healthcare-diabeties

September 2, 2024

##Problem statement Develop a predictive model with high accuracy to determine the accuracy of diabetes in patients.

##Dataset Description The dataset includes various medical predictor variables and one target variable that is outcome. The predictor variables encompass essential health metrics, such as the number of pregnancies , glucose concentration, diabetic blood pressure, skin thickness , insulin levels, Body Mass Index (BMI), diabetic pedigree function, and age.

###Predictor Variables ####Pregnancies:

- Number of times pregnant

####Glucose:

- Plasma glucose concentration at 2 hours in an oral glucose tolerance test.

####BloodPressure:

- Diastolic blood pressure (mm Hg) ####SkinThickness:
- Triceps skinfold thickness (mm) ####Insulin:
- 2-Hour serum insulin (mu U/ml) ####BMI:
- Body mass index (weight in kg)/(height in m^2) ####DiabetiesPedigree Function:
- Diabetes pedigree function ####Age:
- Age in years

##Target Variable ####Outome:

- Class Variable (0 or 1)
- 268 instances are labeled as 1 (indicating diabetes), while others are labeled as 0.

0.1 Import Libraries

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import␣
 ↪train_test_split,GridSearchCV,cross_val_score
import seaborn as sns
```

```
from sklearn.metrics import␣
 ↪accuracy_score,mean_squared_error,classification_report,confusion_matrix,precision_score,re
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2
from sklearn.ensemble import ExtraTreesClassifier,RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB
from sklearn.preprocessing import StandardScaler,MinMaxScaler
from sklearn.neighbors import KNeighborsClassifier
```

0.2 Read the data

[2]:
```
data=pd.read_csv('/content/health care diabetes.csv')
data.head()
```

[2]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	\
0	6	148	72	35	0	33.6	
1	1	85	66	29	0	26.6	
2	8	183	64	0	0	23.3	
3	1	89	66	23	94	28.1	
4	0	137	40	35	168	43.1	

	DiabetesPedigreeFunction	Age	Outcome
0	0.627	50	1
1	0.351	31	0
2	0.672	32	1
3	0.167	21	0
4	2.288	33	1

0.3 Check number of rows and columns

[3]:
```
data.shape
```

[3]:
```
(768, 9)
```

0.4 Shows data and datatypes and null values

[4]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 768 entries, 0 to 767
Data columns (total 9 columns):
 #   Column                    Non-Null Count  Dtype
---  ------                    --------------  -----
 0   Pregnancies               768 non-null    int64
```

```
 1   Glucose                   768 non-null    int64
 2   BloodPressure             768 non-null    int64
 3   SkinThickness             768 non-null    int64
 4   Insulin                   768 non-null    int64
 5   BMI                       768 non-null    float64
 6   DiabetesPedigreeFunction  768 non-null    float64
 7   Age                       768 non-null    int64
 8   Outcome                   768 non-null    int64
dtypes: float64(2), int64(7)
memory usage: 54.1 KB
```

0.5 Summary statistics for dataset. Shows only numerical values.

[5]:
```
data.describe()
```

[5]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	\
count	768.000000	768.000000	768.000000	768.000000	768.000000	
mean	3.845052	120.894531	69.105469	20.536458	79.799479	
std	3.369578	31.972618	19.355807	15.952218	115.244002	
min	0.000000	0.000000	0.000000	0.000000	0.000000	
25%	1.000000	99.000000	62.000000	0.000000	0.000000	
50%	3.000000	117.000000	72.000000	23.000000	30.500000	
75%	6.000000	140.250000	80.000000	32.000000	127.250000	
max	17.000000	199.000000	122.000000	99.000000	846.000000	

	BMI	DiabetesPedigreeFunction	Age	Outcome
count	768.000000	768.000000	768.000000	768.000000
mean	31.992578	0.471876	33.240885	0.348958
std	7.884160	0.331329	11.760232	0.476951
min	0.000000	0.078000	21.000000	0.000000
25%	27.300000	0.243750	24.000000	0.000000
50%	32.000000	0.372500	29.000000	0.000000
75%	36.600000	0.626250	41.000000	1.000000
max	67.100000	2.420000	81.000000	1.000000

0.6 DATA PREPROCESSING:TREAT MISSING VALUES

In this dataset,0 represents the null values, and hence we will replace 0 by means of their feature(variable) columns.

[6]:
```
# Identifying the mean of the features
print(data['Glucose'].mean())
print(data['BloodPressure'].mean())
print(data['SkinThickness'].mean())
print(data['Insulin'].mean())
```

```
120.89453125
69.10546875
```

```
20.536458333333332
79.79947916666667
```

0.7 To find number of rows which has null values

[7]:
```
print('Glucose-',len(data['Glucose'][data['Glucose']==0]))
print('BloodPressure-',len(data['BloodPressure'][data['BloodPressure']==0]))
print('SkinThickness-',len(data['SkinThickness'][data['SkinThickness']==0]))
print('Insulin-',len(data['Insulin'][data['Insulin']==0]))
```

```
Glucose- 5
BloodPressure- 35
SkinThickness- 227
Insulin- 374
```

0.8 Finding the null value percentage

[8]:
```
selected_columns = ['Glucose', 'BloodPressure', 'SkinThickness','Insulin']
null_percentage = (data[selected_columns] == 0).mean() * 100

# Displaying the null value percentage for each selected column
print("Percentage of Null Values for Each Column:")
print(null_percentage)
```

```
Percentage of Null Values for Each Column:
Glucose           0.651042
BloodPressure     4.557292
SkinThickness    29.557292
Insulin          48.697917
dtype: float64
```

##Inference For Null Value Percentage Analysis **Glucose**: Low percentage of null values indicates minimal missing data, providing confidence in the integrity of glucose measurements.

BloodPressure: A moderate percentage of null values indicates some variability in blood pressure measurements.

SkinThickness: A relatively high percentage of null values suggests a substantial amount of missing or undefined data in skin thickness measurements, requiring careful handling or imputation during analysis.

Insulin: With high percentage of null values, we can understand that whether these represent actual measurements or signify missing or undefined data, impacting the reliability of insulin-related insights.

0.9 Replacing the null values with the mean

[9]:
```
data['Glucose']=data['Glucose'].replace([0],[data['Glucose'].mean()])
data['BloodPressure']=data['BloodPressure'].replace([0],[data['BloodPressure'].
 ↪mean()])
data['SkinThickness']=data['SkinThickness'].replace([0],[data['SkinThickness'].
 ↪mean()])
data['Insulin']=data['Insulin'].replace([0],[data['Insulin'].mean()])
```

[10]:
```
#Checking the null value percentage of the treated columns
null_percentage_treated = (data[selected_columns] == 0).mean() * 100
```

[11]:
```
# Displaying the null value percentage for each selected column
print("Percentage of Null Values for Each Column after the null value treatment:
 ↪")
print(null_percentage_treated)
```

```
Percentage of Null Values for Each Column after the null value treatment:
Glucose          0.0
BloodPressure    0.0
SkinThickness    0.0
Insulin          0.0
dtype: float64
```

##Inference for Null Values Treatment The output indicates that, after null value treatment that is imputation by replacing zeros with column means, there are no zero values remaining in the treated columns. Now the data appears to be free of zero values in the treated columns, allowing for a more reliable analysis.

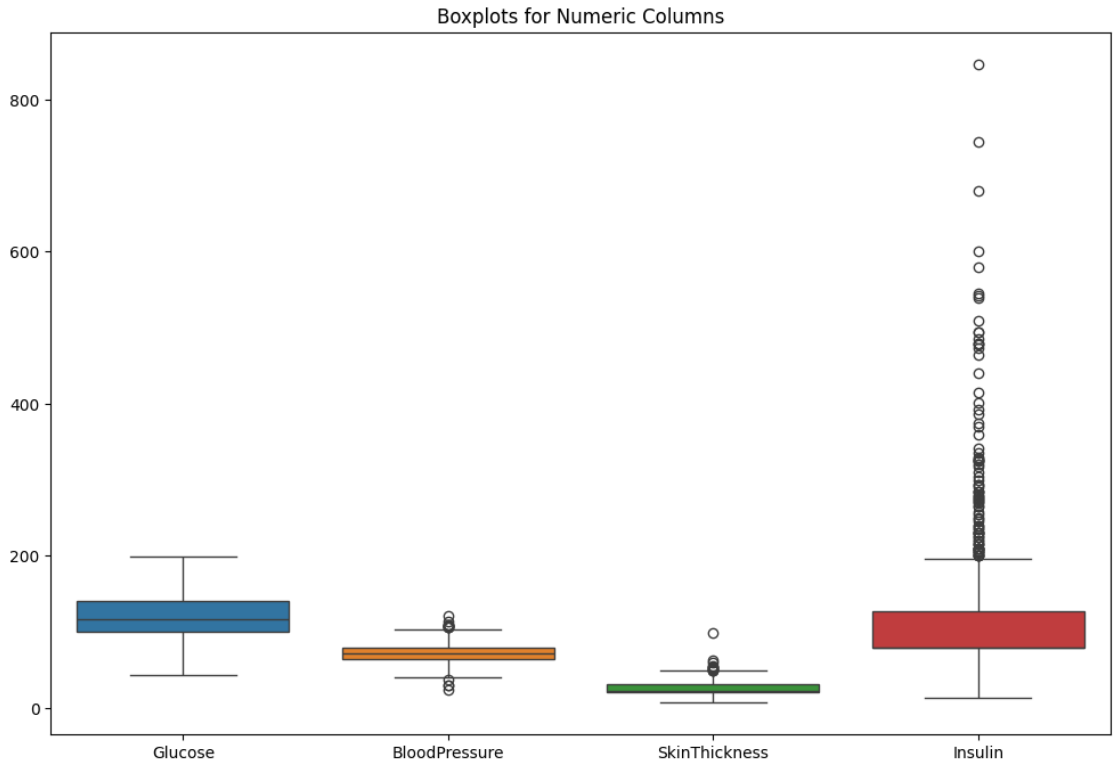
0.10 Detecting Outliers and Treatments

[12]:
```
columns=data[selected_columns]
columns.columns
```

[12]:
```
Index(['Glucose', 'BloodPressure', 'SkinThickness', 'Insulin'], dtype='object')
```

0.11 Display boxplots for numeric columns to visualize outliers

[13]:
```
plt.figure(figsize=(12, 8))
sns.boxplot(data=columns)
plt.title("Boxplots for Numeric Columns")
plt.show()
```



[14]:
```
# Finding the Outlier Count in the selected Columns:
def find_outliers_iqr(data, column_name):
    # Calculate the first quartile (Q1) and third quartile (Q3)
    Q1 = data[column_name].quantile(0.25)
    Q3 = data[column_name].quantile(0.75)

    # Calculate the interquartile range (IQR)
    IQR = Q3 - Q1

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

    # Find outliers
    outliers = data[(data[column_name] < lower_bound) | (data[column_name] >␣
 ↪upper_bound)]

    # Count the number of outliers
    count_outliers = len(outliers)

    return count_outliers
```

[15]:
```
# Calculate and print the number of outliers for each column of interest
for column_name in selected_columns:
    outlier_count = find_outliers_iqr(data, column_name)
    print(f"Number of outliers in the '{column_name}' column: {outlier_count}")
```

```
Number of outliers in the 'Glucose' column: 0
Number of outliers in the 'BloodPressure' column: 14
Number of outliers in the 'SkinThickness' column: 12
Number of outliers in the 'Insulin' column: 89
```

##Box plot/whishker plot - for visualizing the outliers - lower whisker line, upper whisker line.

