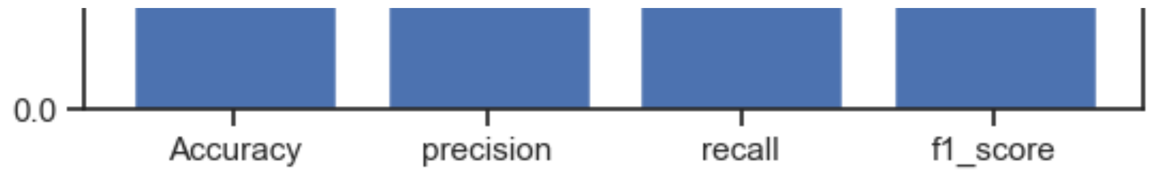
In [48]:
```python
#Evaluation of model
from sklearn import metrics
accuracy=metrics.accuracy_score(y_test,y_pred)
print ("The accuracy is %.2f" % accuracy)

#print the classification report
c_report=metrics.classification_report(y_test, y_pred)
print (c_report)
```

```
The accuracy is 0.97
              precision    recall  f1-score   support

           0       0.97      0.99      0.98        89
           1       0.98      0.94      0.96        54

    accuracy                           0.97       143
   macro avg       0.97      0.97      0.97       143
weighted avg       0.97      0.97      0.97       143
```

In [49]:

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
Accuracy = accuracy_score(y_test, y_pred)
# y_true is the true labels and y_pred is the corresponding predicted labels
tn, fp, fn, tp = confusion_matrix(y_test, y_pred).ravel()
precision = tp / (tp + fp)
recall = tp / (tp + fn)
f1_score = 2 * precision * recall / (precision + recall)
Eval_Metrics = [Accuracy, precision, recall, f1_score]
Metric_Names = ['Accuracy', 'precision', 'recall', 'f1_score']
Metrics_pos = np.arange(len(Metric_Names))
plt.bar(Metrics_pos, Eval_Metrics)
plt.xticks(Metrics_pos, Metric_Names)
plt.title('Accuracy v Precision v Recall v F1 Score of the ANN model')
plt.show()
```





3) TRAINING THE MODEL WITH 30 FEATURES

In [50]:
```
#Define input variable as P
P = df.drop(labels=["label", "id"], axis=1)
print(P.describe().T)
```

```
                         count        mean         std         min  \
radius_mean              569.0   14.127292    3.524049    6.981000
texture_mean             569.0   19.289649    4.301036    9.710000
perimeter_mean           569.0   91.969033   24.298981   43.790000
area_mean                569.0  654.889104  351.914129  143.500000
smoothness_mean          569.0    0.096360    0.014064    0.052630
compactness_mean         569.0    0.104341    0.052813    0.019380
concavity_mean           569.0    0.088799    0.079720    0.000000
concave points_mean      569.0    0.048919    0.038803    0.000000
symmetry_mean            569.0    0.181162    0.027414    0.106000
fractal_dimension_mean   569.0    0.062798    0.007060    0.049960
radius_se                569.0    0.405172    0.277313    0.111500
texture_se               569.0    1.216853    0.551648    0.360200
perimeter_se             569.0    2.866059    2.021855    0.757000
area_se                  569.0   40.337079   45.491006    6.802000
smoothness_se            569.0    0.007041    0.003003    0.001713
compactness_se           569.0    0.025478    0.017908    0.002252
concavity_se             569.0    0.031894    0.030186    0.000000
concave points_se        569.0    0.011796    0.006170    0.000000
symmetry_se              569.0    0.020542    0.008266    0.007882
fractal_dimension_se     569.0    0.003795    0.002646    0.000895
radius_worst             569.0   16.269190    4.833242    7.930000
```

```
texture_worst            569.0   25.677223    6.146258   12.020000
perimeter_worst          569.0  107.261213   33.602542   50.410000
area_worst               569.0  880.583128  569.356993  185.200000
smoothness_worst         569.0    0.132369    0.022832    0.071170
compactness_worst        569.0    0.254265    0.157336    0.027290
concavity_worst          569.0    0.272188    0.208624    0.000000
concave points_worst     569.0    0.114606    0.065732    0.000000
symmetry_worst           569.0    0.290076    0.061867    0.156500
fractal_dimension_worst  569.0    0.083946    0.018061    0.055040
Unnamed: 32                0.0         NaN         NaN         NaN

                                25%         50%          75%         max
radius_mean               11.700000   13.370000    15.780000    28.11000
texture_mean              16.170000   18.840000    21.800000    39.28000
perimeter_mean            75.170000   86.240000   104.100000   188.50000
area_mean                420.300000  551.100000   782.700000  2501.00000
smoothness_mean            0.086370    0.095870     0.105300     0.16340
compactness_mean           0.064920    0.092630     0.130400     0.34540
concavity_mean             0.029560    0.061540     0.130700     0.42680
concave points_mean        0.020310    0.033500     0.074000     0.20120
symmetry_mean              0.161900    0.179200     0.195700     0.30400
fractal_dimension_mean     0.057700    0.061540     0.066120     0.09744
radius_se                  0.232400    0.324200     0.478900     2.87300
texture_se                 0.833900    1.108000     1.474000     4.88500
perimeter_se               1.606000    2.287000     3.357000    21.98000
area_se                   17.850000   24.530000    45.190000   542.20000
smoothness_se              0.005169    0.006380     0.008146     0.03113
compactness_se             0.013080    0.020450     0.032450     0.13540
concavity_se               0.015090    0.025890     0.042050     0.39600
concave points_se          0.007638    0.010930     0.014710     0.05279
symmetry_se                0.015160    0.018730     0.023480     0.07895
fractal_dimension_se       0.002248    0.003187     0.004558     0.02984
radius_worst              13.010000   14.970000    18.790000    36.04000
texture_worst             21.080000   25.410000    29.720000    49.54000
perimeter_worst           84.110000   97.660000   125.400000   251.20000
```

```
area_worst              515.300000  686.500000  1084.000000  4254.00000
smoothness_worst          0.116600    0.131300     0.146000     0.22260
compactness_worst         0.147200    0.211900     0.339100     1.05800
concavity_worst           0.114500    0.226700     0.382900     1.25200
concave points_worst      0.064930    0.099930     0.161400     0.29100
symmetry_worst            0.250400    0.282200     0.317900     0.66380
fractal_dimension_worst   0.071460    0.080040     0.092080     0.20750
Unnamed: 32                    NaN         NaN          NaN         NaN
```

In [51]:
```
#standardize input data
scaled = MinMaxScaler()
scaled.fit(P)
P = scaled.transform(P)
print (P)
```

```
[[0.52103744 0.0226581  0.54598853 ... 0.59846245 0.41886396        nan]
 [0.64314449 0.27257355 0.61578329 ... 0.23358959 0.22287813        nan]
 [0.60149557 0.3902604  0.59574321 ... 0.40370589 0.21343303        nan]
 ...
 [0.45525108 0.62123774 0.44578813 ... 0.12872068 0.1519087         nan]
 [0.64456434 0.66351031 0.66553797 ... 0.49714173 0.45231536        nan]
 [0.03686876 0.50152181 0.02853984 ... 0.25744136 0.10068215        nan]]

/opt/anaconda3/lib/python3.9/site-packages/sklearn/preprocessing/_data.py:461: RuntimeWarning: All-NaN
slice encountered
  data_min = np.nanmin(X, axis=0)
/opt/anaconda3/lib/python3.9/site-packages/sklearn/preprocessing/_data.py:462: RuntimeWarning: All-NaN
slice encountered
  data_max = np.nanmax(X, axis=0)
```

In [52]:
```
#split training and test data
X_train, X_test, y_train, y_test = train_test_split(P, Y, test_size=0.25, random_state=42)
print("Shape of training data:", X_train.shape)
print("Shape of testing data:", X_test.shape)
```

```
Shape of training data: (426, 31)
Shape of testing data: (143, 31)
```

In [53]:
```
#checking for the expected size of model
print("Expected input shape of model:", model.layers[0].input_shape)
```

```
Expected input shape of model: (None, 20)
```

In [54]:
```
#remove a variable to adjust input data size to match model size
X_train = X_train[:, :-1]
X_test = X_test[:, :-1]
```

In [55]:
```
print("Shape of X_train:", X_train.shape)
print("Shape of X_test:", X_test.shape)
```

```
Shape of X_train: (426, 30)
Shape of X_test: (143, 30)
```

Here are the hyperparameters used in the below code:

Number of layers: 3

Neurons per layer: Input layer: 30 Hidden layers: 32, 16 Output layer: 1

Activation function: Hidden layers: 'relu' Output layer: 'sigmoid'

Dropout rate: 0.5 for each dropout layer

Loss function: 'binary_crossentropy'

Optimizer: 'rmsprop'

Evaluation metric: 'accuracy'

verbose: 1

batch_size: 32

validation_split: 0.2

Epoch:50

In [56]:

```
#Build neural network model

import keras.backend as K
K.clear_session()

#setting up layers of Neural network
model = Sequential()
model.add(Dense(32, input_dim=30, activation= 'relu'))
model.add(Dropout(0.5))
model.add(Dense(16, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))

#Compiling the neural network
model.compile(loss='binary_crossentropy', optimizer='rmsprop', metrics = ['accuracy'])

print(model.summary())
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 32)                992

 dropout (Dropout)           (None, 32)                0

 dense_1 (Dense)             (None, 16)                528

 dropout_1 (Dropout)         (None, 16)                0

 dense_2 (Dense)             (None, 1)                 17

=================================================================
Total params: 1,537
Trainable params: 1,537
```

