**Lung cancer prediction**

*#Importing Libraries*

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

*#For ignoring warning*

import warnings

warnings.filterwarnings("ignore")

df=pd.read\_csv('../input/lung-cancer/survey lung cancer.csv')

df

| GENDER | AGE | SMOKING | YELLOW\_FINGERS | ANXIETY | PEER\_PRESSURE | CHRONIC DISEASE | FATIGUE | ALLERGY | WHEEZING | ALCOHOL CONSUMING | COUGHING | SHORTNESS OF BREATH | SWALLOWING DIFFICULTY | CHEST PAIN | LUNG\_CANCER |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | M | 69 | 1 | 2 | 2 | 1 | 1 | 2 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | YES |
| 1 | M | 74 | 2 | 1 | 1 | 1 | 2 | 2 | 2 | 1 | 1 | 1 | 2 | 2 | 2 | YES |
| 2 | F | 59 | 1 | 1 | 1 | 2 | 1 | 2 | 1 | 2 | 1 | 2 | 2 | 1 | 2 | NO |
| 3 | M | 63 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 2 | 1 | 1 | 2 | 2 | NO |
| 4 | F | 63 | 1 | 2 | 1 | 1 | 1 | 1 | 1 | 2 | 1 | 2 | 2 | 1 | 1 | NO |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 304 | F | 56 | 1 | 1 | 1 | 2 | 2 | 2 | 1 | 1 | 2 | 2 | 2 | 2 | 1 | YES |
| 305 | M | 70 | 2 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 2 | YES |
| 306 | M | 58 | 2 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 1 | 1 | 2 | YES |
| 307 | M | 67 | 2 | 1 | 2 | 1 | 1 | 2 | 2 | 1 | 2 | 2 | 2 | 1 | 2 | YES |
| 308 | M | 62 | 1 | 1 | 1 | 2 | 1 | 2 | 2 | 2 | 2 | 1 | 1 | 2 | 1 | YES |

309 rows × 16 columns

**Note: In this dataset, YES=2 & NO=1**

df.shape

(309, 16)

*#Checking for Duplicates*

df.duplicated().sum()

33

*#Removing Duplicates*

df=df.drop\_duplicates()

*#Checking for null values*

df.isnull().sum()

GENDER 0

AGE 0

SMOKING 0

YELLOW\_FINGERS 0

ANXIETY 0

PEER\_PRESSURE 0

CHRONIC DISEASE 0

FATIGUE 0

ALLERGY 0

WHEEZING 0

ALCOHOL CONSUMING 0

COUGHING 0

SHORTNESS OF BREATH 0

SWALLOWING DIFFICULTY 0

CHEST PAIN 0

LUNG\_CANCER 0

dtype: int64

df.info()

<class 'pandas.core.frame.DataFrame'>

Int64Index: 276 entries, 0 to 283

Data columns (total 16 columns):

# Column Non-Null Count Dtype

--- ------ -------------- -----

0 GENDER 276 non-null object

1 AGE 276 non-null int64

2 SMOKING 276 non-null int64

3 YELLOW\_FINGERS 276 non-null int64

4 ANXIETY 276 non-null int64

5 PEER\_PRESSURE 276 non-null int64

6 CHRONIC DISEASE 276 non-null int64

7 FATIGUE 276 non-null int64

8 ALLERGY 276 non-null int64

9 WHEEZING 276 non-null int64

10 ALCOHOL CONSUMING 276 non-null int64

11 COUGHING 276 non-null int64

12 SHORTNESS OF BREATH 276 non-null int64

13 SWALLOWING DIFFICULTY 276 non-null int64

14 CHEST PAIN 276 non-null int64

15 LUNG\_CANCER 276 non-null object

dtypes: int64(14), object(2)

memory usage: 36.7+ KB

df.describe()

|  | AGE | SMOKING | YELLOW\_FINGERS | ANXIETY | PEER\_PRESSURE | CHRONIC DISEASE | FATIGUE | ALLERGY | WHEEZING | ALCOHOL CONSUMING | COUGHING | SHORTNESS OF BREATH | SWALLOWING DIFFICULTY | CHEST PAIN |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| count | 276.000000 | 276.000000 | 276.000000 | 276.000000 | 276.000000 | 276.000000 | 276.000000 | 276.000000 | 276.000000 | 276.000000 | 276.000000 | 276.000000 | 276.000000 | 276.000000 |
| mean | 62.909420 | 1.543478 | 1.576087 | 1.496377 | 1.507246 | 1.521739 | 1.663043 | 1.547101 | 1.547101 | 1.550725 | 1.576087 | 1.630435 | 1.467391 | 1.557971 |
| std | 8.379355 | 0.499011 | 0.495075 | 0.500895 | 0.500856 | 0.500435 | 0.473529 | 0.498681 | 0.498681 | 0.498324 | 0.495075 | 0.483564 | 0.499842 | 0.497530 |
| min | 21.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |
| 25% | 57.750000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |
| 50% | 62.500000 | 2.000000 | 2.000000 | 1.000000 | 2.000000 | 2.000000 | 2.000000 | 2.000000 | 2.000000 | 2.000000 | 2.000000 | 2.000000 | 1.000000 | 2.000000 |
| 75% | 69.000000 | 2.000000 | 2.000000 | 2.000000 | 2.000000 | 2.000000 | 2.000000 | 2.000000 | 2.000000 | 2.000000 | 2.000000 | 2.000000 | 2.000000 | 2.000000 |
| max | 87.000000 | 2.000000 | 2.000000 | 2.000000 | 2.000000 | 2.000000 | 2.000000 | 2.000000 | 2.000000 | 2.000000 | 2.000000 | 2.000000 | 2.000000 | 2.000000 |

**In this dataset, GENDER & LUNG\_CANCER attributes are in object data type. So, let's convert them to numerical values using LabelEncoder from sklearn. LabelEncoder is a utility class to help normalize labels such that they contain only values between 0 and n\_classes-1. It can also be used to transform non-numerical labels (as long as they are hashable and comparable) to numerical labels. Also let's make every other attributes as YES=1 & NO=0.**

from sklearn import preprocessing

le=preprocessing.LabelEncoder()

df['GENDER']=le.fit\_transform(df['GENDER'])

df['LUNG\_CANCER']=le.fit\_transform(df['LUNG\_CANCER'])

df['SMOKING']=le.fit\_transform(df['SMOKING'])

df['YELLOW\_FINGERS']=le.fit\_transform(df['YELLOW\_FINGERS'])

df['ANXIETY']=le.fit\_transform(df['ANXIETY'])

df['PEER\_PRESSURE']=le.fit\_transform(df['PEER\_PRESSURE'])

df['CHRONIC DISEASE']=le.fit\_transform(df['CHRONIC DISEASE'])

df['FATIGUE ']=le.fit\_transform(df['FATIGUE '])

df['ALLERGY ']=le.fit\_transform(df['ALLERGY '])

df['WHEEZING']=le.fit\_transform(df['WHEEZING'])

df['ALCOHOL CONSUMING']=le.fit\_transform(df['ALCOHOL CONSUMING'])

df['COUGHING']=le.fit\_transform(df['COUGHING'])

df['SHORTNESS OF BREATH']=le.fit\_transform(df['SHORTNESS OF BREATH'])

df['SWALLOWING DIFFICULTY']=le.fit\_transform(df['SWALLOWING DIFFICULTY'])

df['CHEST PAIN']=le.fit\_transform(df['CHEST PAIN'])

df['LUNG\_CANCER']=le.fit\_transform(df['LUNG\_CANCER'])