IQR = Inter quartile range.

identation - for loop, while loop, conditions, user defined function. q1 - quantile 1 - 25th percentile q3 - quantile 3 - 75th percentile

data[condition] - filtering (extracting few rows from the dataset)

0.11.1 Boxplot Analysis for Numerical Columns

The boxplot illustrates the distribution of four numerical columns: Glucose, BloodPressure, Skin Thickness, and Insulin. The following inferences can be drawn:

Glucose

- Median glucose level: ~200 mg/dL
- IQR is large, indicating considerable variability in glucose levels.
- There are no outliers

Blood Pressure

- Median blood pressure: 72 mmHg (within the normal range).
- IQR is relatively small, suggesting more consistent blood pressure levels.
- Few outliers, none extremely high or low.

Skin Thickness

- Median skin thickness: ~25 mm
- IQR is small, indicating less considerable variability in skin thickness.
- Few outliers, none extremely high.

Insulin

- Median insulin level: ~79 mIU/L
- IQR is large, indicating considerable variability in insulin levels.
- More outliers, many are extremely high.

Overall Observations

- All columns exhibit a wide range of values, with some outliers. Insulin column has many outliers

- Median values for all columns, except the insulin column fall within the normal range.

Additional Inferences

- Glucose levels show more variability than blood pressure levels.
- More outliers in the insulin columns compared to blood pressure and skin thickness.

Possible Interpretations

- Variability in glucose levels may be influenced by factors like diet, exercise, and stress.
- Outliers in the Insulin column may also be associated with underlying medical conditions or physiological factors. Elevated insulin levels could be indicative of conditions such as insulin resistance or diabetes. Additionally, factors such as dietary habits, genetic predisposition, or specific medical treatments may contribute to higher insulin levels. Further investigation and domain expertise are necessary to understand the potential health implications of these outliers in the Insulin column.

It is essential to note that these inferences are based on a single boxplot, and further information is needed to draw definitive conclusions.

###Outlier Treatment

[16]:
```
sorted(data)
Q1=data.quantile(0.20)
Q3=data.quantile(0.80)
IQR=Q3-Q1
print(IQR)
```

```
Pregnancies                  6.000000
Glucose                     52.000000
BloodPressure               20.000000
SkinThickness               14.463542
Insulin                     70.200521
BMI                         11.900000
DiabetesPedigreeFunction     0.467600
Age                         19.600000
Outcome                      1.000000
dtype: float64
```

[17]:
```
data_cleared_iqr = data[~((data < (Q1 - 1.5 * IQR)) |(data > (Q3 + 1.5 * IQR))).
 ↪any(axis=1)]
data_cleared_iqr
print(data_cleared_iqr.shape)
print(data.shape)
```

```
(678, 9)
(768, 9)
```

[18]:
```
data_cleared_iqr.head()
```

[18]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148.0	72.0	35.000000	79.799479	33.6	0.627	50	1
1	1	85.0	66.0	29.000000	79.799479	26.6	0.351	31	0
2	8	183.0	64.0	20.536458	79.799479	23.3	0.672	32	1
3	1	89.0	66.0	23.000000	94.000000	28.1	0.167	21	0
5	5	116.0	74.0	20.536458	79.799479	25.6	0.201	30	0

##Inferences from Outlier Removal using IQR Method

Data Size Reduction: After removing outliers using the interquartile range (IQR) method, the dataset has been reduced from 768 to 678 rows.

Outliers Identified: Outliers were detected and removed across various columns, particularly impacting features like Glucose, Blood Pressure, Skin Thickness, Insulin, BMI, and Age.

Increased Data Robustness: The IQR-based outlier removal contributes to a more robust dataset, potentially improving the reliability of statistical analyses and modeling.

Preserved Features: The operation was applied to 9 columns, including predictors like Glucose and Skin Thickness, as well as the target variable Outcome.

Consideration for Domain Knowledge: The decision to remove outliers should be made with consideration for domain knowledge, as outliers may contain valuable information or indicate specific health conditions.

Final Dataset Statistics: Dataset size after outlier removal: 678 rows. Original dataset size: 768 rows.

[19]:
```
col=data_cleared_iqr[['Glucose','BloodPressure','SkinThickness','Insulin']]
```

[20]:
```
type(col)
```

[20]:
```
pandas.core.frame.DataFrame
```

[21]:
```
col.head()
```

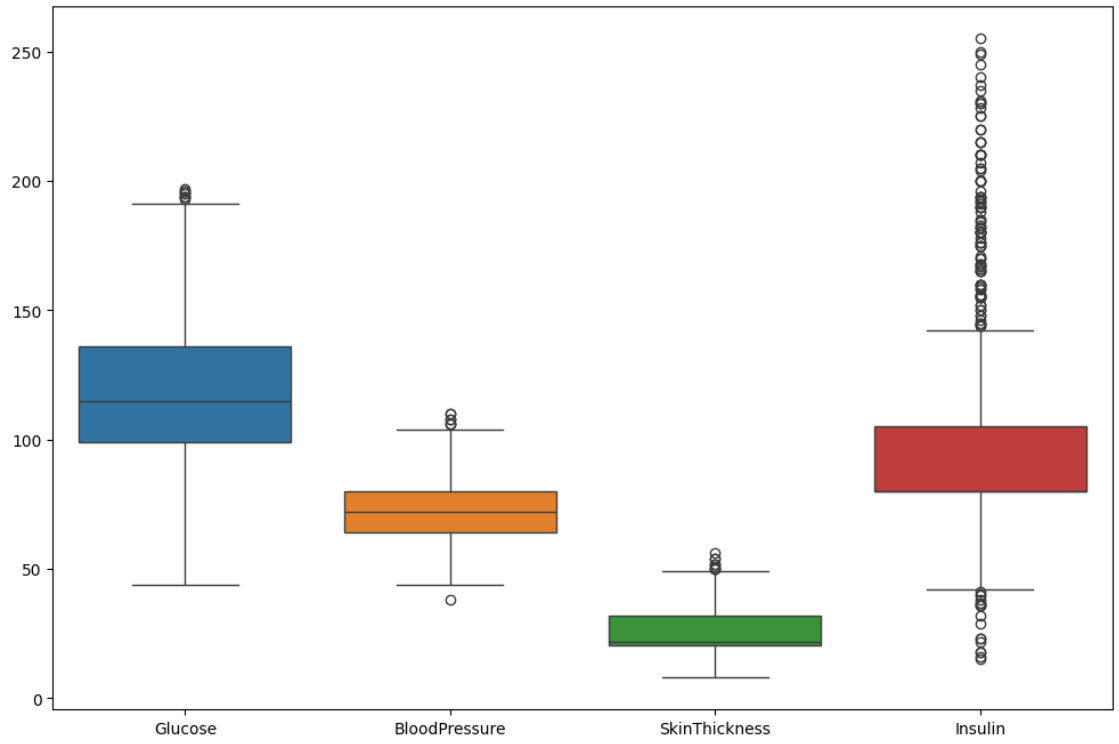
[21]:

	Glucose	BloodPressure	SkinThickness	Insulin
0	148.0	72.0	35.000000	79.799479
1	85.0	66.0	29.000000	79.799479
2	183.0	64.0	20.536458	79.799479
3	89.0	66.0	23.000000	94.000000
5	116.0	74.0	20.536458	79.799479

[22]:
```
col.shape
```

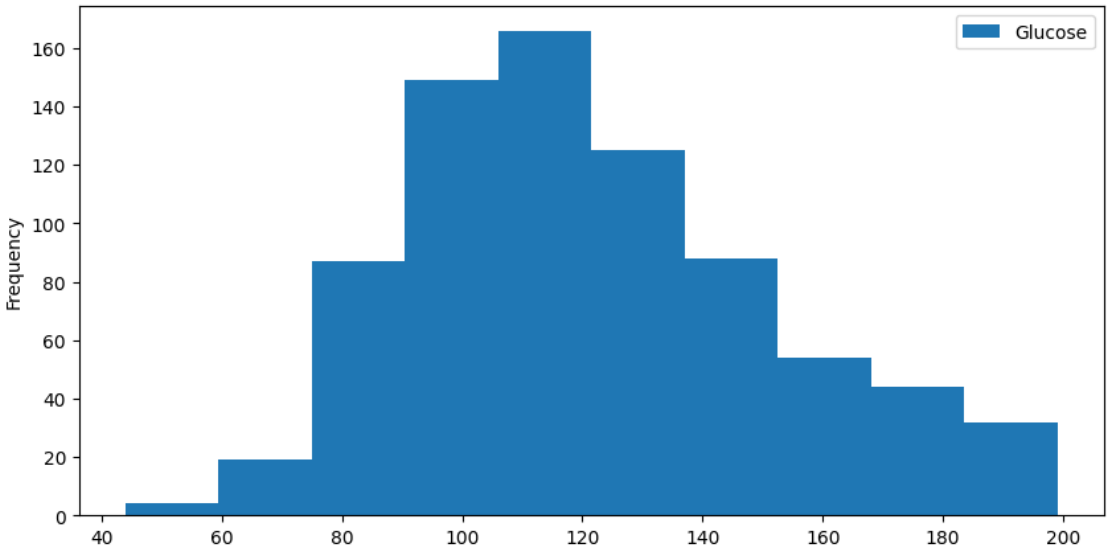
[22]: (678, 4)