```
Non-trainable params: 0
_________________________________________________________________
None
```

In [57]:
```
historyy = model.fit(X_train, y_train, verbose=1, epochs=50, batch_size=32, validation_split=0.2)
```

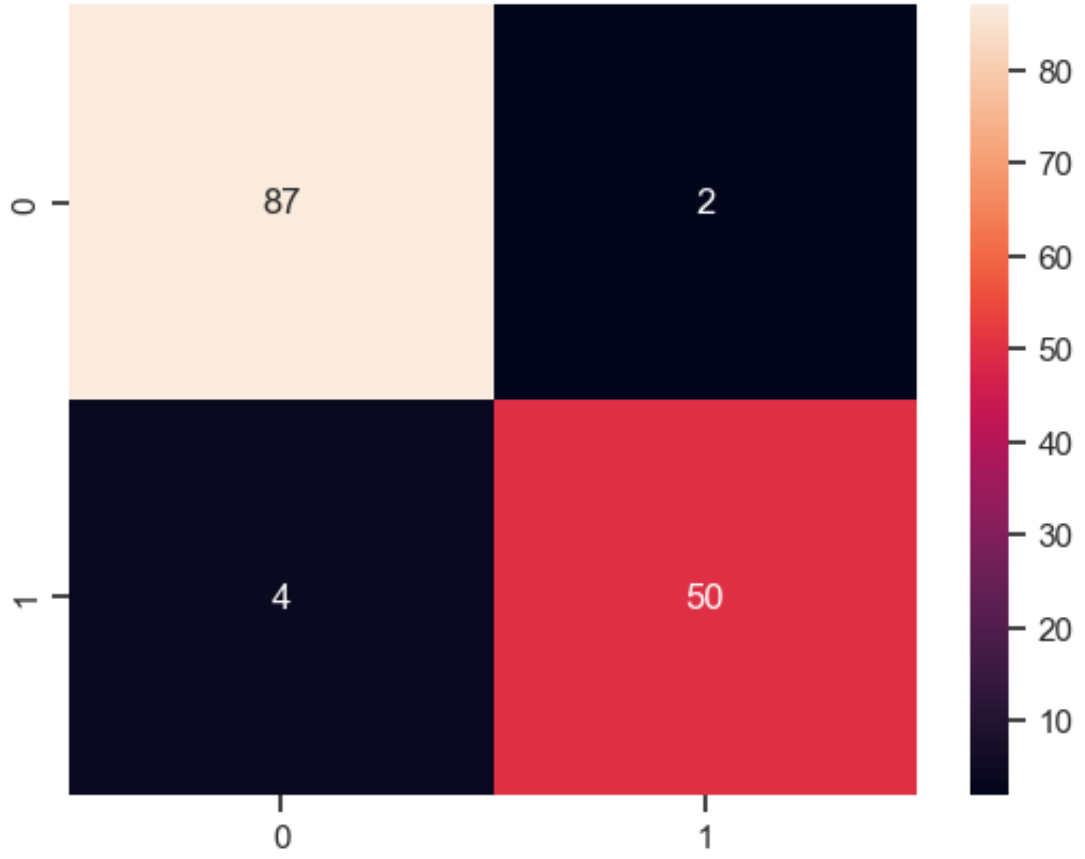
```
245 - val_accuracy: 0.8953
Epoch 11/50
11/11 [==============================] - 0s 8ms/step - loss: 0.5459 - accuracy: 0.7882 - val_loss: 0.50
27 - val_accuracy: 0.8953
Epoch 12/50
11/11 [==============================] - 0s 7ms/step - loss: 0.5400 - accuracy: 0.7941 - val_loss: 0.48
28 - val_accuracy: 0.9070
Epoch 13/50
11/11 [==============================] - 0s 8ms/step - loss: 0.5305 - accuracy: 0.7912 - val_loss: 0.46
28 - val_accuracy: 0.9070
Epoch 14/50
11/11 [==============================] - 0s 7ms/step - loss: 0.5374 - accuracy: 0.7735 - val_loss: 0.44
66 - val_accuracy: 0.9070
Epoch 15/50
11/11 [==============================] - 0s 8ms/step - loss: 0.5061 - accuracy: 0.8118 - val_loss: 0.43
11 - val_accuracy: 0.8953
Epoch 16/50
11/11 [==============================] - 0s 8ms/step - loss: 0.4620 - accuracy: 0.8118 - val_loss: 0.40
94 - val_accuracy: 0.9070
Epoch 17/50
```

In [58]:
```
#predicting the test set results
y_pred = model.predict(X_test)


y_pred = (y_pred > 0.5)
```

```
5/5 [==============================] - 0s 2ms/step
```

In [59]:
```
#confusion matrix

from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)

sns.heatmap(cm, annot=True)
```

Out[59]:
```
<AxesSubplot:>
```



In [60]:
```
accuracy = accuracy_score(y_test, y_pred)
print("Testing accuracy:", accuracy)
```

```
Testing accuracy: 0.958041958041958
```

In [61]:
```
training_accuracy = history.history['accuracy'][-1]
print("Training accuracy:", training_accuracy)
```

```
Training accuracy: 0.9413145780563354
```

In [62]:
```
#Evaluation of model
from sklearn import metrics
accuracy=metrics.accuracy_score(y_test,y_pred)
print ("The accuracy is %.2f" % accuracy)

#print the classification report
c_report=metrics.classification_report(y_test, y_pred)
print (c_report)
```

```
The accuracy is 0.96
              precision    recall  f1-score   support

           0       0.96      0.98      0.97        89
           1       0.96      0.93      0.94        54

    accuracy                           0.96       143
   macro avg       0.96      0.95      0.96       143
weighted avg       0.96      0.96      0.96       143
```

In [63]:
```
# Generating data for input
print("Shape of X_train:", X_train.shape)
print("Shape of X_test:", X_test.shape)
```

```
Shape of X_train: (426, 30)
Shape of X_test: (143, 30)
```

HYPERPARAMETERS TUNING

This consist of various parameters listed below

1.Number of layers: The number of layers in the neural network architecture.

2.Neurons per layer: The number of neurons in each layer of the neural network architecture.

3.Activation function: The activation function is used to introduce non-linearity to the output of each neuron in the neural network.

4.Dropout rate: Dropout is a regularization technique used to reduce overfitting in the neural network. Dropout layers randomly drop out a fraction of the neurons during training.

5.Loss function: The loss function is used to measure how well the neural network is performing during training. In this case, the binary cross-entropy loss function has been used as the data is binary classification.

6.Optimizer: The optimizer is the algorithm used to update the weights and biases of the neural network during training in order to minimize the loss function.

7.Verbose: 0, 1 or 2. Verbosity mode, 0 = silent, 1 = progress bar, 2 = one line per epoch.

8.Epochs: the number of times to iterate over the entire training dataset

9.Batch_size: the number of samples per batch of training data

10.Validation_split: fraction of the training data to use as validation data. For example, if set to 0.2, 20% of the training data will be used as validation data.

VARIOUS HYPERPARAMETER ACHICTECTURE

In [64]:
```

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.metrics import classification_report
from tensorflow.keras import regularizers
import tensorflow as tf

# Set the random seed for reproducibility
seed_value = 42
np.random.seed(seed_value)
tf.random.set_seed(seed_value)

# Architecture 1: 1 hidden layer with 4 neurons
model1 = Sequential()
model1.add(Dense(4, input_dim=30, activation='relu', kernel_regularizer=regularizers.l2(0.01)))
model1.add(Dropout(0.1))
model1.add(Dense(1, activation='sigmoid'))

# Architecture 2: 2 hidden layers with 4 and 8 neurons
model2 = Sequential()
model2.add(Dense(4, input_dim=30, activation='relu', kernel_regularizer=regularizers.l2(0.01)))
model2.add(Dropout(0.2))
model2.add(Dense(8, activation='relu'))
model2.add(Dense(1, activation='sigmoid'))

# Architecture 3: 2 hidden layers with 8 neurons each
model3 = Sequential()
model3.add(Dense(8, input_dim=30, activation='relu', kernel_regularizer=regularizers.l2(0.01)))
model3.add(Dropout(0.3))
model3.add(Dense(8, activation='relu'))
model3.add(Dense(1, activation='sigmoid'))

# Architecture 4: 3 hidden layers with 4, 8, and 16 neurons
model4 = Sequential()
model4.add(Dense(4, input_dim=30, activation='relu', kernel_regularizer=regularizers.l2(0.01)))
model4.add(Dropout(0.4))
model4.add(Dense(8, activation='relu'))
model4.add(Dropout(0.4))
```

```
model4.add(Dropout(0.4))
model4.add(Dense(16, activation='relu'))
model4.add(Dense(1, activation='sigmoid'))

# Architecture 5: 3 hidden layers with 8 neurons each
model5 = Sequential()
model5.add(Dense(8, input_dim=30, activation='relu', kernel_regularizer=regularizers.l2(0.01)))
model5.add(Dropout(0.5))
model5.add(Dense(8, activation='relu'))
model5.add(Dropout(0.5))
model5.add(Dense(8, activation='relu'))
model5.add(Dense(1, activation='sigmoid'))

# Define common hyperparameters
epochs = 10
batch_size = 16
verbose = 1
validation_split = 0.2

# Define the learning rates and dropout rates for each architecture
learning_rates = [0.001, 0.002, 0.003, 0.004, 0.005]
dropout_rates = [0.1, 0.2, 0.3, 0.4, 0.5]

# Compile and train each model with the corresponding learning rate and dropout rate
models = [model1, model2, model3, model4, model5]
for i, model in enumerate(models):
    learning_rate = learning_rates[i]
    dropout_rate = dropout_rates[i]
    optimizer = Adam(learning_rate=learning_rate)
    model.compile(loss='binary_crossentropy', optimizer=optimizer, metrics=['accuracy'])