*#Let's check what's happened now*

df

|  | GENDER | AGE | SMOKING | YELLOW\_FINGERS | ANXIETY | PEER\_PRESSURE | CHRONIC DISEASE | FATIGUE | ALLERGY | WHEEZING | ALCOHOL CONSUMING | COUGHING | SHORTNESS OF BREATH | SWALLOWING DIFFICULTY | CHEST PAIN | LUNG\_CANCER |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 69 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 74 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 2 | 0 | 59 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 |
| 3 | 1 | 63 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 |
| 4 | 0 | 63 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 279 | 0 | 59 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 280 | 0 | 59 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 281 | 1 | 55 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| 282 | 1 | 46 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| 283 | 1 | 60 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

276 rows × 16 columns

**Note: Male=1 & Female=0. Also for other variables, YES=1 & NO=0**

df.info()

<class 'pandas.core.frame.DataFrame'>

Int64Index: 276 entries, 0 to 283

Data columns (total 16 columns):

# Column Non-Null Count Dtype

--- ------ -------------- -----

0 GENDER 276 non-null int64

1 AGE 276 non-null int64

2 SMOKING 276 non-null int64

3 YELLOW\_FINGERS 276 non-null int64

4 ANXIETY 276 non-null int64

5 PEER\_PRESSURE 276 non-null int64

6 CHRONIC DISEASE 276 non-null int64

7 FATIGUE 276 non-null int64

8 ALLERGY 276 non-null int64

9 WHEEZING 276 non-null int64

10 ALCOHOL CONSUMING 276 non-null int64

11 COUGHING 276 non-null int64

12 SHORTNESS OF BREATH 276 non-null int64

13 SWALLOWING DIFFICULTY 276 non-null int64

14 CHEST PAIN 276 non-null int64

15 LUNG\_CANCER 276 non-null int64

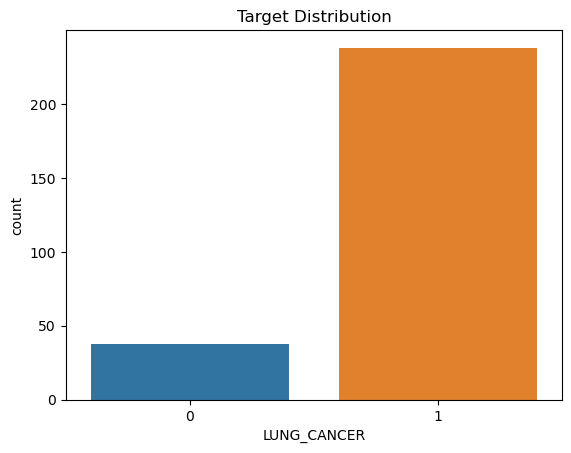
dtypes: int64(16)

memory usage: 36.7 KB

*#Let's check the distributaion of Target variable.*

sns.countplot(x='LUNG\_CANCER', data=df,)

plt.title('Target Distribution');



***That is, Target Distribution is imbalanced.***

df['LUNG\_CANCER'].value\_counts()

1 238

0 38

Name: LUNG\_CANCER, dtype: int64

***We will handle this imbalance before applyig algorithm.***

**Now let's do some Data Visualizations for the better understanding of how the independent features are related to the target variable..**

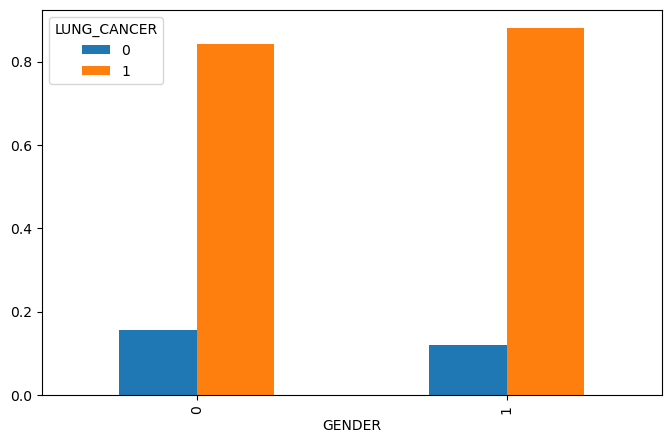
*# function for plotting*

def plot(col, df=df):

return df.groupby(col)['LUNG\_CANCER'].value\_counts(normalize=True).unstack().plot(kind='bar', figsize=(8,5))

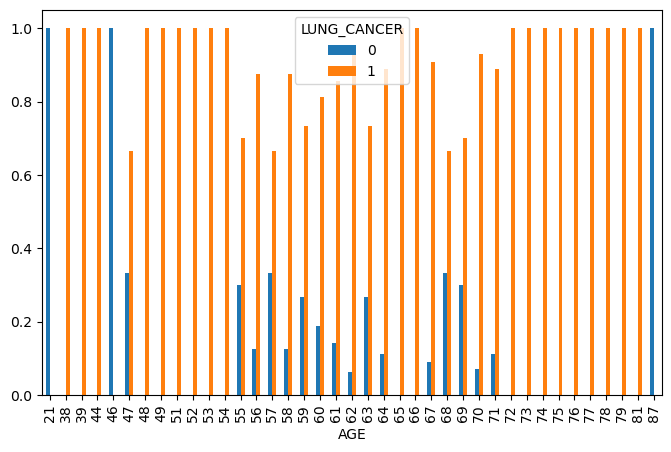
plot('GENDER')

<AxesSubplot:xlabel='GENDER'>



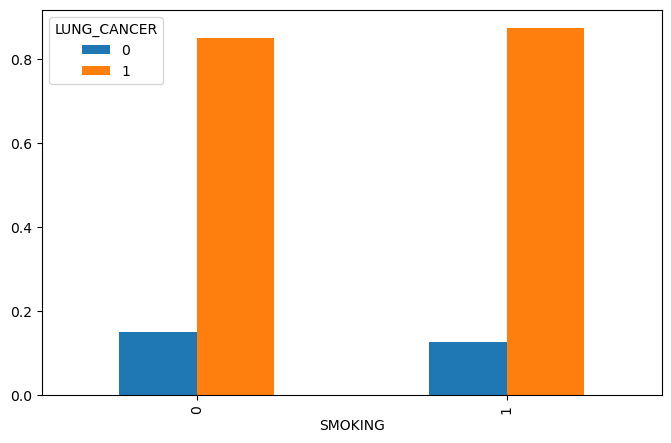
plot('AGE')

<AxesSubplot:xlabel='AGE'>



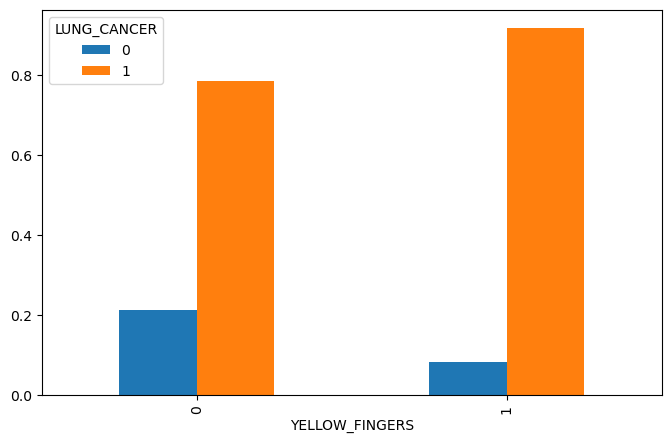
plot('SMOKING')