[23]:
```
#checking the outliers after treatment using box plot
plt.figure(figsize=(12, 8))
sns.boxplot(data=col)
plt.show()
```



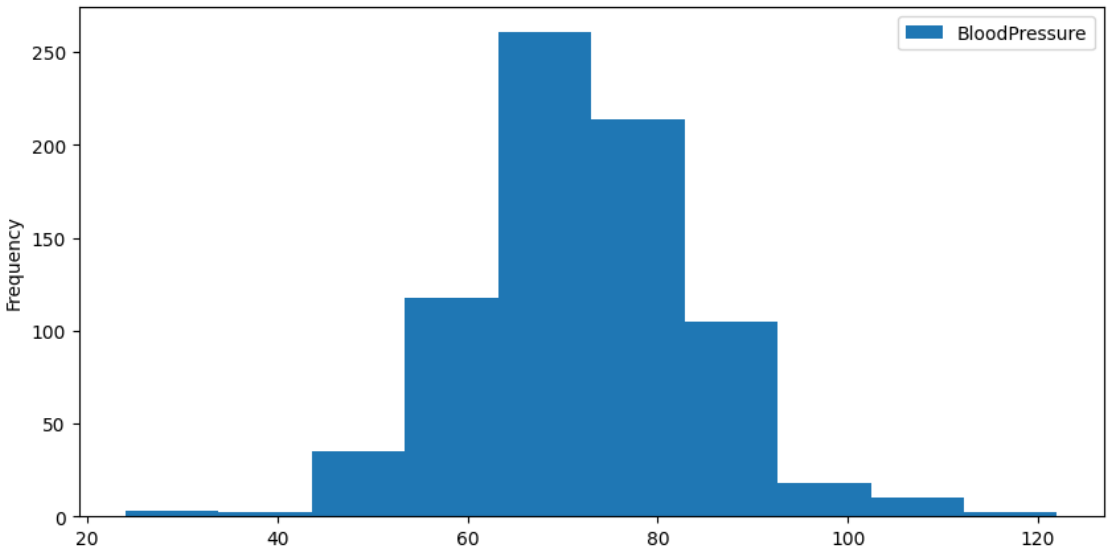
#EDA-univariate analysis for each feature

[24]:
```
#Visually exploring variables using histograms
data['Glucose'].plot(kind='hist',figsize=(10,5))
plt.legend()
plt.show()
```



##Inference of Glucose distribution The histogram represents distribution of glucose levels. The x-axis displays the glucose levels, and the y-axis shows the frequency of occurrences. Most of the data is concentrated within a certain range, as indicated by the peaks in the histogram that is between 100-120.

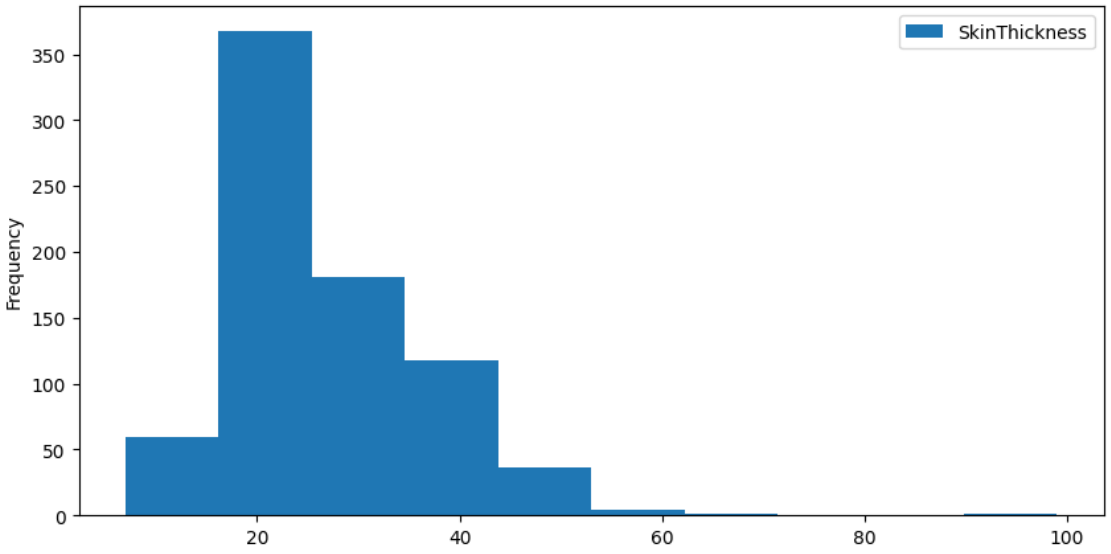
[25]:
```
data['BloodPressure'].plot(kind='hist',figsize=(10,5))
plt.legend()
plt.show()
```



###Inference of BloodPressure distribution The histogram represents distribution of glucose lev-

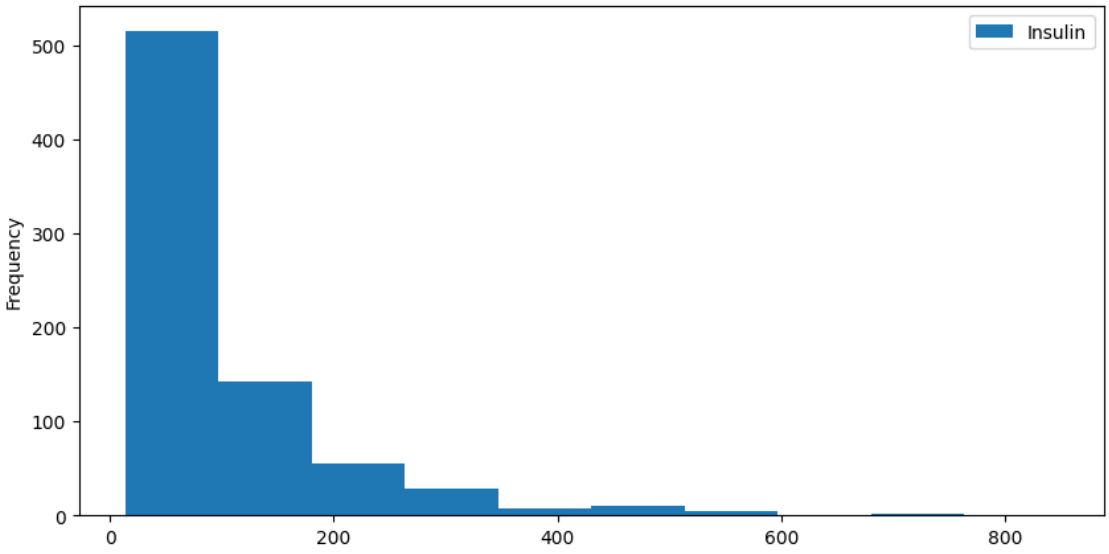
els. The x-axis displays the glucose levels, and the y-axis shows the frequency of occurrences. Most of the data is concentrated within a certain range, as indicated by the peaks in the histogram that is between 63-73.

[26]:
```
data['SkinThickness'].plot(kind='hist',figsize=(10,5))
plt.legend()
plt.show()
```



###Inference of SkinThickness distribution The histogram represents distribution of glucose levels. The x-axis displays the glucose levels, and the y-axis shows the frequency of occurrences. Most of the data is concentrated within a certain range, as indicated by the peaks in the histogram that is between 18-25.

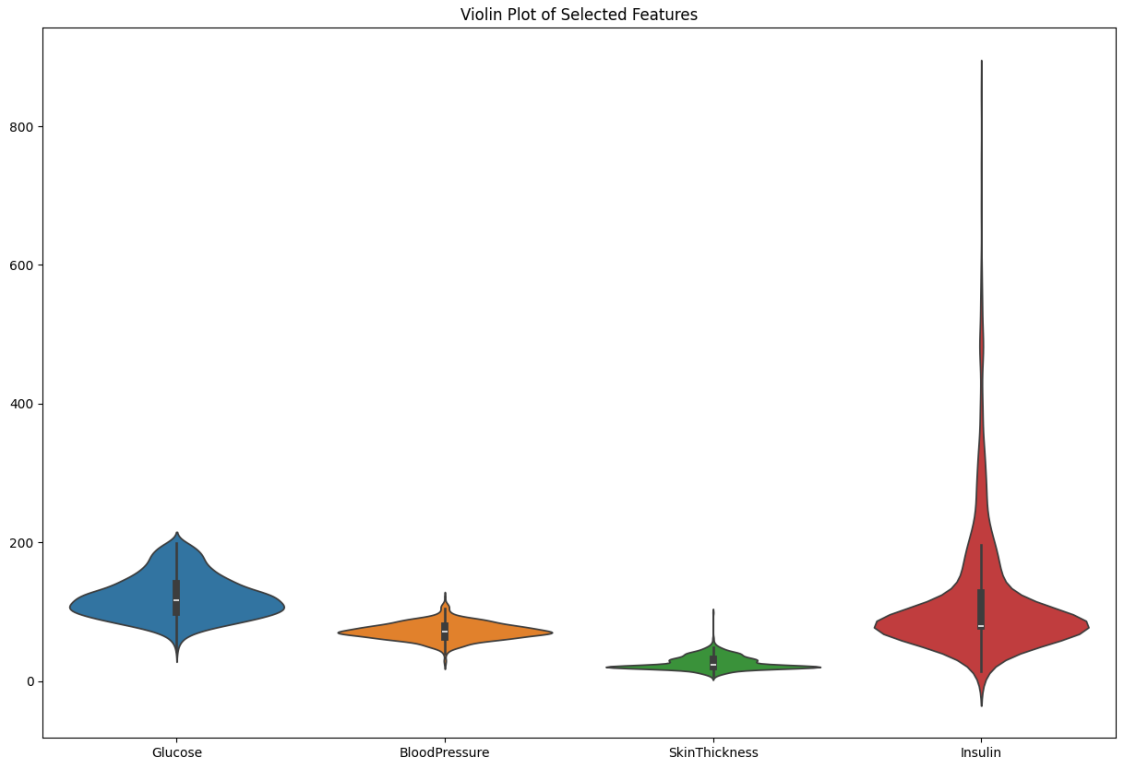
[27]:
```
data['Insulin'].plot(kind='hist',figsize=(10,5))
plt.legend()
plt.show()
```



###Inference of Insulin distribution The histogram represents distribution of glucose levels. The x-axis displays the glucose levels, and the y-axis shows the frequency of occurrences. Most of the data is concentrated within a certain range, as indicated by the peaks in the histogram that is between 2-13.