    # Define early stopping callback
    early_stopping = EarlyStopping(monitor='val_loss', patience=3, restore_best_weights=True)

    print("Training Architecture", i+1)
    history = model.fit(X_train, y_train, epochs=epochs, batch_size=batch_size, verbose=verbose,
                        validation_split=validation_split, callbacks=[early_stopping])
                        validation_split=validation_split, callbacks=[early_stopping])

    # Print training and validation accuracy
    print("Training accuracy:", history.history['accuracy'][-1])
    print("Validation accuracy:", history.history['val_accuracy'][-1])

    # Evaluate the model
    test_loss, test_acc = model.evaluate(X_test, y_test)
    print('Test accuracy:', test_acc)

    # Make predictions on the test set
    y_pred = model.predict(X_test)
    y_pred = np.round(y_pred).flatten()

    # Compute classification report
    report = classification_report(y_test, y_pred)
    print(report)
```

```
Training Architecture 1
Epoch 1/10
22/22 [==============================] - 1s 12ms/step - loss: 0.7618 - accuracy: 0.5559 - val_loss: 0.7
457 - val_accuracy: 0.6163
Epoch 2/10
22/22 [==============================] - 0s 5ms/step - loss: 0.7382 - accuracy: 0.6235 - val_loss: 0.72
63 - val_accuracy: 0.6628
Epoch 3/10
22/22 [==============================] - 0s 6ms/step - loss: 0.7243 - accuracy: 0.6794 - val_loss: 0.70
74 - val_accuracy: 0.7558
Epoch 4/10
22/22 [==============================] - 0s 7ms/step - loss: 0.7085 - accuracy: 0.7265 - val_loss: 0.68
97 - val_accuracy: 0.7791
Epoch 5/10
22/22 [==============================] - 0s 7ms/step - loss: 0.6865 - accuracy: 0.7794 - val_loss: 0.67
25 - val_accuracy: 0.8372
Epoch 6/10
22/22 [==============================] - 0s 7ms/step - loss: 0.6732 - accuracy: 0.8059 - val_loss: 0.65
64 - val_accuracy: 0.8605
```

```
64 - val_accuracy: 0.8605
Epoch 7/10
```

Confusion matrix of varied architectures

PERFORMANCE METRICS

In [65]:
```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix
from tabulate import tabulate

# Define a function to plot the confusion matrix
def plot_confusion_matrix(ax, y_true, y_pred, title):
    cm = confusion_matrix(y_true, y_pred)
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', ax=ax)
    ax.set_title(title)
    ax.set_xlabel('Predicted Label')
    ax.set_ylabel('True Label')

# Train and evaluate each model
models = [model1, model2, model3, model4, model5]
accuracy_scores = []
precision_scores = []
recall_scores = []
f1_scores = []


fig, axs = plt.subplots(1, len(models), figsize=(20, 6))

for i, model in enumerate(models):
    print("Training Architecture", i+1)
    history = model.fit(X_train, y_train, epochs=epochs, batch_size=batch_size, verbose=verbose,
                        validation_split=validation_split, callbacks=[early_stopping])

    # Print training and validation accuracy
    print("Training accuracy:", history.history['accuracy'][-1])
    print("Validation accuracy:", history.history['val_accuracy'][-1])

    # Evaluate the model
    test_loss, test_acc = model.evaluate(X_test, y_test)
    print('Test accuracy:', test_acc)

    # Make predictions on the test set
    y_pred_prob = model.predict(X_test)
    y_pred = np.round(y_pred_prob).flatten()

    # Calculate accuracy, precision, recall, and F1 score
    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred)
    recall = recall_score(y_test, y_pred)
    f1 = f1_score(y_test, y_pred)

    # Append scores to lists
    accuracy_scores.append(accuracy)
    precision_scores.append(precision)
    recall_scores.append(recall)
    f1_scores.append(f1)

    # Plot the confusion matrix
    plot_confusion_matrix(axs[i], y_test, y_pred, title=f"Architecture {i+1}")

# Adjust the spacing between subplots
plt.tight_layout()

# Combine the scores and architectures into a table
table = zip(['Architecture 1', 'Architecture 2', 'Architecture 3', 'Architecture 4', 'Architecture 5'],
            accuracy_scores, precision_scores, recall_scores, f1_scores)

# Define the headers for the table
headers = ['Architecture', 'Accuracy', 'Precision', 'Recall', 'F1-Score']
```

```
# Print the table using tabulate
print(tabulate(table, headers=headers, tablefmt='grid'))

# Plot the performance metrics comparison
architectures = ['Architecture 1', 'Architecture 2', 'Architecture 3', 'Architecture 4', 'Architecture 5
x = np.arange(len(architectures))
width = 0.2

plt.figure(figsize=(10, 8))
plt.bar(x - 1.5 * width, accuracy_scores, width, label='Accuracy')
plt.bar(x - 0.5 * width, precision_scores, width, label='Precision')
plt.bar(x + 0.5 * width, recall_scores, width, label='Recall')
plt.bar(x + 1.5 * width, f1_scores, width, label='F1-Score')

plt.xlabel('Architecture')
plt.ylabel('Score')
plt.title('Performance Metrics Comparison')
plt.xticks(x, architectures)

# Adjust the position of the legend
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')

plt.show()
```

```
Training Architecture 1
Epoch 1/10
22/22 [==============================] - 0s 11ms/step - loss: 0.6067 - accuracy: 0.8529 - val_loss: 0.5
870 - val_accuracy: 0.8605
Epoch 2/10
22/22 [==============================] - 0s 7ms/step - loss: 0.5876 - accuracy: 0.8794 - val_loss: 0.57
51 - val_accuracy: 0.8721
Epoch 3/10
22/22 [==============================] - 0s 13ms/step - loss: 0.5875 - accuracy: 0.8706 - val_loss: 0.5
629 - val_accuracy: 0.8721
Epoch 4/10
```

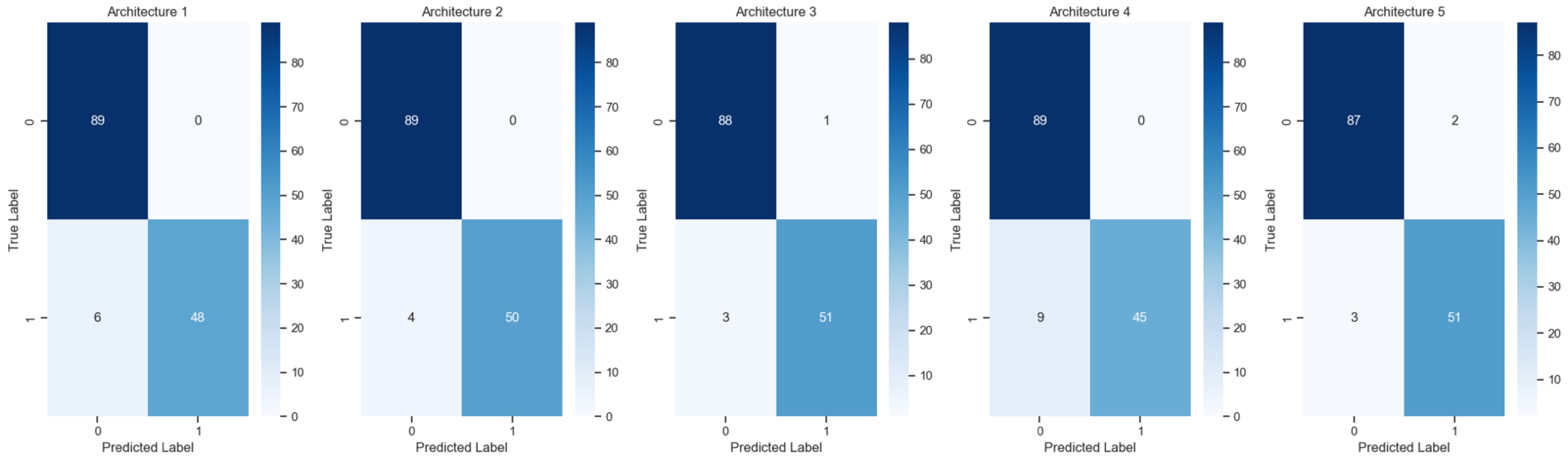
```
22/22 [==============================] - 0s 9ms/step - loss: 0.5687 - accuracy: 0.8794 - val_loss: 0.55
17 - val_accuracy: 0.8721
Epoch 5/10
22/22 [==============================] - 0s 7ms/step - loss: 0.5593 - accuracy: 0.8824 - val_loss: 0.54
03 - val_accuracy: 0.9070
Epoch 6/10
22/22 [==============================] - 0s 6ms/step - loss: 0.5443 - accuracy: 0.8824 - val_loss: 0.52
97 - val_accuracy: 0.9070
Epoch 7/10
22/22 [==============================] - 0s 8ms/step - loss: 0.5336 - accuracy: 0.9059 - val_loss: 0.51
92 - val_accuracy: 0.8953
Epoch 8/10
22/22 [==============================] - 0s 8ms/step - loss: 0.5338 - accuracy: 0.8676 - val_loss: 0.50
93 - val_accuracy: 0.8953
Epoch 9/10
22/22 [==============================] - 0s 6ms/step - loss: 0.5163 - accuracy: 0.8794 - val_loss: 0.49