<AxesSubplot:xlabel='SMOKING'>



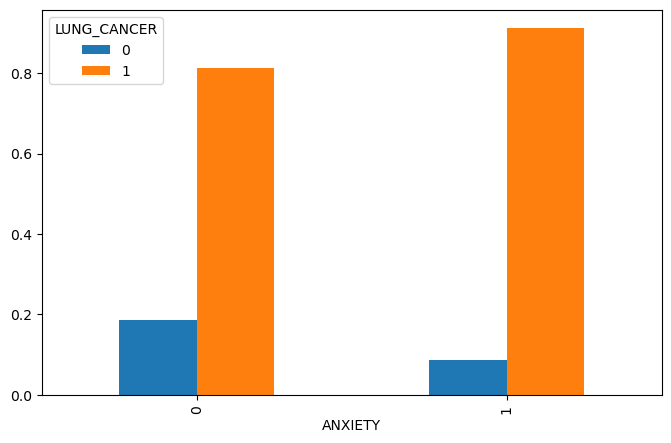
plot('YELLOW\_FINGERS')

<AxesSubplot:xlabel='YELLOW\_FINGERS'>



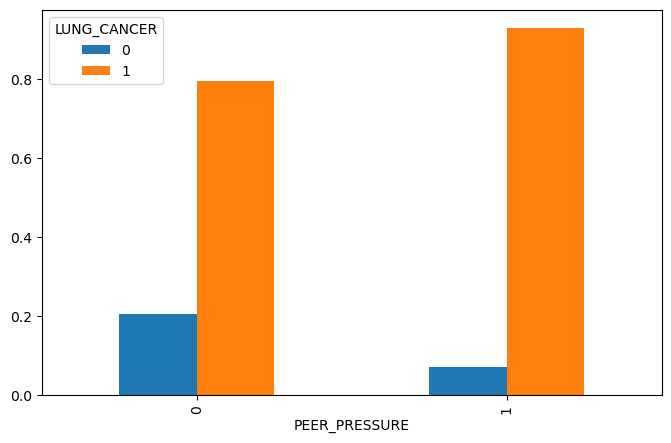
plot('ANXIETY')

<AxesSubplot:xlabel='ANXIETY'>



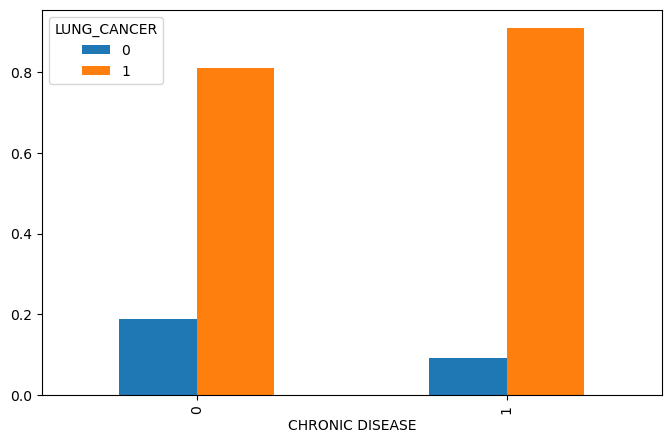
plot('PEER\_PRESSURE')

<AxesSubplot:xlabel='PEER\_PRESSURE'>



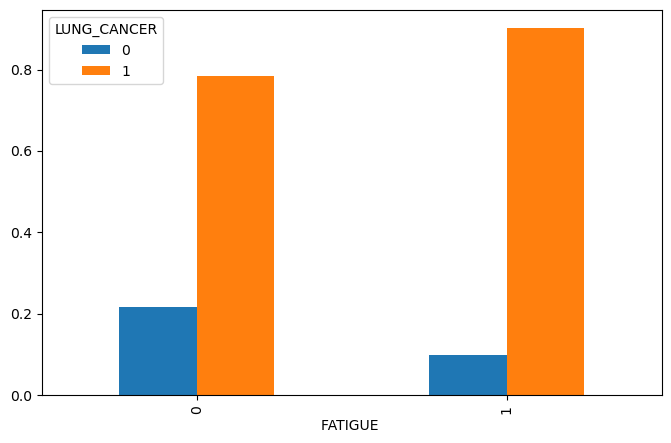
plot('CHRONIC DISEASE')

<AxesSubplot:xlabel='CHRONIC DISEASE'>



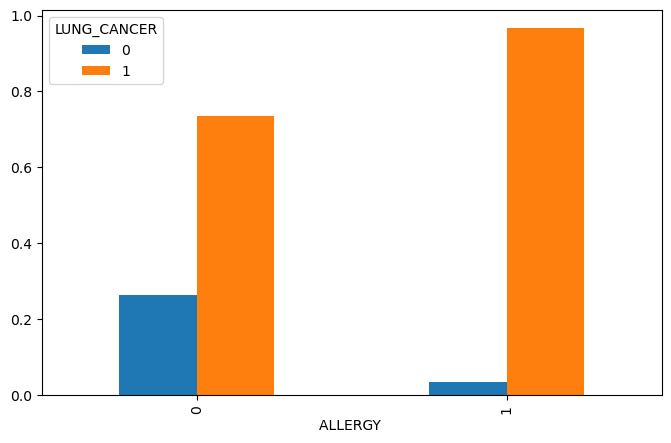
plot('FATIGUE ')

<AxesSubplot:xlabel='FATIGUE '>



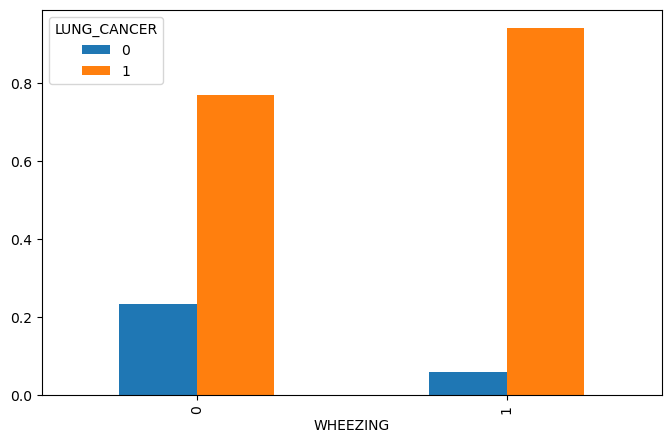
plot('ALLERGY ')