#Violin plot for the selected features

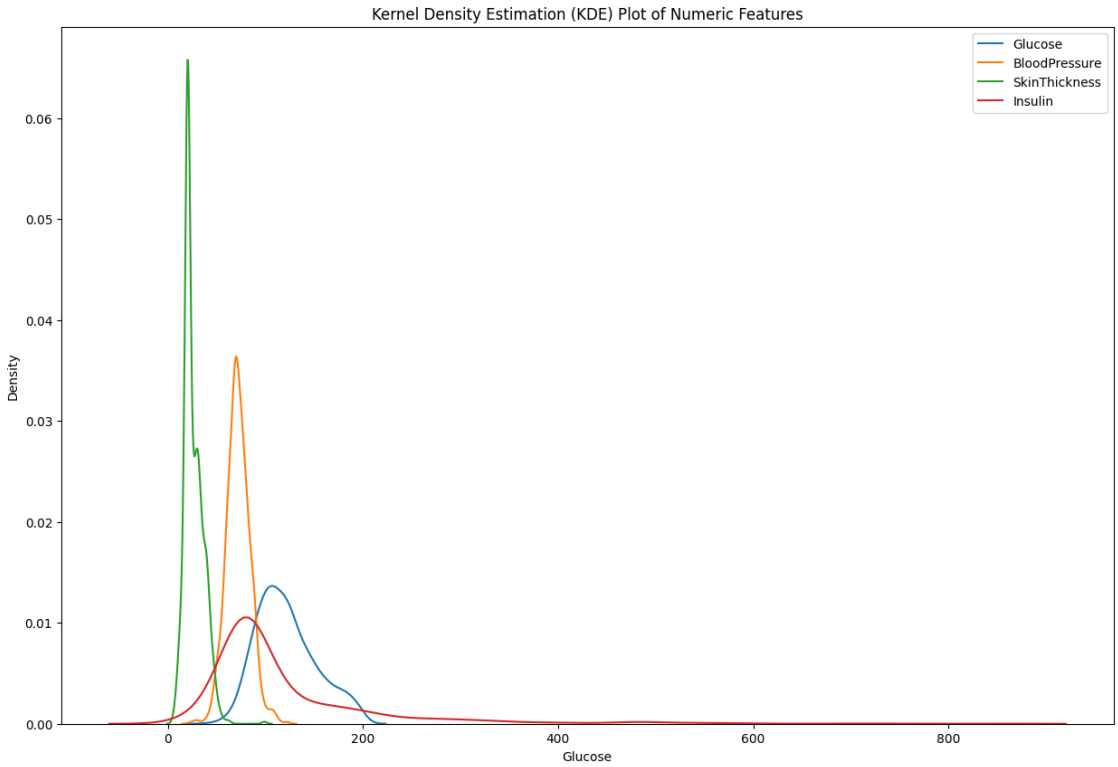
[28]:
```
plt.figure(figsize=(15, 10))
sns.violinplot(data=data[selected_columns])
plt.title("Violin Plot of Selected Features")
plt.show()
```



The violin plot shows the distribution of four numerical features: Glucose, BloodPressure, Skin Thickness, and Insulin. The violin shape represents the probability density function (PDF) of each feature, and the box plot embedded within each violin plot shows the median, interquartile range (IQR), and outliers.

##Kernal Density Estimaton plot for selected features

[29]:
```
plt.figure(figsize=(15, 10))
for column in selected_columns:
    sns.kdeplot(data[column], label=column)
plt.title("Kernel Density Estimation (KDE) Plot of Numeric Features")
plt.legend()
plt.show()
```



The image shows a Kernel Density Estimation (KDE) plot of four numerical features: Glucose, BloodPressure, Skin Thickness, and Insulin. KDE is a non-parametric method for estimating the probability density function (PDF) of a random variable. The KDE plot shows the estimated PDF of each feature, which can be used to visualize the distribution of the data.

###Creating a count(frequency) plot describing the data types and the count of variables

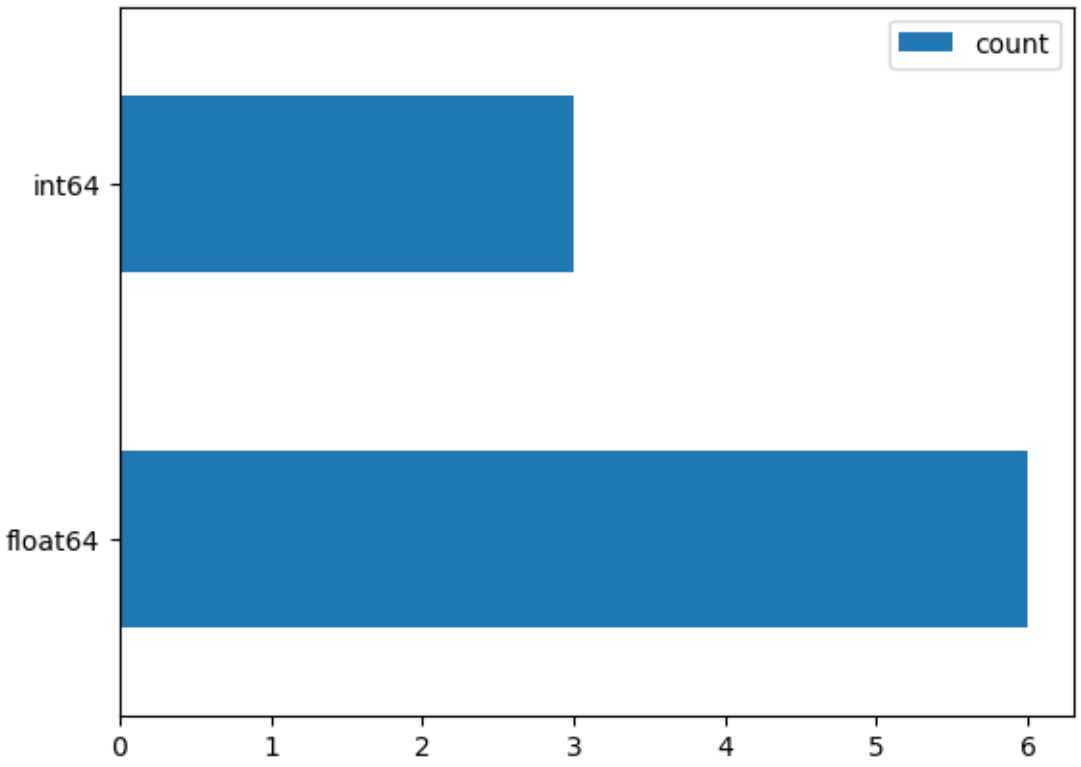
[30]:
```
data.dtypes
```

[30]:
```
Pregnancies                 int64
Glucose                   float64
BloodPressure             float64
SkinThickness             float64
Insulin                   float64
BMI                       float64
DiabetesPedigreeFunction  float64
Age                         int64
Outcome                     int64
dtype: object
```

[31]:
```
#count of every datatype
data.dtypes.value_counts()
```

[31]:
```
float64    6
int64      3
Name: count, dtype: int64
```

[32]:
```
figsize=(16,2)
data.dtypes.value_counts().plot(kind='barh')
plt.legend()
plt.show()
```



It can be observed that there are three features of integer data type and six float data type

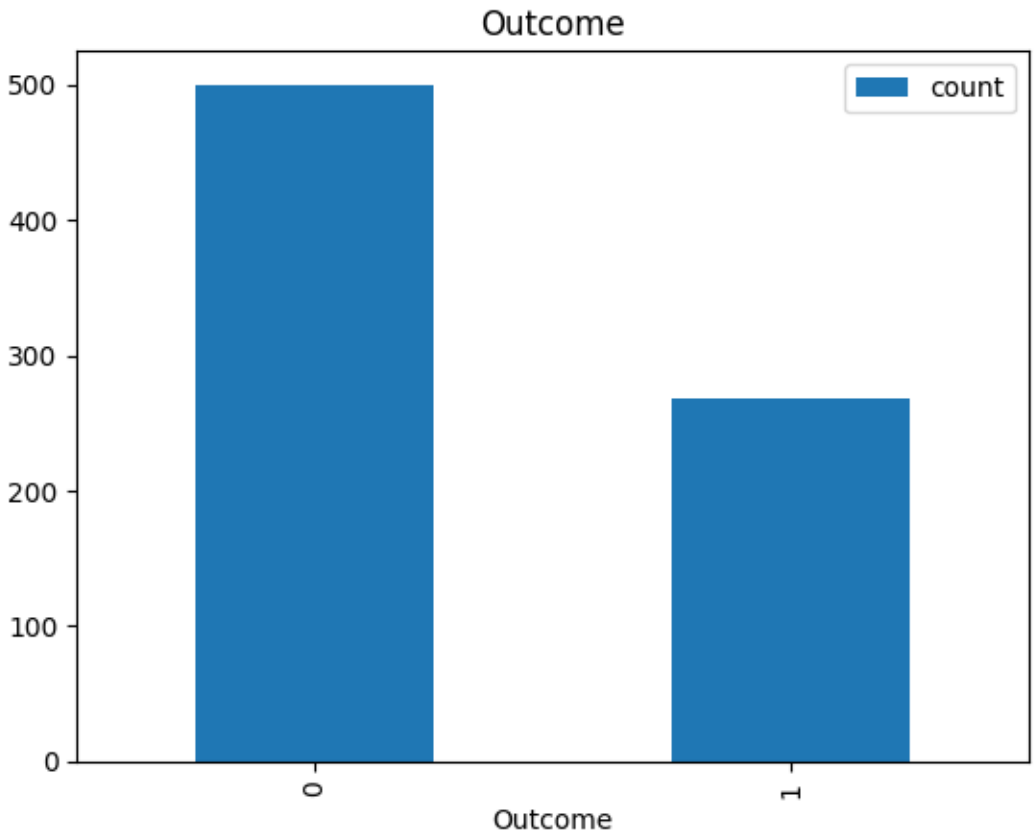
##DATA EXPLORATION

check the balance of the data by plotting the count of outcomes by their value.

[33]:
```
data['Outcome'].value_counts()
```

[33]:
```
Outcome
0    500
1    268
Name: count, dtype: int64
```

[34]:
```
data['Outcome'].value_counts().plot(kind='bar')
plt.legend()
plt.title('Outcome')
plt.show()
```



[35]:
```
#Finding outcome percentage
data['Outcome'].value_counts(1)
```

[35]:
```
Outcome
0    0.651042
1    0.348958
Name: proportion, dtype: float64
```

[36]:
```
outcome=(data['Outcome'].value_counts()/data['Outcome'].shape)*100
outcome
```

[36]:
```
Outcome
0    65.104167
1    34.895833
Name: count, dtype: float64
```

##Inference from outcome distribution

Class Imbalance: The dataset exhibits class imbalance in the 'Outcome' variable. Class 0 (No Diabetes) has 500 instances. Class 1 (Diabetes) has 268 instances.