93 - val_accuracy: 0.8953
Epoch 10/10
22/22 [==============================] - 0s 7ms/step - loss: 0.5114 - accuracy: 0.8765 - val_loss: 0.48
95 - val_accuracy: 0.8837
Training accuracy: 0.8764705657958984
Validation accuracy: 0.8837209343910217
5/5 [==============================] - 0s 3ms/step - loss: 0.4722 - accuracy: 0.9580
Test accuracy: 0.9580419659614563
5/5 [==============================] - 0s 3ms/step
Training Architecture 2
Epoch 1/10
22/22 [==============================] - 0s 18ms/step - loss: 0.4579 - accuracy: 0.8441 - val_loss: 0.3
926 - val_accuracy: 0.9186
Epoch 2/10
22/22 [==============================] - 0s 13ms/step - loss: 0.4396 - accuracy: 0.8500 - val_loss: 0.3
785 - val_accuracy: 0.8953
Epoch 3/10
22/22 [==============================] - 0s 17ms/step - loss: 0.4316 - accuracy: 0.8382 - val_loss: 0.3
529 - val_accuracy: 0.9419
```

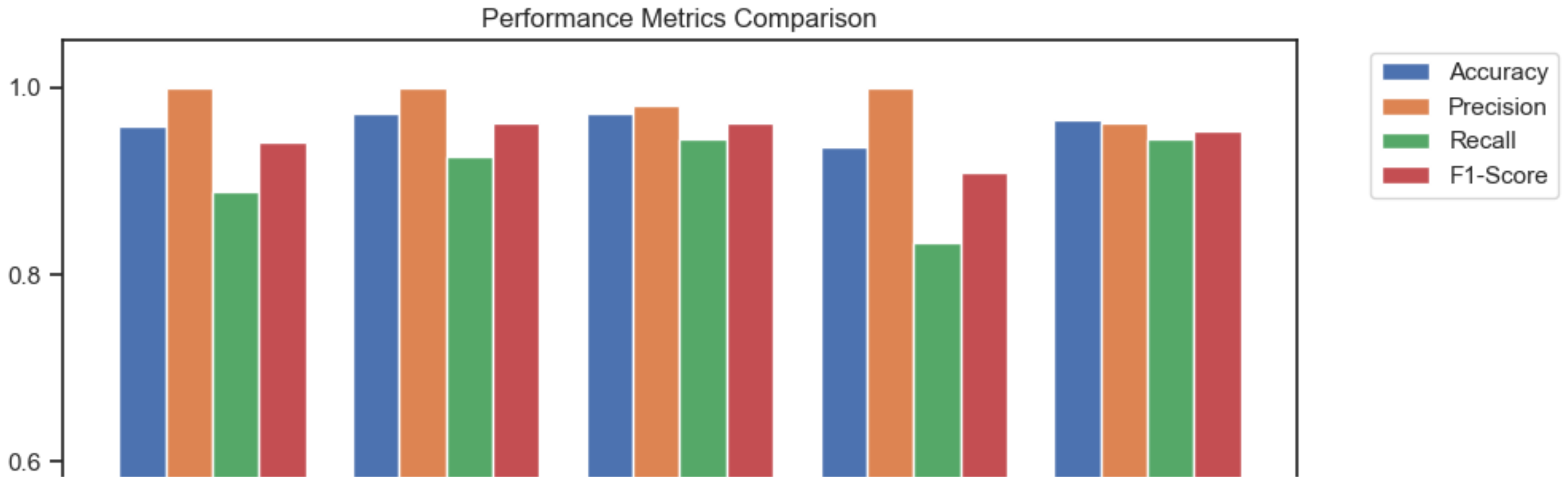
```
Epoch 4/10
22/22 [==============================] - 0s 12ms/step - loss: 0.4195 - accuracy: 0.8500 - val_loss: 0.3
389 - val_accuracy: 0.9186
Epoch 5/10
22/22 [==============================] - 0s 14ms/step - loss: 0.4058 - accuracy: 0.8471 - val_loss: 0.3
258 - val_accuracy: 0.9070
Epoch 6/10
22/22 [==============================] - 0s 13ms/step - loss: 0.3820 - accuracy: 0.8588 - val_loss: 0.3
049 - val_accuracy: 0.9535
Epoch 7/10
22/22 [==============================] - 0s 13ms/step - loss: 0.3890 - accuracy: 0.8676 - val_loss: 0.2
979 - val_accuracy: 0.9535
Epoch 8/10
22/22 [==============================] - 0s 11ms/step - loss: 0.3998 - accuracy: 0.8529 - val_loss: 0.2
893 - val_accuracy: 0.9419
Epoch 9/10

22/22 [==============================] - 0s 9ms/step - loss: 0.3655 - accuracy: 0.8676 - val_loss: 0.28
24 - val_accuracy: 0.9419
Epoch 10/10
22/22 [==============================] - 0s 13ms/step - loss: 0.3744 - accuracy: 0.8559 - val_loss: 0.2
740 - val_accuracy: 0.9419
Training accuracy: 0.8558823466300964
Validation accuracy: 0.9418604373931885
5/5 [==============================] - 0s 3ms/step - loss: 0.2464 - accuracy: 0.9720
Test accuracy: 0.9720279574394226
5/5 [==============================] - 0s 10ms/step
Training Architecture 3
Epoch 1/10
22/22 [==============================] - 0s 12ms/step - loss: 0.3083 - accuracy: 0.9235 - val_loss: 0.2
441 - val_accuracy: 0.9767
Epoch 2/10
22/22 [==============================] - 0s 12ms/step - loss: 0.3064 - accuracy: 0.9059 - val_loss: 0.2
382 - val_accuracy: 0.9419
Epoch 3/10
22/22 [==============================] - 0s 11ms/step - loss: 0.2562 - accuracy: 0.9412 - val_loss: 0.1
976 - val_accuracy: 0.9884
```

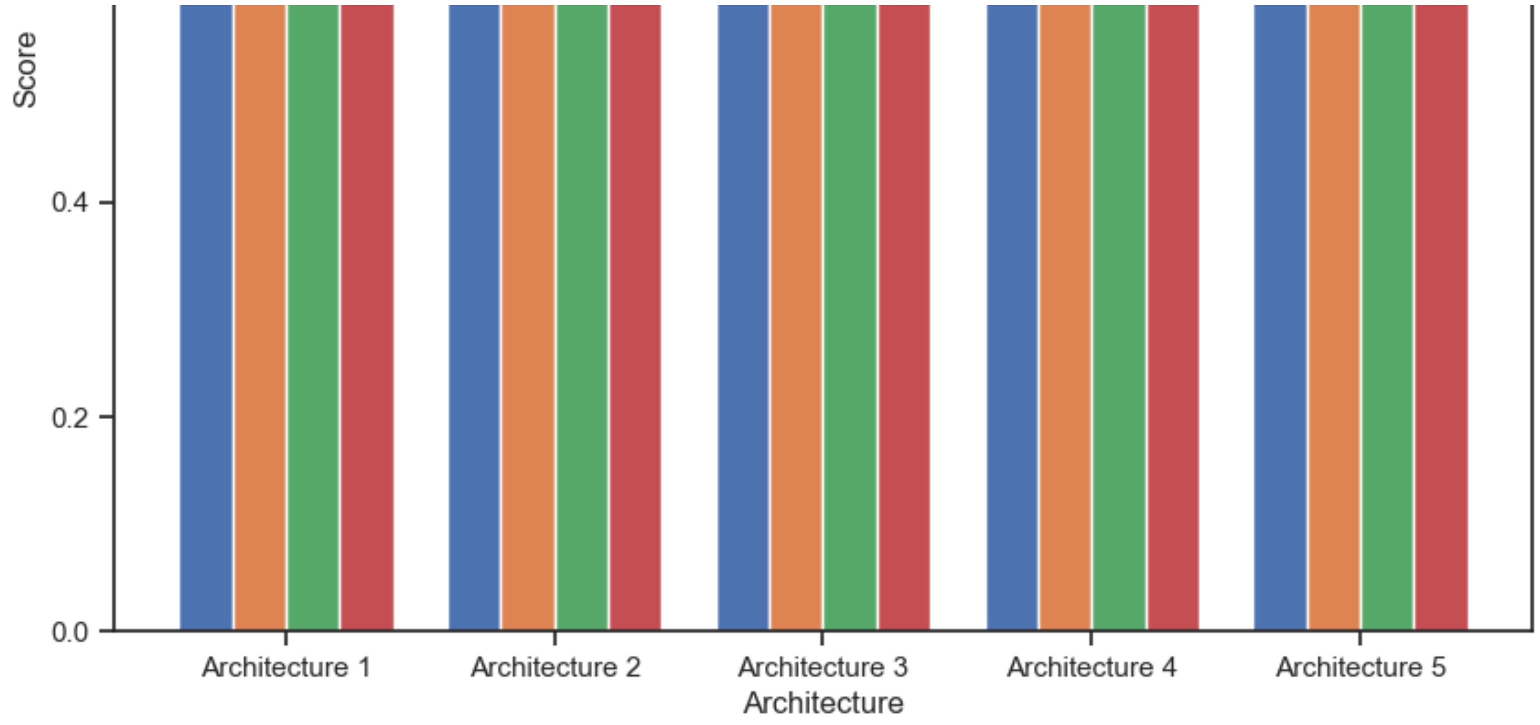
```
976 - val_accuracy: 0.9884
Epoch 4/10
22/22 [==============================] - 0s 15ms/step - loss: 0.2360 - accuracy: 0.9382 - val_loss: 0.1
819 - val_accuracy: 0.9884
Epoch 5/10
22/22 [==============================] - 0s 13ms/step - loss: 0.2590 - accuracy: 0.9235 - val_loss: 0.1
718 - val_accuracy: 0.9767
Epoch 6/10
22/22 [==============================] - 0s 10ms/step - loss: 0.2092 - accuracy: 0.9559 - val_loss: 0.1
627 - val_accuracy: 0.9651
Epoch 7/10
22/22 [==============================] - 0s 16ms/step - loss: 0.2034 - accuracy: 0.9471 - val_loss: 0.1
554 - val_accuracy: 0.9884
Epoch 8/10
22/22 [==============================] - 0s 17ms/step - loss: 0.2120 - accuracy: 0.9412 - val_loss: 0.1
485 - val_accuracy: 0.9884