<AxesSubplot:xlabel='ALLERGY '>



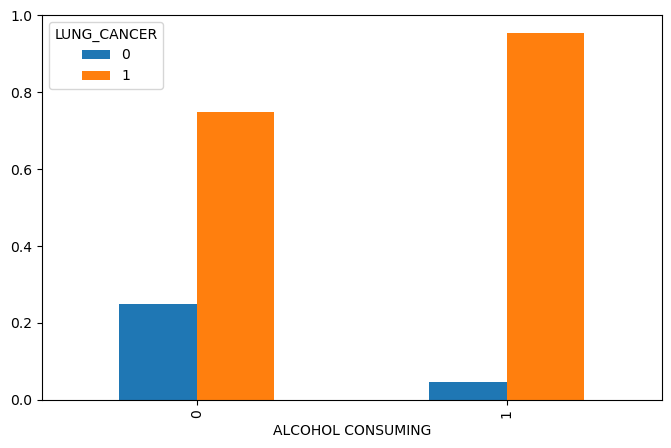
plot('WHEEZING')

<AxesSubplot:xlabel='WHEEZING'>



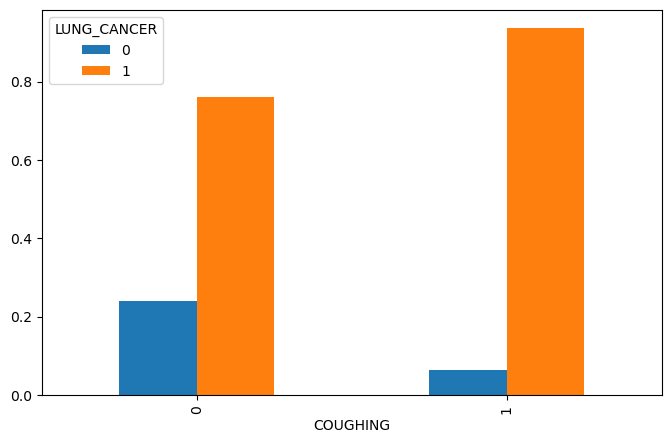
plot('ALCOHOL CONSUMING')

<AxesSubplot:xlabel='ALCOHOL CONSUMING'>



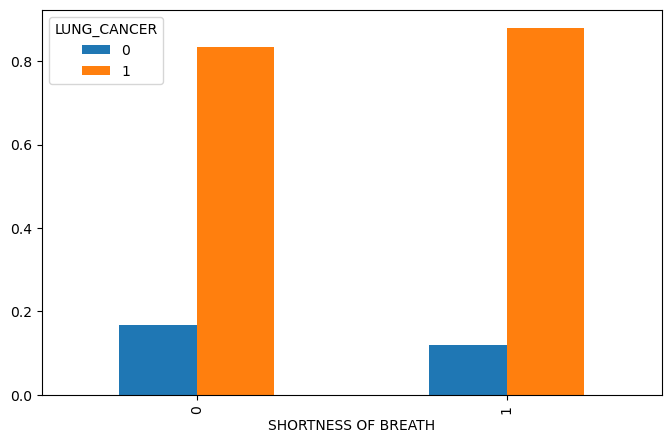
plot('COUGHING')

<AxesSubplot:xlabel='COUGHING'>



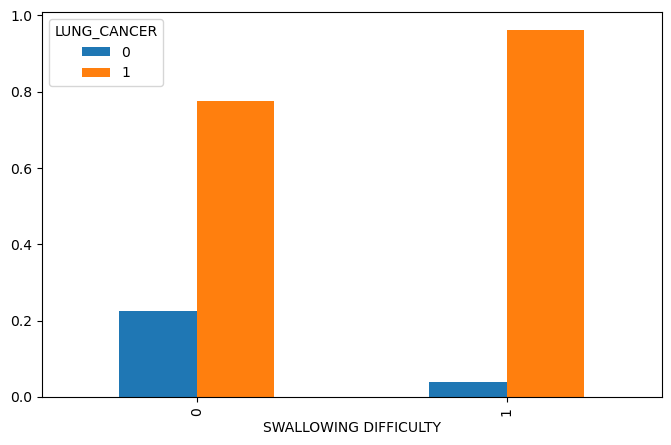
plot('SHORTNESS OF BREATH')

<AxesSubplot:xlabel='SHORTNESS OF BREATH'>



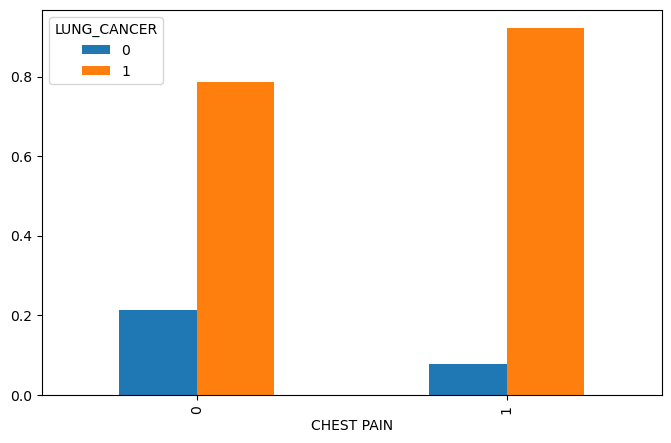
plot('SWALLOWING DIFFICULTY')

<AxesSubplot:xlabel='SWALLOWING DIFFICULTY'>



plot('CHEST PAIN')

<AxesSubplot:xlabel='CHEST PAIN'>



**From the visualizations, it is clear that in the given dataset, the features GENDER, AGE, SMOKING and SHORTNESS OF BREATH don't have that much relationship with LUNG CANCER. So let's drop those features to make this dataset more clean.**

df\_new=df.drop(columns=['GENDER','AGE', 'SMOKING', 'SHORTNESS OF BREATH'])

df\_new

|  | YELLOW\_FINGERS | ANXIETY | PEER\_PRESSURE | CHRONIC DISEASE | FATIGUE | ALLERGY | WHEEZING | ALCOHOL CONSUMING | COUGHING | SWALLOWING DIFFICULTY | CHEST PAIN | LUNG\_CANCER |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| 2 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 3 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 |
| 4 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 279 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
| 280 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 281 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| 282 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| 283 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |

276 rows × 12 columns

**CORRELATION**

*#Finding Correlation*

cn=df\_new.corr()