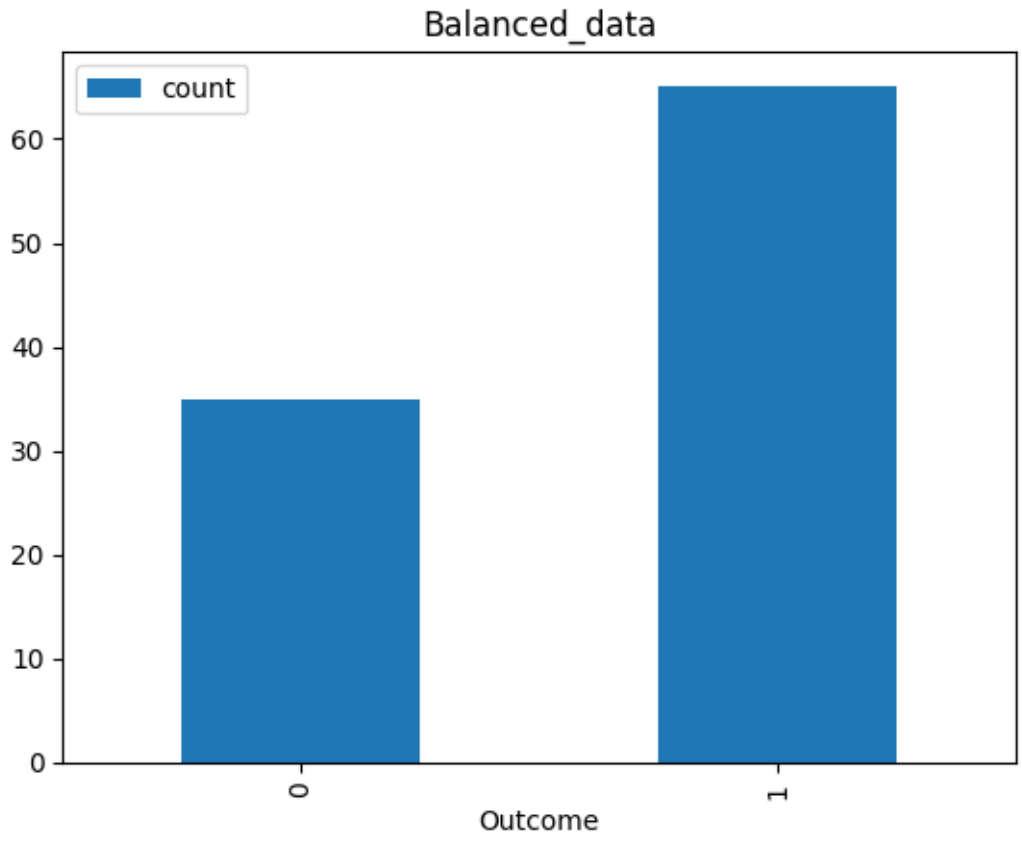
Potential Impact on Modeling: Class imbalances may affect the performance of machine learning models, particularly for binary classification tasks. Addressing class imbalance through techniques like resampling or using appropriate evaluation metrics may be necessary.

Consideration for Predictive Models: Models may need to be evaluated and tuned considering the imbalanced distribution to avoid biased predictions toward the majority class.

[37]:
```
balanced_data=100-outcome
balanced_data
```

[37]:
```
Outcome
0    34.895833
1    65.104167
Name: count, dtype: float64
```

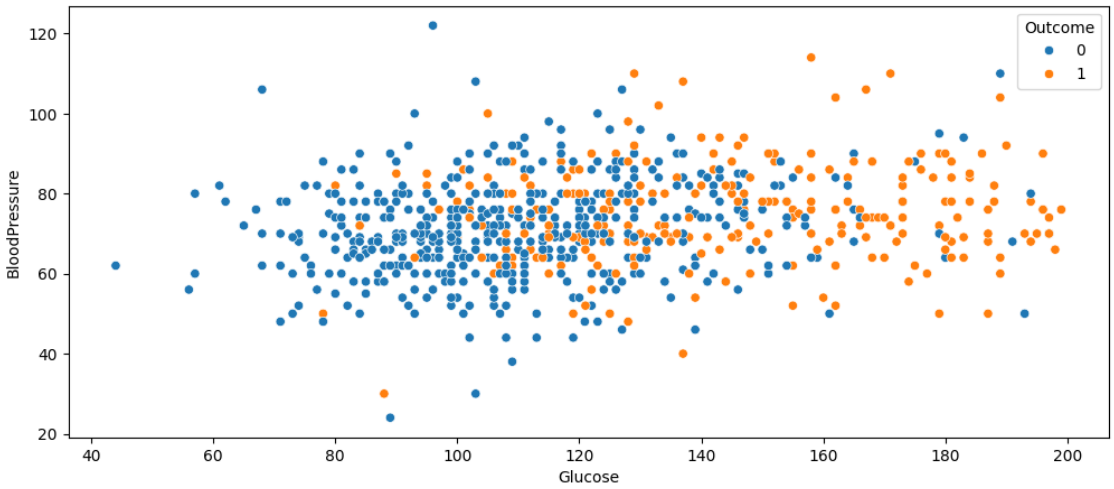
[38]:
```
balanced_data.plot(kind='bar')
plt.legend()
plt.title('Balanced_data')
plt.show()
```



##Bi-Variate Analysis

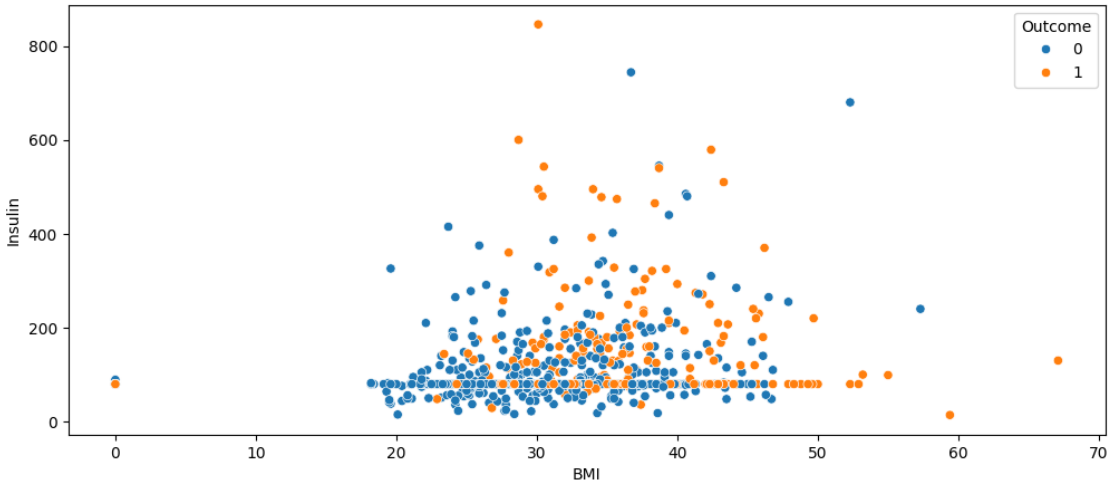
creating scatter plot charts btw the pair of variables to understand the relationships

[39]:
```
plt.figure(figsize=(12,5))
sns.scatterplot(x='Glucose',y='BloodPressure',hue='Outcome',data=data)
plt.show()
```



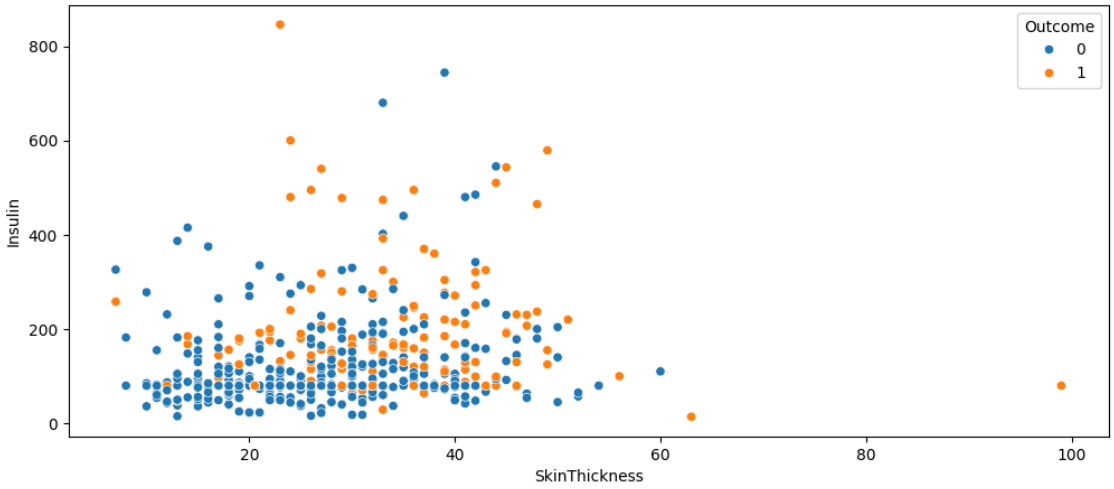
In this scatter plot, we can observe a diverse distribution of data points across the Glucose-BloodPressure space.We notice some clustering, particularly among points 60-80 for Bloodpressure and 80-120 for Glucose.