Epoch 9/10
22/22 [==============================] - 0s 9ms/step - loss: 0.2435 - accuracy: 0.9088 - val_loss: 0.14
69 - val_accuracy: 0.9884
Epoch 10/10
22/22 [==============================] - 0s 11ms/step - loss: 0.2309 - accuracy: 0.9206 - val_loss: 0.1
438 - val_accuracy: 0.9651
Training accuracy: 0.9205882549285889
Validation accuracy: 0.9651162624359131
5/5 [==============================] - 0s 13ms/step - loss: 0.1232 - accuracy: 0.9720
Test accuracy: 0.9720279574394226
5/5 [==============================] - 0s 3ms/step
Training Architecture 4
Epoch 1/10
22/22 [==============================] - 0s 13ms/step - loss: 0.4741 - accuracy: 0.8088 - val_loss: 0.4
639 - val_accuracy: 0.8140
Epoch 2/10
22/22 [==============================] - 0s 12ms/step - loss: 0.4418 - accuracy: 0.8382 - val_loss: 0.4
641 - val_accuracy: 0.8023
Epoch 3/10
22/22 [==============================] - 0s 6ms/step - loss: 0.4858 - accuracy: 0.8059 - val_loss: 0.44
```

```
22/22 [==============================] - 0s 6ms/step - loss: 0.4858 - accuracy: 0.8059 - val_loss: 0.44
70 - val_accuracy: 0.8256
Epoch 4/10
22/22 [==============================] - 0s 8ms/step - loss: 0.4365 - accuracy: 0.8294 - val_loss: 0.37
73 - val_accuracy: 0.8721
Epoch 5/10
22/22 [==============================] - 0s 18ms/step - loss: 0.4929 - accuracy: 0.8000 - val_loss: 0.4
383 - val_accuracy: 0.8256
Epoch 6/10
22/22 [==============================] - 0s 9ms/step - loss: 0.4553 - accuracy: 0.8206 - val_loss: 0.40
25 - val_accuracy: 0.8488
Epoch 7/10
22/22 [==============================] - 0s 12ms/step - loss: 0.4628 - accuracy: 0.8176 - val_loss: 0.3
689 - val_accuracy: 0.8721
Epoch 8/10
22/22 [==============================] - 0s 13ms/step - loss: 0.4407 - accuracy: 0.8324 - val_loss: 0.4

389 - val_accuracy: 0.8256
Epoch 9/10
22/22 [==============================] - 0s 14ms/step - loss: 0.4840 - accuracy: 0.7971 - val_loss: 0.4
675 - val_accuracy: 0.8140
Epoch 10/10
22/22 [==============================] - 0s 10ms/step - loss: 0.4392 - accuracy: 0.8265 - val_loss: 0.4
204 - val_accuracy: 0.8256
Training accuracy: 0.8264706134796143
Validation accuracy: 0.8255813717842102
5/5 [==============================] - 0s 9ms/step - loss: 0.3030 - accuracy: 0.9371
Test accuracy: 0.9370629191398621
5/5 [==============================] - 0s 4ms/step
Training Architecture 5
Epoch 1/10
22/22 [==============================] - 0s 14ms/step - loss: 0.3136 - accuracy: 0.9000 - val_loss: 0.1
751 - val_accuracy: 0.9651
Epoch 2/10
22/22 [==============================] - 0s 16ms/step - loss: 0.3409 - accuracy: 0.8912 - val_loss: 0.2
082 - val_accuracy: 0.9302
```

```
Epoch 3/10
22/22 [==============================] - 0s 13ms/step - loss: 0.2787 - accuracy: 0.9206 - val_loss: 0.1
867 - val_accuracy: 0.9535
Epoch 4/10
22/22 [==============================] - 0s 10ms/step - loss: 0.3093 - accuracy: 0.8912 - val_loss: 0.1
591 - val_accuracy: 0.9767
Epoch 5/10
22/22 [==============================] - 0s 13ms/step - loss: 0.3298 - accuracy: 0.8971 - val_loss: 0.1
632 - val_accuracy: 0.9767
Epoch 6/10
22/22 [==============================] - 0s 12ms/step - loss: 0.2338 - accuracy: 0.9353 - val_loss: 0.1
480 - val_accuracy: 0.9884
Epoch 7/10
22/22 [==============================] - 0s 10ms/step - loss: 0.2415 - accuracy: 0.9235 - val_loss: 0.1
607 - val_accuracy: 0.9535
Epoch 8/10

22/22 [==============================] - 0s 11ms/step - loss: 0.2626 - accuracy: 0.9235 - val_loss: 0.1
690 - val_accuracy: 0.9302
Epoch 9/10
22/22 [==============================] - 0s 13ms/step - loss: 0.2757 - accuracy: 0.9118 - val_loss: 0.1
508 - val_accuracy: 0.9651
Training accuracy: 0.9117646813392639
Validation accuracy: 0.9651162624359131
5/5 [==============================] - 0s 10ms/step - loss: 0.1353 - accuracy: 0.9650
Test accuracy: 0.9650349617004395
5/5 [==============================] - 0s 5ms/step
+----------------+------------+-------------+----------+------------+
| Architecture   |   Accuracy |   Precision |   Recall |   F1-Score |
+================+============+=============+==========+============+
| Architecture 1 |   0.958042 |    1        | 0.888889 |   0.941176 |
+----------------+------------+-------------+----------+------------+
| Architecture 2 |   0.972028 |    1        | 0.925926 |   0.961538 |
+----------------+------------+-------------+----------+------------+
| Architecture 3 |   0.972028 |    0.980769 | 0.944444 |   0.962264 |
+----------------+------------+-------------+----------+------------+
| Architecture 4 |   0.937063 |    1         | 0.833333 |   0.909091 |
+----------------+------------+--------------+----------+------------+
| Architecture 5 |   0.965035 |    0.962264 | 0.944444 |   0.953271 |
+----------------+------------+--------------+----------+------------+
```

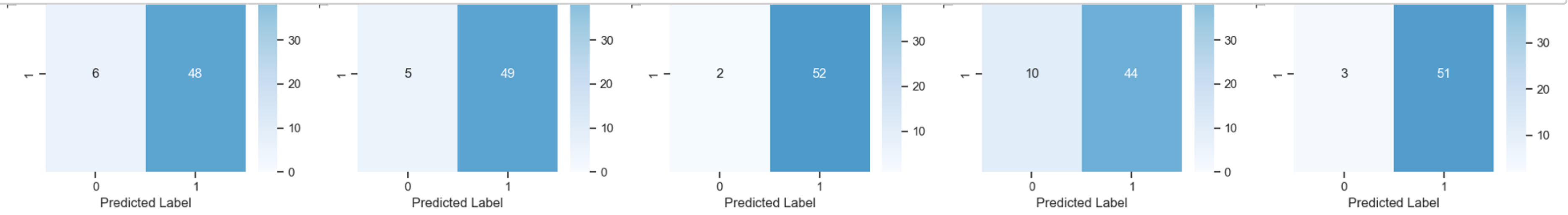


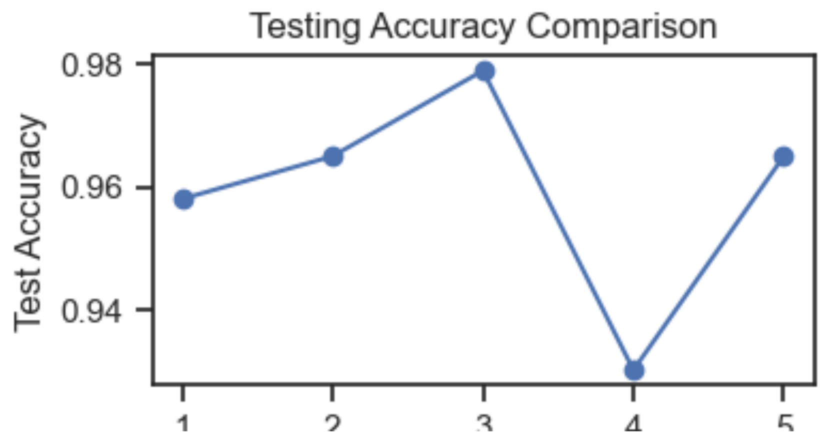




In [66]:
```
# Define a function to plot the confusion matrix
def plot_confusion_matrix(ax, y_true, y_pred, title):
    cm = confusion_matrix(y_true, y_pred)
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', ax=ax)
    ax.set_title(title)
    ax.set_xlabel('Predicted Label')
    ax.set_ylabel('True Label')

# Train and evaluate each model
models = [model1, model2, model3, model4, model5]
test_scores = []

fig, axs = plt.subplots(1, len(models), figsize=(20, 6))
```

```
for i, model in enumerate(models):
    print("Training Architecture", i+1)
    optimizer = Adam(learning_rate=learning_rates[i])
    model.compile(loss='binary_crossentropy', optimizer=optimizer, metrics=['accuracy'])

    history = model.fit(X_train, y_train, epochs=epochs, batch_size=batch_size, verbose=verbose,
                        validation_split=validation_split, callbacks=[early_stopping])

    # Print training and validation accuracy
    print("Training accuracy:", history.history['accuracy'][-1])
    print("Validation accuracy:", history.history['val_accuracy'][-1])

    # Evaluate the model
    test_loss, test_acc = model.evaluate(X_test, y_test)
    print('Test accuracy:', test_acc)
    test_scores.append(test_acc)