cn

|  | YELLOW\_FINGERS | ANXIETY | PEER\_PRESSURE | CHRONIC DISEASE | FATIGUE | ALLERGY | WHEEZING | ALCOHOL CONSUMING | COUGHING | SWALLOWING DIFFICULTY | CHEST PAIN | LUNG\_CANCER |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| YELLOW\_FINGERS | 1.000000 | 0.558344 | 0.313067 | 0.015316 | -0.099644 | -0.147130 | -0.058756 | -0.273643 | 0.020803 | 0.333349 | -0.099169 | 0.189192 |
| ANXIETY | 0.558344 | 1.000000 | 0.210278 | -0.006938 | -0.181474 | -0.159451 | -0.174009 | -0.152228 | -0.218843 | 0.478820 | -0.123182 | 0.144322 |
| PEER\_PRESSURE | 0.313067 | 0.210278 | 1.000000 | 0.042893 | 0.094661 | -0.066887 | -0.037769 | -0.132603 | -0.068224 | 0.327764 | -0.074655 | 0.195086 |
| CHRONIC DISEASE | 0.015316 | -0.006938 | 0.042893 | 1.000000 | -0.099411 | 0.134309 | -0.040546 | 0.010144 | -0.160813 | 0.068263 | -0.048895 | 0.143692 |
| FATIGUE | -0.099644 | -0.181474 | 0.094661 | -0.099411 | 1.000000 | -0.001841 | 0.152151 | -0.181573 | 0.148538 | -0.115727 | 0.013757 | 0.160078 |
| ALLERGY | -0.147130 | -0.159451 | -0.066887 | 0.134309 | -0.001841 | 1.000000 | 0.166517 | 0.378125 | 0.206367 | -0.037581 | 0.245440 | 0.333552 |
| WHEEZING | -0.058756 | -0.174009 | -0.037769 | -0.040546 | 0.152151 | 0.166517 | 1.000000 | 0.261061 | 0.353657 | 0.108304 | 0.142846 | 0.249054 |
| ALCOHOL CONSUMING | -0.273643 | -0.152228 | -0.132603 | 0.010144 | -0.181573 | 0.378125 | 0.261061 | 1.000000 | 0.198023 | -0.000635 | 0.310767 | 0.294422 |
| COUGHING | 0.020803 | -0.218843 | -0.068224 | -0.160813 | 0.148538 | 0.206367 | 0.353657 | 0.198023 | 1.000000 | -0.136885 | 0.077988 | 0.253027 |
| SWALLOWING DIFFICULTY | 0.333349 | 0.478820 | 0.327764 | 0.068263 | -0.115727 | -0.037581 | 0.108304 | -0.000635 | -0.136885 | 1.000000 | 0.102674 | 0.268940 |
| CHEST PAIN | -0.099169 | -0.123182 | -0.074655 | -0.048895 | 0.013757 | 0.245440 | 0.142846 | 0.310767 | 0.077988 | 0.102674 | 1.000000 | 0.194856 |
| LUNG\_CANCER | 0.189192 | 0.144322 | 0.195086 | 0.143692 | 0.160078 | 0.333552 | 0.249054 | 0.294422 | 0.253027 | 0.268940 | 0.194856 | 1.000000 |

*#Correlation*

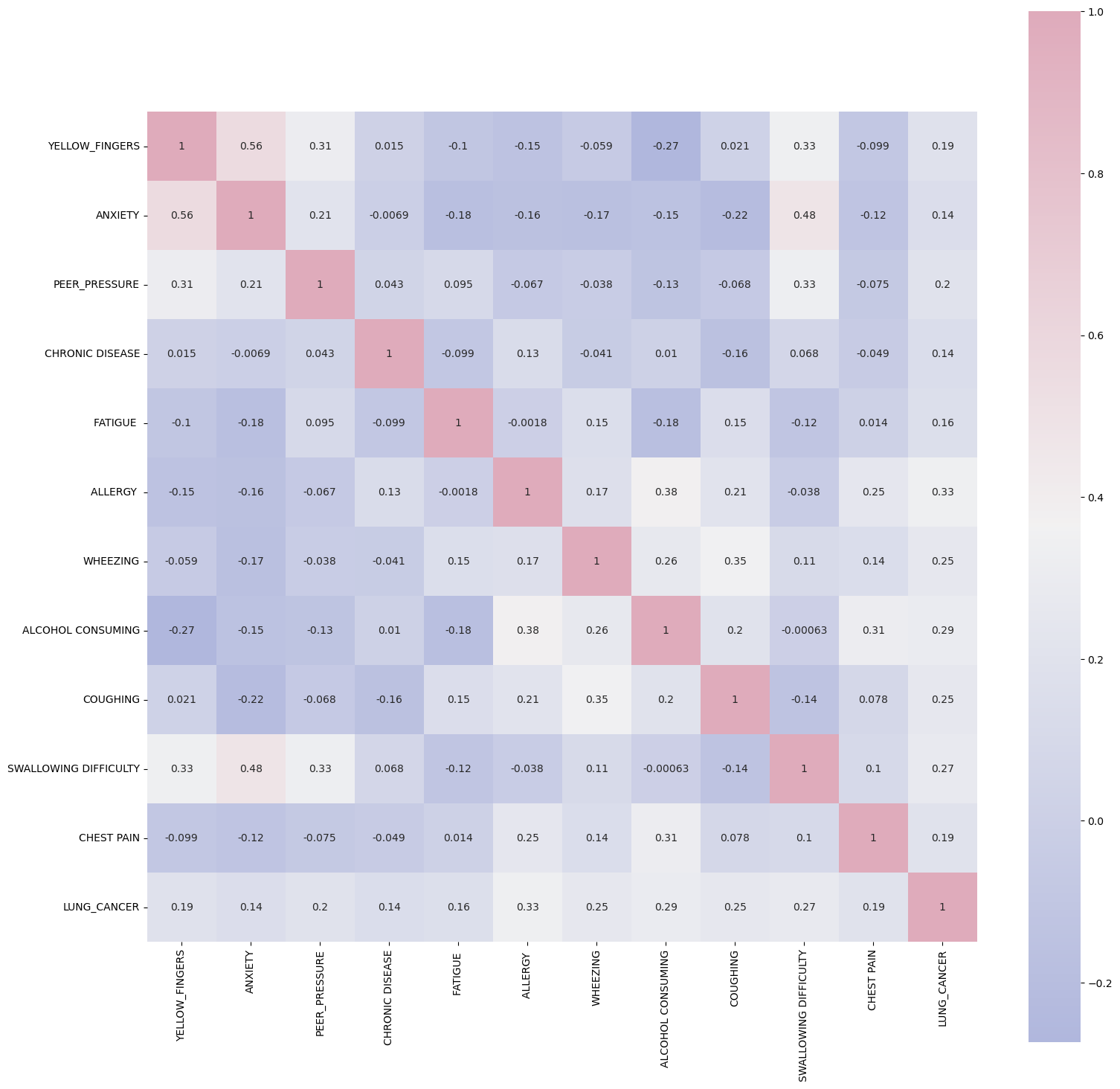
cmap=sns.diverging\_palette(260,-10,s=50, l=75, n=6,

as\_cmap=True)

plt.subplots(figsize=(18,18))

sns.heatmap(cn,cmap=cmap,annot=True, square=True)

plt.show()