[40]:
```
plt.figure(figsize=(12,5))
sns.scatterplot(x='BMI',y='Insulin',hue='Outcome',data=data)
plt.show()
```



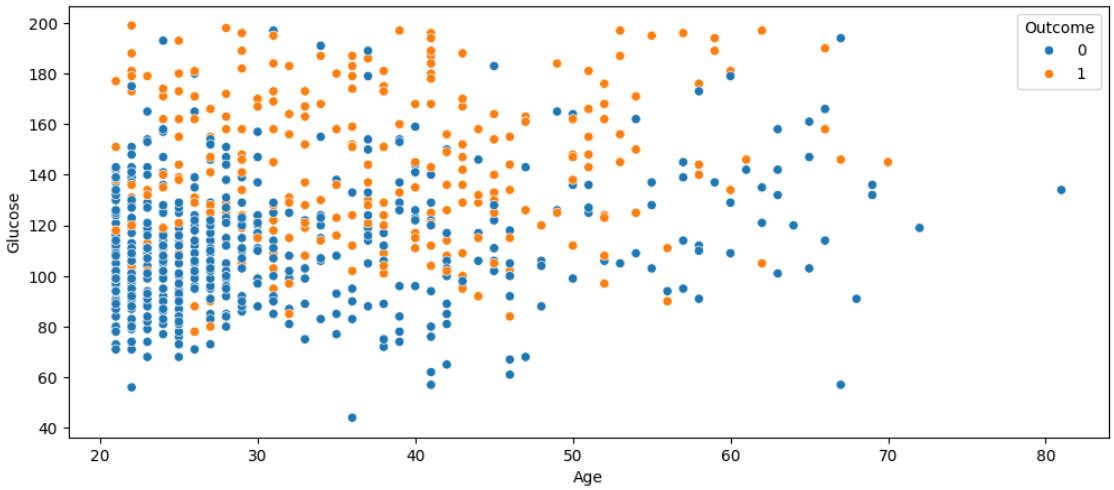
In this scatter plot, we can observe a diverse distribution of data points across the BMI-Insulin space.We notice some clustering, particularly among points 0-200 for Insulin and 20-40 for BMI.

[41]:
```
plt.figure(figsize=(12,5))
sns.scatterplot(x='SkinThickness',y='Insulin',hue='Outcome',data=data)
plt.show()
```



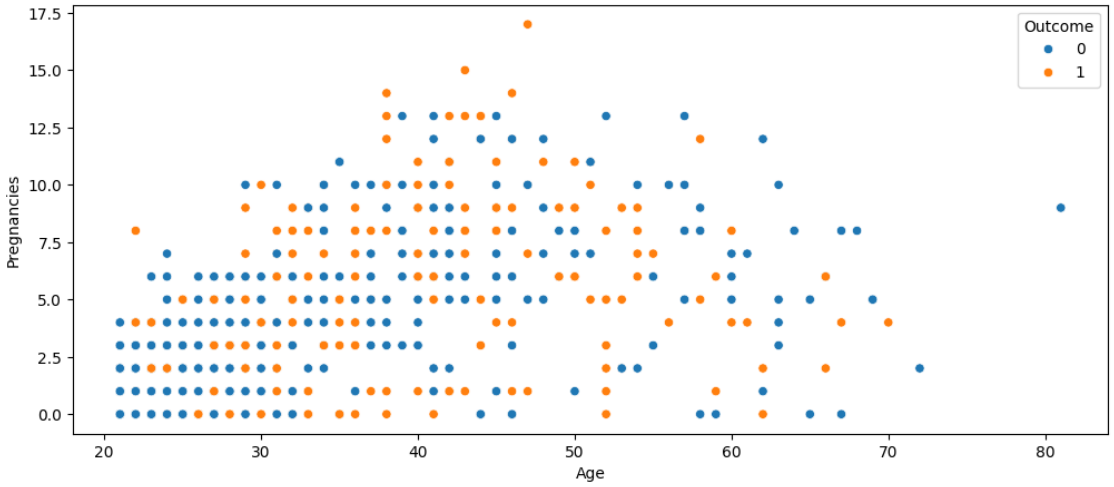
In this scatter plot, we can observe a diverse distribution of data points across the SkinThickness-Insulin space.We notice some clustering, particularly among points 0-200 for Insulin and 10-40 for SkinThickness.

[42]:
```
plt.figure(figsize=(12,5))
sns.scatterplot(x='Age',y='Glucose',hue='Outcome',data=data)
plt.show()
```



In this scatter plot, we can observe a diverse distribution of data points across the Age-Glucose space.We notice some clustering, particularly among points 20-30 for Age and 100-180 for Glucose.

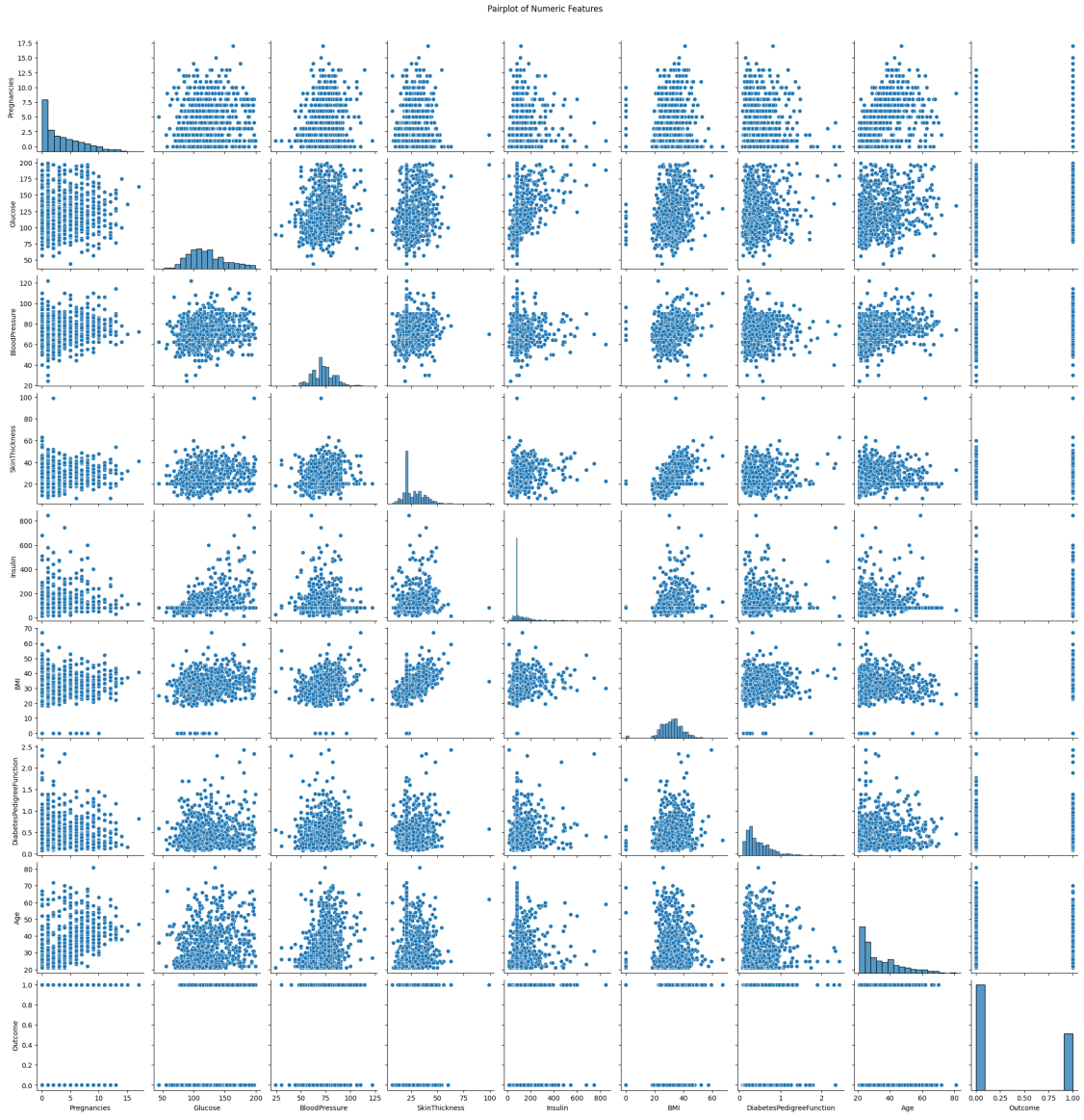
[43]:
```
plt.figure(figsize=(12,5))
sns.scatterplot(x='Age',y='Pregnancies',hue='Outcome',data=data)
plt.show()
```



In this scatter plot, we can observe a diverse distribution of data points across the Age-Pregnancies space.We notice some clustering, particularly among points 20-45 for Age and 0.0-10.0 for Pregnancies.

##PAIR PLOT

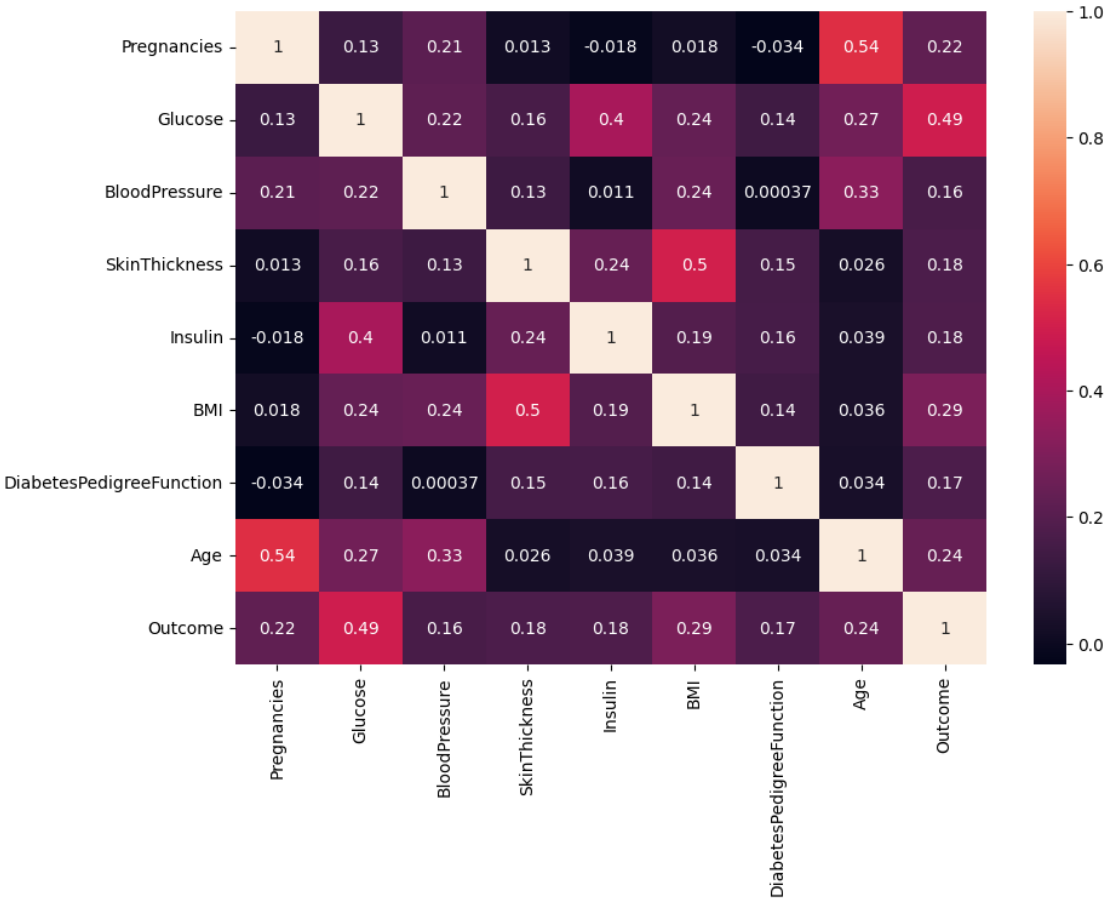
[44]:
```
sns.pairplot(data)
plt.suptitle("Pairplot of Numeric Features", y=1.02)
plt.show()
```



##Multi-Variate Analysis

Perform correlation analysis. Visually explore it using heat map.