    # Make predictions on the test set
    y_pred = model.predict(X_test)
    y_pred_classes = np.where(y_pred > 0.5, 1, 0)

    # Plot the confusion matrix
    plot_confusion_matrix(axs[i], y_test, y_pred_classes, title=f"Architecture {i+1}")

    # Print precision, recall, and f1-score
    print(classification_report(y_test, y_pred_classes))

# Adjust the spacing between subplots
plt.tight_layout()

# Plot the testing scores comparison
plt.figure(figsize=(4, 2))
plt.plot(range(1, 6), test_scores, marker='o')
plt.xlabel('Architecture')
plt.ylabel('Test Accuracy')
plt.title('Testing Accuracy Comparison')
plt.xticks(range(1, 6))
plt.show()
```



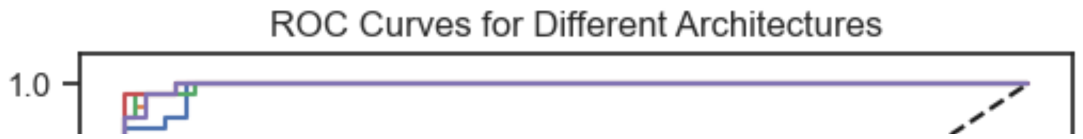


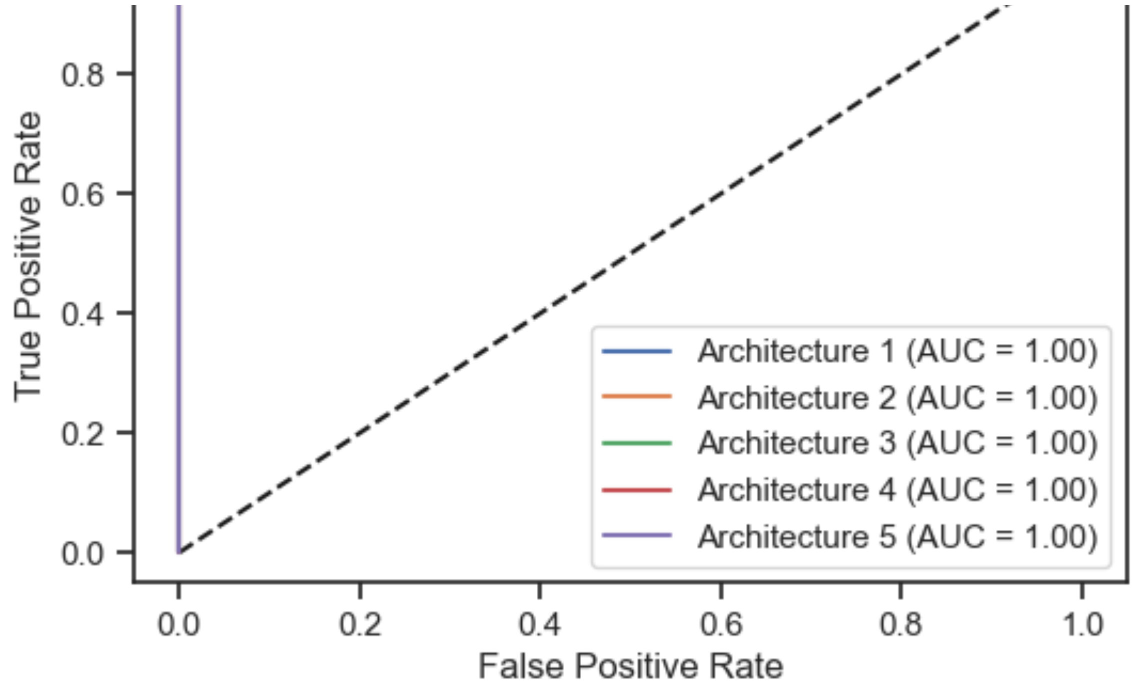
ROC CURVE FOR THE ARCHITECTURES

In [67]:

```
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, auc

# Function to plot the ROC curve
def plot_roc_curve(model, X_test, y_test, label):
    y_pred = model.predict(X_test).ravel()
    fpr, tpr, thresholds = roc_curve(y_test, y_pred)
    roc_auc = auc(fpr, tpr)
    plt.plot(fpr, tpr, label=f"{label} (AUC = {roc_auc:.2f})")

# Plot the ROC curves for all architectures
plt.figure(figsize=(6, 4))
plt.plot([0, 1], [0, 1], 'k--')

models = [model1, model2, model3, model4, model5]
labels = ["Architecture 1", "Architecture 2", "Architecture 3", "Architecture 4", "Architecture 5"]

for model, label in zip(models, labels):
    plot_roc_curve(model, X_test, y_test, label)

plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curves for Different Architectures')
plt.legend(loc='lower right')
plt.show()
```

```
5/5 [==============================] - 0s 2ms/step
5/5 [==============================] - 0s 2ms/step
5/5 [==============================] - 0s 2ms/step
5/5 [==============================] - 0s 2ms/step
5/5 [==============================] - 0s 2ms/step
```





IMPACT OF THE HYPERPARAMETERS TUNING

Increasing the number of hidden layers, neurons, epochs, and batch size can have varying effects and consequences on a neural network's performance, depending on its specific task, dataset, and architecture. Here are some general points to consider:

-More hidden layers can help the network learn complex and abstract representations of input data, but it may overfit and memorize the training data instead of generalizing to new examples. To prevent this the network should be validated on a separate set.

-More neurons per layer can increase the network's capacity, but it can also slow down training, increase overfitting, and lead to vanishing or exploding gradients. The number of neurons should be chosen based on the task's complexity and dataset size.

-Increasing epochs allows the network to see more examples and improve its parameters, but it can also lead to overfitting. Validation loss should be monitored, and early stopping techniques can be used.

-Larger batch sizes can speed up training and stabilize optimization, but they can also require more resources, reduce generalization performance, and increase the risk of getting stuck in local minima. Batch size should be chosen based on available resources and dataset/network characteristics.

OTHER MACHINE LEARNING ALGORITHMS ON BREAST CANCER DATASET

SUPPORT VECTOR MACHINE & NAIVE BAYES

Support Vector Machine (SVM) algorithm is used to classify or predict outcomes based on input data. SVM works by finding a straight line (called a hyperplane) that separates the input data into different categories or classes. In binary classification like ours, SVM finds the hyperplane that maximizes the distance between the hyperplane and the closest data points from each category.

Define input and target variable

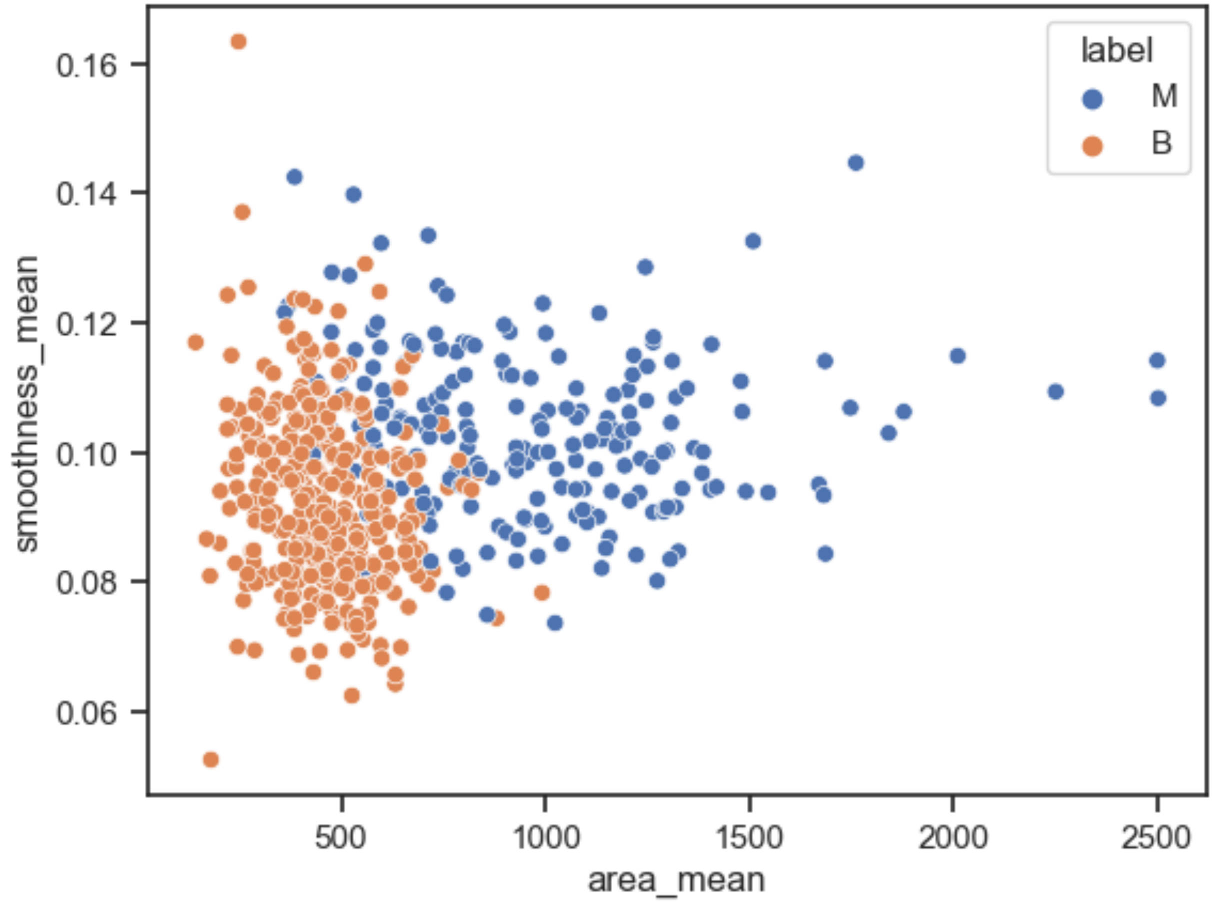
In [68]:
```
#dropping column "id", NaN values, and target "label"
X = df.drop(labels=["label", "id","Unnamed: 32"], axis=1)
```

In [69]:
```
Y = labelencoder.fit_transform(df["label"].values)
print("Label after encoding are: ", np.unique(Y))
```