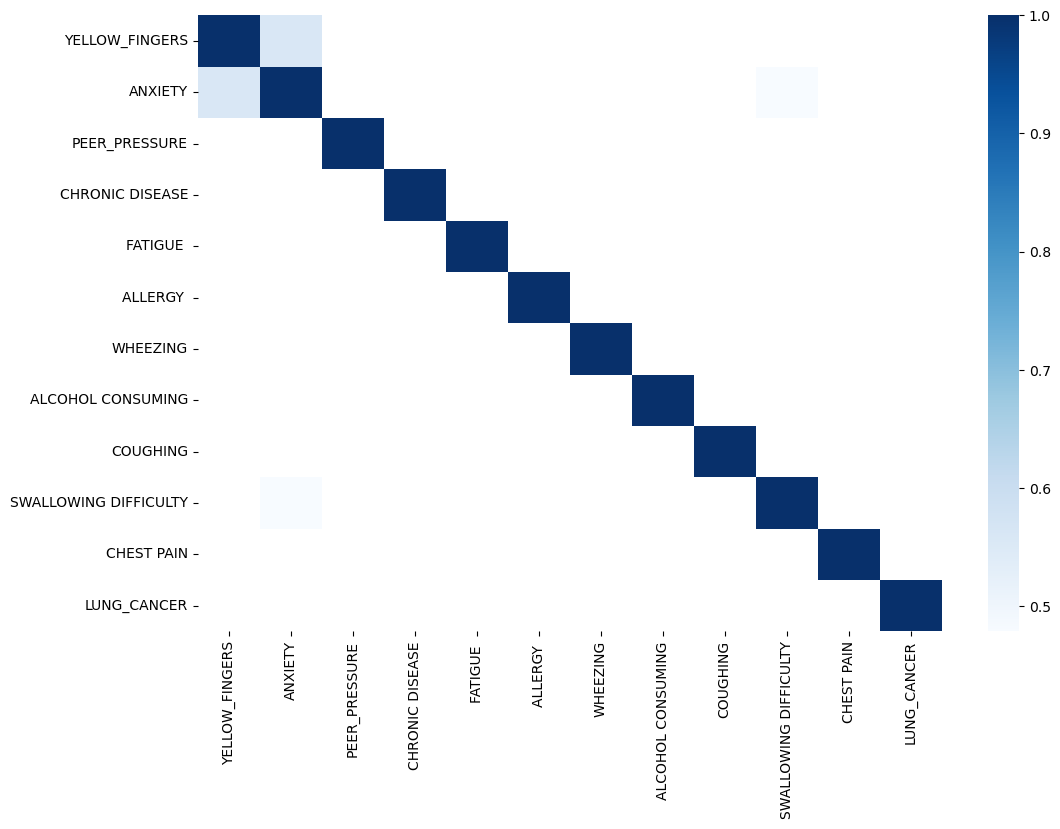


kot = cn[cn>=.40]

plt.figure(figsize=(12,8))

sns.heatmap(kot, cmap="Blues")

<AxesSubplot:>



***Feature Engineering***

Feature Engineering is the process of creating new features using existing features.

***The correlation matrix shows that ANXIETY and YELLOW\_FINGERS are correlated more than 50%. So, lets create a new feature combining them.***

df\_new['ANXYELFIN']=df\_new['ANXIETY']\*df\_new['YELLOW\_FINGERS']

df\_new

|  | YELLOW\_FINGERS | ANXIETY | PEER\_PRESSURE | CHRONIC DISEASE | FATIGUE | ALLERGY | WHEEZING | ALCOHOL CONSUMING | COUGHING | SWALLOWING DIFFICULTY | CHEST PAIN | LUNG\_CANCER | ANXYELFIN |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 |
| 2 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 3 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 |
| 4 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 279 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 280 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 281 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 282 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 |
| 283 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

276 rows × 13 columns

*#Splitting independent and dependent variables*

X = df\_new.drop('LUNG\_CANCER', axis = 1)

y = df\_new['LUNG\_CANCER']

***Target Distribution Imbalance Handling***

from imblearn.over\_sampling import ADASYN

adasyn = ADASYN(random\_state=42)

X, y = adasyn.fit\_resample(X, y)

len(X)

477

**Logistic Regression**

*#Splitting data for training and testing*

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test= train\_test\_split(X, y, test\_size= 0.25, random\_state=0)

*#Fitting training data to the model*

from sklearn.linear\_model import LogisticRegression

lr\_model=LogisticRegression(random\_state=0)

lr\_model.fit(X\_train, y\_train)

LogisticRegression(random\_state=0)

*#Predicting result using testing data*

y\_lr\_pred= lr\_model.predict(X\_test)

y\_lr\_pred

array([1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0,

1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0,

1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1,

0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1,

0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0,

1, 0, 0, 0, 0, 1, 0, 1, 1, 0])

*#Model accuracy*

from sklearn.metrics import classification\_report, accuracy\_score, f1\_score

lr\_cr=classification\_report(y\_test, y\_lr\_pred)

print(lr\_cr)

precision recall f1-score support

0 0.96 1.00 0.98 64

1 1.00 0.95 0.97 56

accuracy 0.97 120

macro avg 0.98 0.97 0.97 120

weighted avg 0.98 0.97 0.97 120

**This model is almost 97% accurate.**

**Decision Tree**

*#Fitting training data to the model*

from sklearn.tree import DecisionTreeClassifier

dt\_model= DecisionTreeClassifier(criterion='entropy', random\_state=0)

dt\_model.fit(X\_train, y\_train)

DecisionTreeClassifier(criterion='entropy', random\_state=0)

*#Predicting result using testing data*

y\_dt\_pred= dt\_model.predict(X\_test)

y\_dt\_pred

array([1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0,

1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0,

1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1,

1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1,

1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0,

1, 0, 0, 0, 0, 1, 0, 1, 1, 0])

*#Model accuracy*

dt\_cr=classification\_report(y\_test, y\_dt\_pred)

print(dt\_cr)

precision recall f1-score support

0 0.93 0.97 0.95 64

1 0.96 0.91 0.94 56

accuracy 0.94 120

macro avg 0.94 0.94 0.94 120

weighted avg 0.94 0.94 0.94 120

**This model is 94% accurate.**

**K Nearest Neighbor**

*#Fitting K-NN classifier to the training set*

from sklearn.neighbors import KNeighborsClassifier

knn\_model= KNeighborsClassifier(n\_neighbors=5, metric='minkowski', p=2 )

knn\_model.fit(X\_train, y\_train)

KNeighborsClassifier()

*#Predicting result using testing data*

y\_knn\_pred= knn\_model.predict(X\_test)

y\_knn\_pred

array([1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0,

1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0,

1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1,

1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1,

0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0,

1, 0, 0, 0, 0, 1, 0, 1, 1, 0])

*#Model accuracy*

knn\_cr=classification\_report(y\_test, y\_knn\_pred)

print(knn\_cr)

precision recall f1-score support

0 0.93 1.00 0.96 64

1 1.00 0.91 0.95 56

accuracy 0.96 120

macro avg 0.96 0.96 0.96 120

weighted avg 0.96 0.96 0.96 120

**This model is 96% accurate.**

**Gaussian Naive Bayes**

*#Fitting Gaussian Naive Bayes classifier to the training set*

from sklearn.naive\_bayes import GaussianNB

gnb\_model = GaussianNB()

gnb\_model.fit(X\_train, y\_train)

GaussianNB()

*#Predicting result using testing data*

y\_gnb\_pred= gnb\_model.predict(X\_test)

y\_gnb\_pred

array([1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0,

1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0,

1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1,

1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1,

0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0,

1, 1, 1, 0, 0, 1, 0, 1, 1, 0])

*#Model accuracy*

gnb\_cr=classification\_report(y\_test, y\_gnb\_pred)

print(gnb\_cr)

precision recall f1-score support

0 0.95 0.89 0.92 64

1 0.88 0.95 0.91 56

accuracy 0.92 120

macro avg 0.92 0.92 0.92 120

weighted avg 0.92 0.92 0.92 120

**This model is 92% accurate.**

**Multinomial Naive Bayes**

*#Fitting Multinomial Naive Bayes classifier to the training set*

from sklearn.naive\_bayes import MultinomialNB

mnb\_model = MultinomialNB()

mnb\_model.fit(X\_train, y\_train)

MultinomialNB()

*#Predicting result using testing data*

y\_mnb\_pred= mnb\_model.predict(X\_test)

y\_mnb\_pred

array([1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0,

1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0,

1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1,

1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1,

0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0,

1, 1, 1, 1, 1, 1, 0, 0, 1, 0])