[45]:
```
plt.figure(figsize=(10,7))
sns.heatmap(data.corr(),annot=True)
plt.show()
```



We can see outcome has maximum relation with Glucose and minimum with Blood Pressure than the other features.

##DATA MODELLING

strategies for model building :-

```
1. Descriptive Analysis :-
    -Identify ID, Input and Target features
    -Identify categorical and numerical features
    -Identify columns with missing values

2. Data Treatment (Missing values treatment) :-
    - Detecting outliers & removing them.
    - Imputing mean, mode or median value at a place of missing value as per dataset

3. Feature Extraction / Feature Engineering :-
    -we will remove noisy features from data
    -By the help of correlation / heatmap / differnt types of feature selection techniques.

4. Data is imbalanced
```

-For balancing the data we wil use SMOTE over sampling techinque.

5. Building a model :-
   - select a best algorithms for model

6. Train a model

7. Evaluation
   - check a accuracy & mean squared error of model

8. Hyper Parameter Tuning :-
   -for decrese in RMSE check a best parameters for model.

9. Create a clasification report.

###Feature Selection

[46]:
```
# Data preparation for modeling
x=data.drop(['Outcome'],axis=1)
y=data.Outcome
```

[47]:
```
# Finding the correlation of every feature with the outcome (Target Variable)
data.corrwith(data['Outcome'])
```

[47]:
```
Pregnancies                 0.221898
Glucose                     0.492908
BloodPressure               0.162986
SkinThickness               0.175026
Insulin                     0.179185
BMI                         0.292695
DiabetesPedigreeFunction    0.173844
Age                         0.238356
Outcome                     1.000000
dtype: float64
```

[48]:
```
bestfeatures = SelectKBest(score_func=chi2, k='all')
fit = bestfeatures.fit(x,y)
dfscores = pd.DataFrame(fit.scores_)
dfcolumns = pd.DataFrame(x.columns)
#concat two dataframes for better visualization
featureScores = pd.concat([dfcolumns,dfscores],axis=1)
featureScores.columns = ['Specs','Score'] #naming the dataframe columns
print(featureScores.nlargest(8,'Score')) #print 10 best features
```

```
     Specs        Score
4  Insulin  1798.088682
1  Glucose  1418.660636
7      Age   181.303689
```

```
5                         BMI   127.669343
0                 Pregnancies   111.519691
3               SkinThickness    81.917622
2               BloodPressure    41.394665
6   DiabetesPedigreeFunction     5.392682
```

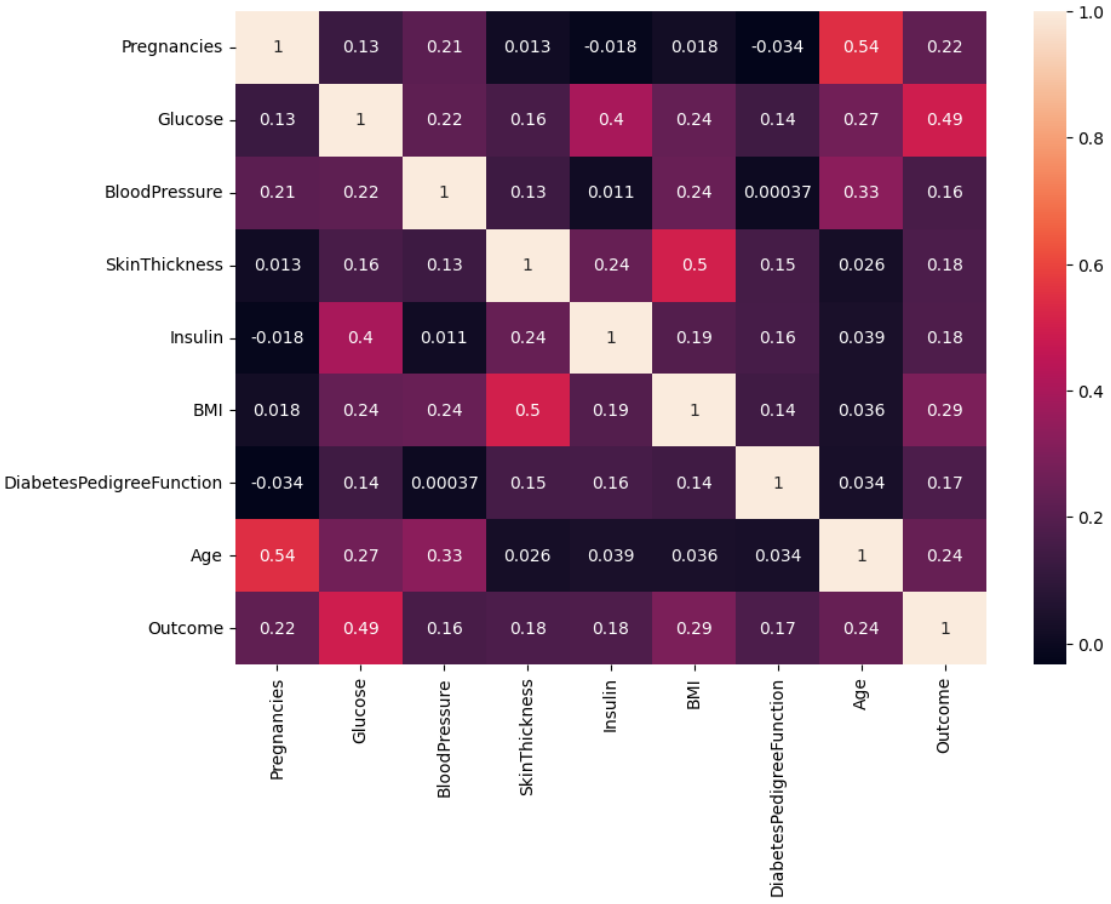
[49]:
```
type(fit)
```

[49]:
```
sklearn.feature_selection._univariate_selection.SelectKBest
```

[50]:
```
fit.scores_
```

[50]:
```
array([ 111.51969064, 1418.66063574,   41.39466535,   81.91762154,
        1798.08868209,  127.66934333,    5.39268155,  181.30368904])
```

[51]:
```
plt.figure(figsize=(10,7))
sns.heatmap(data.corr(),annot=True)
plt.show()
```



- We can see BloodPressure features has lowest relation with output column.
- So we can remove BloodPressure for training a good model with high accuracy.

[52]:
```
new_x=data.drop(['Outcome','BloodPressure'],axis=1).values
new_y=data.Outcome.values
```

##SMOTE to adress the class Imbalance

###Train a model

[53]:
```
# Train-Test Split for Data Modeling
 trainx,testx,trainy,testy=train_test_split(new_x,new_y,test_size=0.
 ↪20,random_state=10)
```

[54]:
```
print("Before OverSampling, counts of label '1': {}".format(sum(trainy == 1)))
print("Before OverSampling, counts of label '0': {} \n".format(sum(trainy ==␣
 ↪0)))
from imblearn.over_sampling import SMOTE
sm = SMOTE(random_state =63)
trainx_res,trainy_res = sm.fit_resample(trainx,trainy.ravel())
print('After OverSampling, the shape of train_X: {}'.format(trainx_res.shape))
print('After OverSampling, the shape of train_y: {} \n'.format(trainy_res.
 ↪shape))
print("After OverSampling, counts of label '1': {}".format(sum(trainy_res ==␣
 ↪1)))
print("After OverSampling, counts of label '0': {}".format(sum(trainy_res ==␣
 ↪0)))
```

```
Before OverSampling, counts of label '1': 209
Before OverSampling, counts of label '0': 405

After OverSampling, the shape of train_X: (810, 7)
After OverSampling, the shape of train_y: (810,)

After OverSampling, counts of label '1': 405
After OverSampling, counts of label '0': 405
```

##Applying an appropriate classification algorithm to build a model.

###Model-1 Building a logistic Regession Model

[55]:
```
logreg=LogisticRegression(solver='liblinear',random_state=123)
```

[56]:
```
logreg.fit(trainx_res,trainy_res)
```

[56]:
```
LogisticRegression(random_state=123, solver='liblinear')
```

[57]:
```
prediction=logreg.predict(testx)
```

[58]:
```
print('Accuracy_score -',accuracy_score(testy,prediction))
print('Mean_squared_error -',mean_squared_error(testy,prediction))
```

```
Accuracy_score - 0.7337662337662337
Mean_squared_error - 0.2662337662337662
```