```
Label after encoding are:  [0 1]
```

In [70]:
```
sns.scatterplot(x='area_mean',y='smoothness_mean',hue='label',data=df)
plt.ioff()
```

Out[70]:
```
<matplotlib.pyplot._IoffContext at 0x7ff14f8e7490>
```



In [71]:
```python




#splitting dataset to train and test set
X_train,X_test,y_train,y_test=train_test_split(X,Y,test_size=0.2,random_state=42)
```

In [72]:
```python
#check shape of train
X_train.shape
```

Out[72]:
```
(455, 30)
```

In [73]:
```python
#check shape of test
X_test.shape
```

Out[73]:
```
(114, 30)
```

SUPPORT VECTOR MACHINE MODEL BUILDING

In [74]:
```python
from sklearn.svm import SVC
from sklearn.metrics import classification_report,confusion_matrix, roc_auc_score, roc_curve
svc_model=SVC()
svc_model = SVC(probability=True)
# Fit the SVM model on the training data
svc_model.fit(X_train,y_train)
```

Out[74]:
```
SVC(probability=True)
```

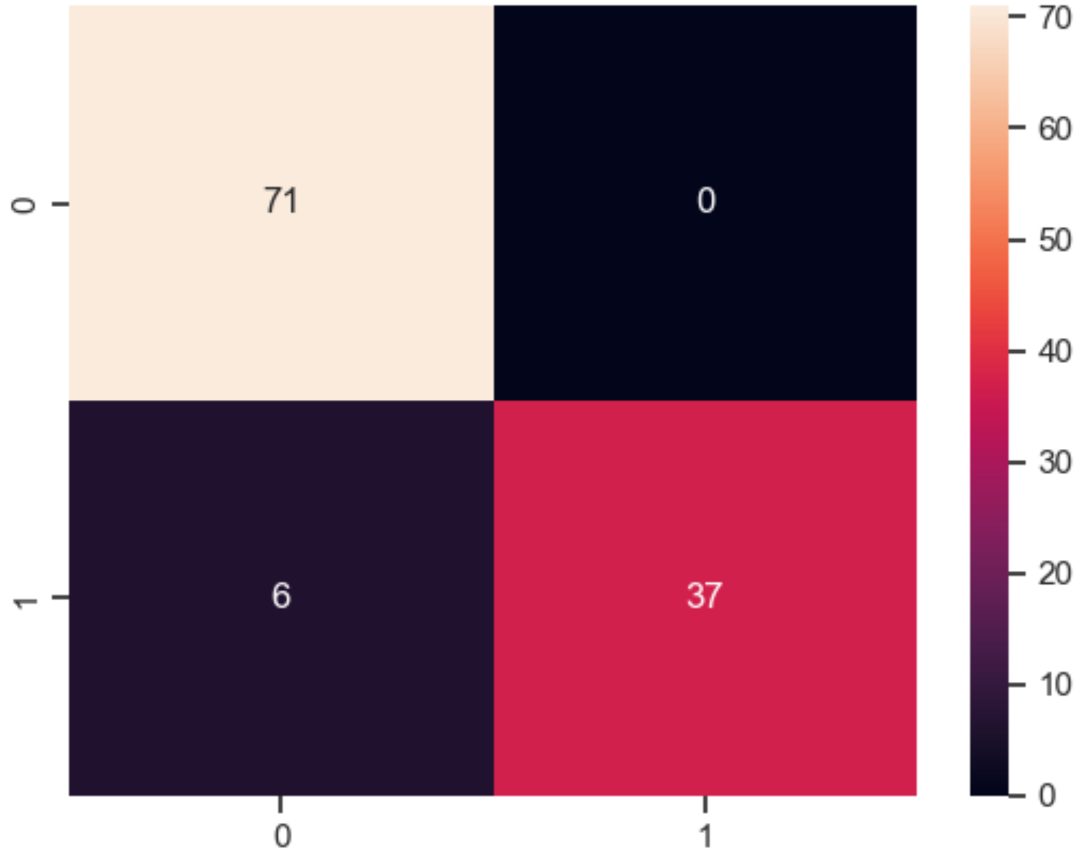
MODEL EVALUATION

In [75]:

```
# Generate predictions on the test set
y_predict = svc_model.predict(X_test)

# Compute the AUC score
auc = roc_auc_score(y_test, y_predict)
print("AUC Score: {:.2f}".format(auc))

# Print classification report
print(classification_report(y_test, y_predict))
```

```
AUC Score: 0.93
              precision    recall  f1-score   support

           0       0.92      1.00      0.96        71
           1       1.00      0.86      0.92        43

    accuracy                           0.95       114
   macro avg       0.96      0.93      0.94       114
weighted avg       0.95      0.95      0.95       114
```

```
In [76]: # Assuming y_test and y_predict are the true and predicted labels respectively
         cm = confusion_matrix(y_test, y_predict)

         # Plot the confusion matrix
         sns.heatmap(cm, annot=True)

         # Add axis labels and title

         plt.show()
```



MODEL OPTIMIZATION

In [77]:
```
#find best hyper parameters
from sklearn.model_selection import GridSearchCV
param_grid = {'C':[0.1,1,10,100,1000],'gamma':[1,0.1,0.01,0.001,0.001], 'kernel':['rbf']}
grid = GridSearchCV(SVC(),param_grid,verbose = 4)
grid.fit(X_train,y_train)
grid.best_params_
grid.best_estimator_
grid_predictions = grid.predict(X_test)
cmG = confusion_matrix(y_test,grid_predictions)
sns.heatmap(cmG, annot=True)
print(classification_report(y_test,grid_predictions))
```

```
Fitting 5 folds for each of 25 candidates, totalling 125 fits
[CV 1/5] END ........C=0.1, gamma=1, kernel=rbf;, score=0.637 total time=   0.1s
[CV 2/5] END ........C=0.1, gamma=1, kernel=rbf;, score=0.626 total time=   0.0s
[CV 3/5] END ........C=0.1, gamma=1, kernel=rbf;, score=0.626 total time=   0.0s
[CV 4/5] END ........C=0.1, gamma=1, kernel=rbf;, score=0.626 total time=   0.0s
[CV 5/5] END ........C=0.1, gamma=1, kernel=rbf;, score=0.626 total time=   0.0s
[CV 1/5] END ......C=0.1, gamma=0.1, kernel=rbf;, score=0.637 total time=   0.0s
[CV 2/5] END ......C=0.1, gamma=0.1, kernel=rbf;, score=0.626 total time=   0.0s
[CV 3/5] END ......C=0.1, gamma=0.1, kernel=rbf;, score=0.626 total time=   0.0s
[CV 4/5] END ......C=0.1, gamma=0.1, kernel=rbf;, score=0.626 total time=   0.0s
[CV 5/5] END ......C=0.1, gamma=0.1, kernel=rbf;, score=0.626 total time=   0.0s
[CV 1/5] END .....C=0.1, gamma=0.01, kernel=rbf;, score=0.637 total time=   0.0s
[CV 2/5] END .....C=0.1, gamma=0.01, kernel=rbf;, score=0.626 total time=   0.0s
[CV 3/5] END .....C=0.1, gamma=0.01, kernel=rbf;, score=0.626 total time=   0.0s
[CV 4/5] END .....C=0.1, gamma=0.01, kernel=rbf;, score=0.626 total time=   0.0s
[CV 5/5] END .....C=0.1, gamma=0.01, kernel=rbf;, score=0.626 total time=   0.0s
[CV 1/5] END ....C=0.1, gamma=0.001, kernel=rbf;, score=0.637 total time=   0.0s
[CV 2/5] END ....C=0.1, gamma=0.001, kernel=rbf;, score=0.626 total time=   0.0s
[CV 3/5] END ....C=0.1, gamma=0.001, kernel=rbf;, score=0.626 total time=   0.0s
[CV 4/5] END ....C=0.1, gamma=0.001, kernel=rbf;, score=0.626 total time=   0.0s
```

HYPERPARAMETER OPTIMIZATION

The last model improvement did not yield the percentage of accuracy. Hence, I created a machine learning pipeline that trains a Support Vector Machine (SVM) classifier using a linear kernel and C=1 hyperparameter . This is to classify new data as either one of two categories (binary classification). The dataset is preprocessed using the StandardScaler function, which standardizes the features by removing the mean and scaling to unit variance to ensure that all features have the same impact on the SVM model.

The SVM classifier used is LinearSVC, which finds the best hyperplane to separate the two classes in a high-dimensional space. The hyperplane is chosen to maximize the distance between the two classes, and the classifier is initialized with a C=1 hyperparameter to control the trade-off between maximizing the margin and minimizing the classification error.

In [78]:
```
#Building a pipeline using a linear classifier

from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.svm import LinearSVC

svm_clf = Pipeline([
    ("scaler", StandardScaler()),
    ("linear_svc", LinearSVC(C=1, loss="hinge", random_state=42))
])

svm_clf.fit(X_train, y_train)
```

Out[78]:
```
Pipeline(steps=[('scaler', StandardScaler()),
                ('linear_svc', LinearSVC(C=1, loss='hinge', random_state=42))])
```

In [79]:
```
#prediction on few values from training set:
predictions = svm_clf.predict(X_train.iloc[:5])
actual = y_train[:5]

print("Predictions\t", "Actual\t\t")
for index in range(len(predictions)):
    print(predictions[index], "\t\t", actual[index])
```