*#Model accuracy*

mnb\_cr=classification\_report(y\_test, y\_mnb\_pred)

print(mnb\_cr)

precision recall f1-score support

0 0.89 0.73 0.80 64

1 0.75 0.89 0.81 56

accuracy 0.81 120

macro avg 0.82 0.81 0.81 120

weighted avg 0.82 0.81 0.81 120

**This model is 81% accurate.**

**Support Vector Classifier**

*#Fitting SVC to the training set*

from sklearn.svm import SVC

svc\_model = SVC()

svc\_model.fit(X\_train, y\_train)

SVC()

*#Predicting result using testing data*

y\_svc\_pred= svc\_model.predict(X\_test)

y\_svc\_pred

array([1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0,

1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0,

1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1,

1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1,

0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0,

1, 0, 0, 0, 0, 1, 0, 1, 1, 0])

*#Model accuracy*

svc\_cr=classification\_report(y\_test, y\_svc\_pred)

print(svc\_cr)

precision recall f1-score support

0 0.98 0.98 0.98 64

1 0.98 0.98 0.98 56

accuracy 0.98 120

macro avg 0.98 0.98 0.98 120

weighted avg 0.98 0.98 0.98 120

**This model is 98% accurate.**

**Random Forest**

*#Training*

from sklearn.ensemble import RandomForestClassifier

rf\_model = RandomForestClassifier()

rf\_model.fit(X\_train, y\_train)

RandomForestClassifier()

*#Predicting result using testing data*

y\_rf\_pred= rf\_model.predict(X\_test)

y\_rf\_pred

array([1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0,

1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0,

1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1,

1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1,

0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0,

1, 0, 0, 0, 0, 1, 0, 1, 1, 0])

*#Model accuracy*

rf\_cr=classification\_report(y\_test, y\_rf\_pred)

print(rf\_cr)

precision recall f1-score support

0 0.98 0.98 0.98 64

1 0.98 0.98 0.98 56

accuracy 0.98 120

macro avg 0.98 0.98 0.98 120

weighted avg 0.98 0.98 0.98 120

**This model is also 98% accurate.**

**XGBoost**

from xgboost import XGBClassifier

xgb\_model = XGBClassifier()

xgb\_model.fit(X\_train, y\_train)

XGBClassifier(base\_score=0.5, booster='gbtree', callbacks=None,

colsample\_bylevel=1, colsample\_bynode=1, colsample\_bytree=1,

early\_stopping\_rounds=None, enable\_categorical=False,

eval\_metric=None, gamma=0, gpu\_id=-1, grow\_policy='depthwise',

importance\_type=None, interaction\_constraints='',

learning\_rate=0.300000012, max\_bin=256, max\_cat\_to\_onehot=4,

max\_delta\_step=0, max\_depth=6, max\_leaves=0, min\_child\_weight=1,

missing=nan, monotone\_constraints='()', n\_estimators=100,

n\_jobs=0, num\_parallel\_tree=1, predictor='auto', random\_state=0,

reg\_alpha=0, reg\_lambda=1, ...)

*#Predicting result using testing data*

y\_xgb\_pred= xgb\_model.predict(X\_test)

y\_xgb\_pred

array([1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0,

1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0,

1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1,

1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1,

1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0,

1, 0, 0, 0, 0, 1, 0, 1, 1, 0])

*#Model accuracy*

xgb\_cr=classification\_report(y\_test, y\_xgb\_pred)

print(xgb\_cr)

precision recall f1-score support

0 0.98 0.97 0.98 64

1 0.96 0.98 0.97 56

accuracy 0.97 120

macro avg 0.97 0.98 0.97 120

weighted avg 0.98 0.97 0.98 120

**This model is also 97% accurate.**

**Multi-layer Perceptron classifier**

*#Training a neural network model*

from sklearn.neural\_network import MLPClassifier

mlp\_model = MLPClassifier()

mlp\_model.fit(X\_train, y\_train)

MLPClassifier()

*#Predicting result using testing data*

y\_mlp\_pred= mlp\_model.predict(X\_test)

y\_mlp\_pred

array([1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0,

1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0,

1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1,

1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1,

0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0,

1, 0, 0, 0, 0, 1, 0, 1, 1, 0])

*#Model accuracy*

mlp\_cr=classification\_report(y\_test, y\_mlp\_pred)

print(mlp\_cr)

precision recall f1-score support

0 0.98 0.98 0.98 64

1 0.98 0.98 0.98 56

accuracy 0.98 120

macro avg 0.98 0.98 0.98 120

weighted avg 0.98 0.98 0.98 120

**This model is also 98% accurate.**

**Gradient Boosting**

*#Training*

from sklearn.ensemble import GradientBoostingClassifier

gb\_model = GradientBoostingClassifier()

gb\_model.fit(X\_train, y\_train)

GradientBoostingClassifier()

*#Predicting result using testing data*

y\_gb\_pred= gb\_model.predict(X\_test)

y\_gb\_pred

array([1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0,

1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0,

1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1,

1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1,

0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0,

1, 0, 0, 0, 0, 1, 0, 1, 1, 0])

*#Model accuracy*

gb\_cr=classification\_report(y\_test, y\_gb\_pred)

print(gb\_cr)

precision recall f1-score support

0 0.98 0.98 0.98 64

1 0.98 0.98 0.98 56

accuracy 0.98 120

macro avg 0.98 0.98 0.98 120

weighted avg 0.98 0.98 0.98 120

**This model is also 98% accurate.**

From the above calculated accuracies, it is clear that the SVC, Random Forest, Multi-layer Perceptron and Gradient Boost models performed atmost level while the worst performed one is Multinomial Naive Bayes. However, I'm interested in a more efficient way of evaluating these models. Let's go for the Cross Validation methods using both K-Fold and Stratified K-Fold

**Cross Validation**

K-Fold cross validation is a popular technique used in machine learning for model evaluation and selection. It involves dividing a dataset into K subsets of equal size, called folds. The algorithm then trains and evaluates the model K times, each time using a different fold as the validation set and the remaining K-1 folds as the training set.

During each iteration of K-Fold cross validation, the model is trained on K-1 folds and evaluated on the remaining fold. The performance metrics are then averaged over all K iterations to obtain an estimate of the model's overall performance.

K-Fold cross validation is a robust method for model evaluation because it uses all the available data for training and testing. It also helps to reduce the risk of overfitting and provides a more accurate estimate of the model's performance than using a single training-test split.

Typically, values of K between 5 and 10 are used for K-Fold cross validation, but the optimal value of K may vary depending on the size and complexity of the dataset, as well as the type of model being evaluated.

Here are some general guidelines that can help you choose an appropriate value of K:

1. For small datasets, it is recommended to use a larger value of K, such as 10. This is because the larger value of K allows for more robust estimates of model performance, given the limited amount of data.
2. For larger datasets, a smaller value of K can be used, such as 5. This is because a larger value of K will result in K smaller training sets, which may not be representative of the full dataset. Using a smaller value of K ensures that each fold has a sufficient amount of data for both training and testing.
3. For models that are computationally expensive or time-consuming to train, a smaller value of K is preferred to reduce the overall training time.
4. It's also essential to note that the choice of K should not be based solely on the accuracy of the model. Other metrics, such as precision, recall, and F1 score, should also be considered, as they can provide valuable insights into the performance of the model.