```
Predictions      Actual
0                0
1                1
0                0
0                0
0                0
```

In [80]:
```
#building model using linear classifier
from sklearn.svm import SVC

# hyperparameter C
C = 5
alpha = 1 / (C * len(X))

svm_clf = SVC(kernel="linear", C=C)
```

In [81]:
```
#standardizing input data
import numpy as np

scaler = StandardScaler()

# pre-process the train and test data
X_train_scaled = scaler.fit_transform(X_train.astype(np.float32))
X_test_scaled = scaler.transform(X_test.astype(np.float32))
```

In [82]:
```
# train the model
svm_clf.fit(X_train_scaled, y_train)
```

Out[82]:
```
SVC(C=5, kernel='linear')
```

In [83]:
```
print("SVC:                          ", svm_clf.intercept_, svm_clf.coef_)
```

```
SVC:                          [0.11182825] [[-0.14169589 -0.01603748 -0.41509059 -0.26099686 -0.1446300
9 -1.12303578
   0.89064088  2.40044064 -0.27473911  0.43838811  1.90485622 -0.28903005
  -0.73747757  1.12785897  0.61471225  0.12299309 -0.94256721  0.53829543
  -0.77244493 -1.30536627  1.68589978  1.35857697  0.05551069  1.57397528
  -0.05466699 -0.59138542  1.66515095 -0.04341149  1.34119342  0.45562077]]
```

MODEL EVALUATION ON TRAINING DATA

In [84]:
```
# function to print out classification model report
def classification_report(model_name, test, pred, label):
    from sklearn.metrics import precision_score, recall_score
    from sklearn.metrics import accuracy_score, f1_score, roc_auc_score

    print(model_name, ":\n")

    print("Accuracy Score: ", '{:,.3f}'.format(float(accuracy_score(test, pred)) * 100), "%")
    print("     Precision: ", '{:,.3f}'.format(float(precision_score(test, pred, pos_label=label)) * 100
    print("        Recall: ", '{:,.3f}'.format(float(recall_score(test, pred, pos_label=label)) * 100),
    print("      F1 score: ", '{:,.3f}'.format(float(f1_score(test, pred, pos_label=label)) * 100), "%")
    print("       AUC-ROC: ", '{:,.3f}'.format(float(roc_auc_score(test, pred)) * 100), "%")
```

In [85]:
```
svm_clf_pred = svm_clf.predict(X_train_scaled)
classification_report("SVM with linear kernel and C=5 Hyperparameter", y_train, svm_clf_pred, 0)
```

```
SVM with linear kernel and C=5 Hyperparameter :

Accuracy Score:  98.901 %
     Precision:  98.616 %
        Recall:  99.650 %
      F1 score:  99.130 %
        AUC-ROC:  98.642 %
```

SVM MODEL EVALUATION ON TEST DATA

In [86]:
```
svm_clf_pred_test = svm_clf.predict(X_test_scaled)
```

In [87]:
```
classification_report("SVM on Test Set", y_test, svm_clf_pred_test, 0)
```

```
SVM on Test Set :

Accuracy Score:  96.491 %
     Precision:  98.551 %
        Recall:  95.775 %
      F1 score:  97.143 %
        AUC-ROC:  96.725 %
```

NAIVE BAYES ALGORITHM ON BREAST CANCER DATASET

Define input and target variable

In [88]:
```
#dropping column "id", NaN values, and target "label"
X = df.drop(labels=["label", "id","Unnamed: 32"], axis=1)
Y = labelencoder.fit_transform(df["label"].values)
print("Label after encoding are: ", np.unique(Y))
```

```
Label after encoding are:  [0 1]
```

Dataset is normalized to adjust the scale of input features, so that they are all on a similar range. This helps to avoid situations where certain features have a larger impact on the model than others, which can cause the model to make biased predictions.

In [89]:
```
# Normalization of dataset
X = (X - np.min(X)) / (np.max(X) - np.min(X))
```

```
/opt/anaconda3/lib/python3.9/site-packages/numpy/core/fromnumeric.py:84: FutureWarning: In a future ver
sion, DataFrame.min(axis=None) will return a scalar min over the entire DataFrame. To retain the old be
havior, use 'frame.min(axis=0)' or just 'frame.min()'
  return reduction(axis=axis, out=out, **passkwargs)
/opt/anaconda3/lib/python3.9/site-packages/numpy/core/fromnumeric.py:84: FutureWarning: In a future ver
sion, DataFrame.max(axis=None) will return a scalar max over the entire DataFrame. To retain the old be
havior, use 'frame.max(axis=0)' or just 'frame.max()'
  return reduction(axis=axis, out=out, **passkwargs)
```

In [90]:
```
#Splitting data to train and test set
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size = 0.3, random_state = 42)
```

Bernoulli Naive Bayes

The Bernoulli Naive Bayes (BNB) algorithm is used to classify input data that consists of binary or boolean features, where each feature can take on one of two possible values: 0 or 1. The algorithm assumes that each feature is independent of all the other features given the class variable.

In [91]:

In [91]:

```
from sklearn.metrics import classification_report, roc_auc_score
from sklearn.naive_bayes import BernoulliNB

BNB = BernoulliNB()
BNB.fit(X_train, y_train)
y_pred = BNB.predict(X_test)
y_prob = BNB.predict_proba(X_test)[:, 1]

print("Accuracy:", BNB.score(X_test, y_test))
print(classification_report(y_test, y_pred, labels=[0, 1]))
print("AUC Score:", roc_auc_score(y_test, y_prob))
```

```
Accuracy: 0.631578947368421
              precision    recall  f1-score   support

           0       0.63      1.00      0.77       108
           1       0.00      0.00      0.00        63

    accuracy                           0.63       171
   macro avg       0.32      0.50      0.39       171
weighted avg       0.40      0.63      0.49       171

AUC Score: 0.5324074074074073

/opt/anaconda3/lib/python3.9/site-packages/sklearn/metrics/_classification.py:1318: UndefinedMetricWarn
ing: Precision and F-score are ill-defined and being set to 0.0 in labels with no predicted samples. Us
e `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/opt/anaconda3/lib/python3.9/site-packages/sklearn/metrics/_classification.py:1318: UndefinedMetricWarn
ing: Precision and F-score are ill-defined and being set to 0.0 in labels with no predicted samples. Us
e `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/opt/anaconda3/lib/python3.9/site-packages/sklearn/metrics/_classification.py:1318: UndefinedMetricWarn
ing: Precision and F-score are ill-defined and being set to 0.0 in labels with no predicted samples. Us
e `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
```

MODEL OPTIMIZATION OF THE NAIVE BAYES MODEL

Multinomial Naive Bayes

Multinomial Naive Bayes (MNB) algorithm is used to classify input data consisting of count or discrete data such as text classification. This means that the input features represent the count of a certain event or occurrence.

In [92]:
```
#Building the Multinomial Naive Bayes Classifier Model
from sklearn.naive_bayes import MultinomialNB
MNB = MultinomialNB()
MNB.fit(X_train, y_train)
MNB.score(X_test, y_test)
y_pred = MNB.predict(X_test)
y_prob = MNB.predict_proba(X_test)[:, 1]
print("Accuracy:", MNB.score(X_test, y_test))
print(classification_report(y_test, y_pred))
print("AUC Score:", roc_auc_score(y_test, y_prob))
```

```
Accuracy: 0.8304093567251462
              precision    recall  f1-score   support

           0       0.79      1.00      0.88       108
           1       1.00      0.54      0.70        63

    accuracy                           0.83       171
   macro avg       0.89      0.77      0.79       171
weighted avg       0.87      0.83      0.82       171

AUC Score: 0.9528218694885362
```

GUASSIAN NAIVE BAYES

Gaussian Naive Bayes (GNB) algorithm is used to classify input data where the continuous-valued features of each class are assumed to be normally distributed. GNB assumes that the distribution of the features is normal. GNB is commonly used when dealing with continuous data and when the distribution of the features is assumed to be Gaussian.

In [93]:
```
#Building the Naive Bayes Classifier Model
from sklearn.naive_bayes import GaussianNB
GNB = GaussianNB()
GNB.fit(X_train, y_train)
print("Naive Bayes score: ",GNB.score(X_test, y_test))
y_pred = GNB.predict(X_test)
y_prob = GNB.predict_proba(X_test)[:, 1]
print("Accuracy:", GNB.score(X_test, y_test))
print(classification_report(y_test, y_pred))
print("AUC Score:", roc_auc_score(y_test, y_prob))
```

```
Naive Bayes score:  0.935672514619883
Accuracy: 0.935672514619883
              precision    recall  f1-score   support

           0       0.94      0.95      0.95       108
           1       0.92      0.90      0.91        63

    accuracy                           0.94       171
   macro avg       0.93      0.93      0.93       171
weighted avg       0.94      0.94      0.94       171

AUC Score: 0.9926513815402704
```

REFERENCE

kaggle.com. (n.d.). Naive Bayes Implementation on Cancer Dataset. [online] Available at: https://www.kaggle.com/code/nisasoylu/naive-bayes-implementation-on-cancer-dataset (https://www.kaggle.com/code/nisasoylu/naive-bayes-implementation-on-cancer-dataset) [Accessed 14 May 2023].

Eren, M.E. (2020). Support Vector Machines on the Breast Cancer Wisconsin (Diagnostic) Data Set. [online] GitHub. Available at: https://github.com/MaksimEkin/Breast-Cancer-Prediction-SVM/blob/master/breast_cancer_prediction.ipynb (https://github.com/MaksimEkin/Breast-Cancer-Prediction-SVM/blob/master/breast_cancer_prediction.ipynb) [Accessed 14 May 2023].