*# K-Fold Cross Validation*

from sklearn.model\_selection import KFold

from sklearn.model\_selection import cross\_val\_score

k = 10

kf = KFold(n\_splits=k, shuffle=True, random\_state=42)

*# Logistic regerssion model*

lr\_model\_scores = cross\_val\_score(lr\_model,X, y, cv=kf)

*# Decision tree model*

dt\_model\_scores = cross\_val\_score(dt\_model,X, y, cv=kf)

*# KNN model*

knn\_model\_scores = cross\_val\_score(knn\_model,X, y, cv=kf)

*# Gaussian naive bayes model*

gnb\_model\_scores = cross\_val\_score(gnb\_model,X, y, cv=kf)

*# Multinomial naive bayes model*

mnb\_model\_scores = cross\_val\_score(mnb\_model,X, y, cv=kf)

*# Support Vector Classifier model*

svc\_model\_scores = cross\_val\_score(svc\_model,X, y, cv=kf)

*# Random forest model*

rf\_model\_scores = cross\_val\_score(rf\_model,X, y, cv=kf)

*# XGBoost model*

xgb\_model\_scores = cross\_val\_score(xgb\_model,X, y, cv=kf)

*# Multi-layer perceptron model*

mlp\_model\_scores = cross\_val\_score(mlp\_model,X, y, cv=kf)

*# Gradient boost model*

gb\_model\_scores = cross\_val\_score(gb\_model,X, y, cv=kf)

print("Logistic regression models' average accuracy:", np.mean(lr\_model\_scores))

print("Decision tree models' average accuracy:", np.mean(dt\_model\_scores))

print("KNN models' average accuracy:", np.mean(knn\_model\_scores))

print("Gaussian naive bayes models' average accuracy:", np.mean(gnb\_model\_scores))

print("Multinomial naive bayes models' average accuracy:", np.mean(mnb\_model\_scores))

print("Support Vector Classifier models' average accuracy:", np.mean(svc\_model\_scores))

print("Random forest models' average accuracy:", np.mean(rf\_model\_scores))

print("XGBoost models' average accuracy:", np.mean(xgb\_model\_scores))

print("Multi-layer perceptron models' average accuracy:", np.mean(mlp\_model\_scores))

print("Gradient boost models' average accuracy:", np.mean(gb\_model\_scores))

Logistic regression models' average accuracy: 0.930895390070922

Decision tree models' average accuracy: 0.9539893617021278

KNN models' average accuracy: 0.9330673758865249

Gaussian naive bayes models' average accuracy: 0.8846187943262411

Multinomial naive bayes models' average accuracy: 0.7421542553191489

Support Vector Classifier models' average accuracy: 0.9497340425531917

Random forest models' average accuracy: 0.949822695035461

XGBoost models' average accuracy: 0.9497783687943263

Multi-layer perceptron models' average accuracy: 0.9372340425531915

Gradient boost models' average accuracy: 0.9560726950354612

**So the K-Fold cross validation is showing Gradient Boost model gives the most accuracy of 95.6%, and Decision Tree model also gives almost same accuracy, while Multinomial Naive Bayes model gives the least accuarcy of 74.2%.**

Stratified K-Fold cross-validation is a modification of the standard K-Fold cross-validation technique that is commonly used in machine learning when working with imbalanced datasets. The goal of Stratified K-Fold cross-validation is to ensure that each fold is representative of the overall dataset in terms of the class distribution.

In standard K-Fold cross-validation, the data is split into K folds, and each fold is used as the validation set in turn. However, if the dataset has an imbalanced class distribution, this can lead to some of the folds having significantly fewer samples from the minority class, which can result in biased performance estimates.

To address this issue, Stratified K-Fold cross-validation ensures that each fold has a similar proportion of samples from each class. It works by first dividing the dataset into K folds, as in standard K-Fold cross-validation. Then, for each fold, the algorithm ensures that the proportion of samples from each class is roughly the same as the proportion in the full dataset. This ensures that the model is evaluated on a representative sample of the data, regardless of the class distribution.

Stratified K-Fold cross-validation is a powerful tool for evaluating the performance of machine learning models on imbalanced datasets. It can help to ensure that the model's performance is accurately estimated and that the model is robust to class imbalances in the dataset.

*# K-Fold Cross Validation*

from sklearn.model\_selection import StratifiedKFold

from sklearn.model\_selection import cross\_val\_score

k = 10

kf = StratifiedKFold(n\_splits=k)

*# Logistic regerssion model*

lr\_model\_scores = cross\_val\_score(lr\_model,X, y, cv=kf)

*# Decision tree model*

dt\_model\_scores = cross\_val\_score(dt\_model,X, y, cv=kf)

*# KNN model*

knn\_model\_scores = cross\_val\_score(knn\_model,X, y, cv=kf)

*# Gaussian naive bayes model*

gnb\_model\_scores = cross\_val\_score(gnb\_model,X, y, cv=kf)

*# Multinomial naive bayes model*

mnb\_model\_scores = cross\_val\_score(mnb\_model,X, y, cv=kf)

*# Support Vector Classifier model*

svc\_model\_scores = cross\_val\_score(svc\_model,X, y, cv=kf)

*# Random forest model*

rf\_model\_scores = cross\_val\_score(rf\_model,X, y, cv=kf)

*# XGBoost model*

xgb\_model\_scores = cross\_val\_score(xgb\_model,X, y, cv=kf)

*# Multi-layer perceptron model*

mlp\_model\_scores = cross\_val\_score(mlp\_model,X, y, cv=kf)

*# Gradient boost model*

gb\_model\_scores = cross\_val\_score(gb\_model,X, y, cv=kf)

print("Logistic regression models' average accuracy:", np.mean(lr\_model\_scores))

print("Decision tree models' average accuracy:", np.mean(dt\_model\_scores))

print("KNN models' average accuracy:", np.mean(knn\_model\_scores))

print("Gaussian naive bayes models' average accuracy:", np.mean(gnb\_model\_scores))

print("Multinomial naive bayes models' average accuracy:", np.mean(mnb\_model\_scores))

print("Support Vector Classifier models' average accuracy:", np.mean(svc\_model\_scores))

print("Random forest models' average accuracy:", np.mean(rf\_model\_scores))

print("XGBoost models' average accuracy:", np.mean(xgb\_model\_scores))

print("Multi-layer perceptron models' average accuracy:", np.mean(mlp\_model\_scores))

print("Gradient boost models' average accuracy:", np.mean(gb\_model\_scores))

Logistic regression models' average accuracy: 0.9288120567375886

Decision tree models' average accuracy: 0.9227393617021278

KNN models' average accuracy: 0.9184397163120567

Gaussian naive bayes models' average accuracy: 0.8870124113475178

Multinomial naive bayes models' average accuracy: 0.7572251773049644

Support Vector Classifier models' average accuracy: 0.9476063829787235

Random forest models' average accuracy: 0.9456560283687944

XGBoost models' average accuracy: 0.9457446808510639

Multi-layer perceptron models' average accuracy: 0.93927304964539

Gradient boost models' average accuracy: 0.947695035460993

**So the Stratified K-Fold cross validation is showing Random Forest model gives the most accuracy of 94.6%, and also other models like Gradient Boost, Support Vector Classifier, XGBoost gives almost same accuracies, while Multinomial Naive Bayes model gives the least accuarcy of 75.7%.**