[59]:
```
print((confusion_matrix(testy,prediction)))
```

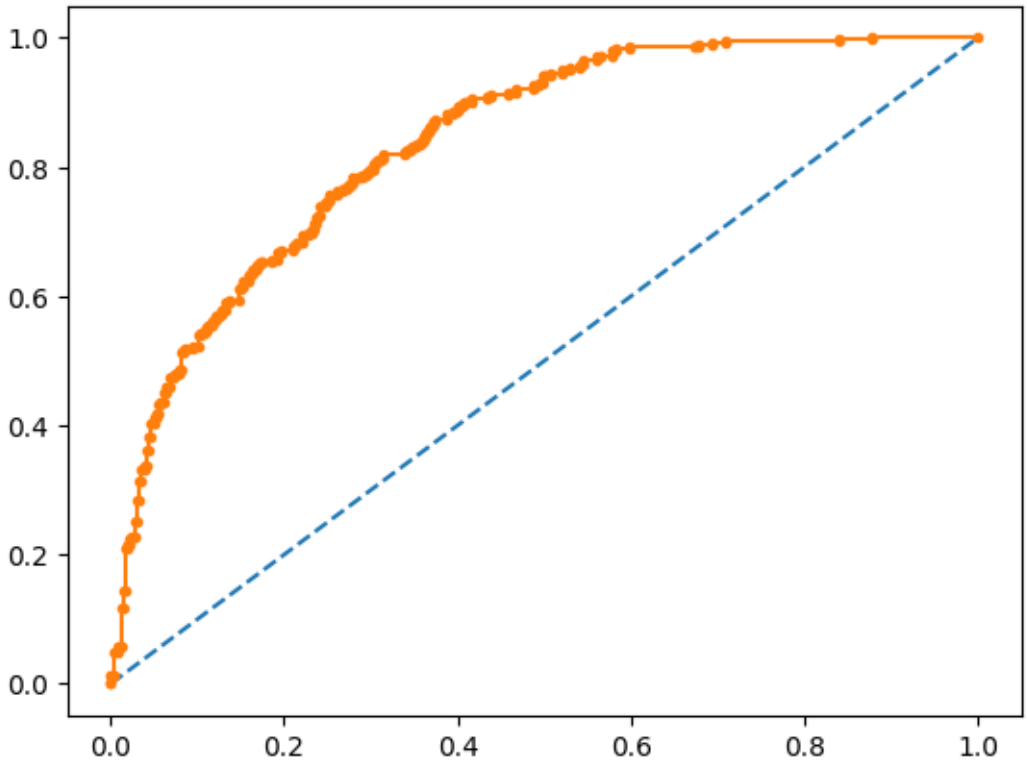
```
[[70 25]
 [16 43]]
```

[60]:
```
print(classification_report(testy,prediction))
```

```
              precision    recall  f1-score   support

           0       0.81      0.74      0.77        95
           1       0.63      0.73      0.68        59

    accuracy                           0.73       154
   macro avg       0.72      0.73      0.73       154
weighted avg       0.74      0.73      0.74       154
```

[61]:
```
#Preparing ROC Curve (Receiver Operating Characteristics Curve)
from sklearn.metrics import roc_curve
from sklearn.metrics import roc_auc_score

# predict probabilities
probs = logreg.predict_proba(trainx_res)
# keep probabilities for the positive outcome only
probs = probs[:, 1]
# calculate AUC
auc = roc_auc_score(trainy_res, probs)
print('AUC: %.3f' % auc)
# calculate roc curve
fpr, tpr, thresholds = roc_curve(trainy_res, probs)
# plot no skill
plt.plot([0, 1], [0, 1], linestyle='--')
# plot the roc curve for the model
plt.plot(fpr, tpr, marker='.')
plt.show()
```

```
AUC: 0.839
```



##Model-2 Random ForestClassifier

[62]:
```
rf=RandomForestClassifier(random_state=42,max_depth=5)
```

[63]:
```
rf.fit(trainx_res,trainy_res)
```

[63]:
```
RandomForestClassifier(max_depth=5, random_state=42)
```

[64]:
```
rf_predict=rf.predict(testx)
```

[65]:
```
print('Accuracy_score -',accuracy_score(testy,rf_predict))
print('Mean_squared_error -',mean_squared_error(testy,rf_predict))
```

```
Accuracy_score - 0.7402597402597403
Mean_squared_error - 0.2597402597402597
```

###Random ForestClassifier (Hyper Parameter Tunning)

[66]:
```
param_grid={'n_estimators':[100,400,200,300],'criterion':
  ↪['gini','entropy'],'max_depth':[1,2,3],'min_samples_split':
  ↪[2,4,3],'min_samples_leaf':[1,2,3],
'max_leaf_nodes':[1,2,3],'max_samples':[2,4,3]}
```

[67]:
```
grid=GridSearchCV( estimator=rf,param_grid=param_grid,n_jobs=-1,cv=5,verbose=2)
```

[68]:
```
rf_grid=RandomForestClassifier(criterion= 'gini',max_depth=␣
↪2,max_leaf_nodes=3,max_samples=4,min_samples_leaf= 1,min_samples_split=3,
n_estimators= 400,random_state=42)
```

[69]:
```
rf_grid.fit(trainx_res,trainy_res)
```

[69]:
```
RandomForestClassifier(max_depth=2, max_leaf_nodes=3, max_samples=4,
                       min_samples_split=3, n_estimators=400, random_state=42)
```

[70]:
```
rf_grid_predict=rf_grid.predict(testx)
```

[71]:
```
print('Accuracy_score -',accuracy_score(testy,rf_grid_predict))
print('Mean_squared_error -',mean_squared_error(testy,rf_grid_predict))
```

```
Accuracy_score - 0.7337662337662337
Mean_squared_error - 0.2662337662337662
```

[72]:
```
print((confusion_matrix(testy,prediction)))
```

```
[[70 25]
 [16 43]]
```

[73]:
```
print(classification_report(testy,prediction))
```

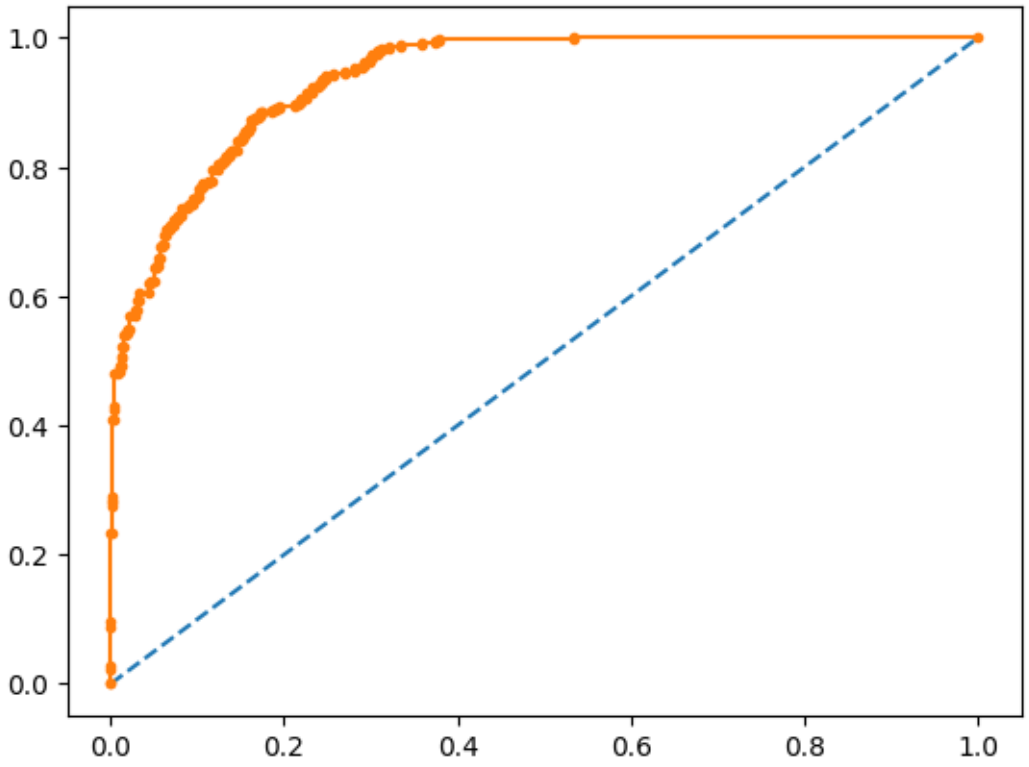
```
              precision    recall  f1-score   support

           0       0.81      0.74      0.77        95
           1       0.63      0.73      0.68        59

    accuracy                           0.73       154
   macro avg       0.72      0.73      0.73       154
weighted avg       0.74      0.73      0.74       154
```

[74]:
```
#Preparing ROC Curve (Receiver Operating Characteristics Curve)
from sklearn.metrics import roc_curve
from sklearn.metrics import roc_auc_score
```

[75]:
```
# predict probabilities
probs = rf.predict_proba(trainx_res)
# keep probabilities for the positive outcome only
probs = probs[:, 1]
# calculate AUC
auc = roc_auc_score(trainy_res, probs)
print('AUC: %.3f' % auc)
```

```
# calculate roc curve
fpr, tpr, thresholds = roc_curve(trainy_res, probs)
# plot no skill
plt.plot([0, 1], [0, 1], linestyle='--')
# plot the roc curve for the model
plt.plot(fpr, tpr, marker='.')
plt.show()
```

```
AUC: 0.938
```



##Model-3 Decision TreeClassifier

[76]:
```
dc=DecisionTreeClassifier(random_state=42)
```

[77]:
```
dc.fit(trainx_res,trainy_res)
```

[77]:
```
DecisionTreeClassifier(random_state=42)
```

[78]:
```
dc_pred=dc.predict(testx)
```

[79]:
```
print('Accuracy_score -',accuracy_score(testy,dc_pred))
print('Mean_squared_error -',mean_squared_error(testy,dc_pred))
```

```
Accuracy_score - 0.6623376623376623
Mean_squared_error - 0.33766233766233766
```

##Decision TreeClassifier (Hyper Parameter Tunning)

[80]:
```
dc_param_grid={'splitter':["best", "random"],'criterion':
 ↪['gini','entropy'],'max_depth':[1,2,3],
'min_samples_split':[1,2,3],'min_samples_leaf':[1,2,3],'max_leaf_nodes':[1,2,3]}
```

[81]:
```
import warnings
warnings.filterwarnings('ignore')
dc_grid=GridSearchCV(estimator=dc,param_grid=dc_param_grid,n_jobs=-1,cv=5,verbose=2)
dc_grid.fit(trainx_res,trainy_res)
```

```
Fitting 5 folds for each of 324 candidates, totalling 1620 fits
```

[81]:
```
GridSearchCV(cv=5, estimator=DecisionTreeClassifier(random_state=42), n_jobs=-1,
             param_grid={'criterion': ['gini', 'entropy'],
                         'max_depth': [1, 2, 3], 'max_leaf_nodes': [1, 2, 3],
                         'min_samples_leaf': [1, 2, 3],
                         'min_samples_split': [1, 2, 3],
                         'splitter': ['best', 'random']},
             verbose=2)
```

[82]:
```
dc_grid.best_params_
```

[82]:
```
{'criterion': 'gini',
 'max_depth': 1,
 'max_leaf_nodes': 2,
 'min_samples_leaf': 1,
 'min_samples_split': 2,
 'splitter': 'best'}
```

[83]:
```
dc_final=DecisionTreeClassifier(criterion= 'gini',␣
 ↪max_depth=2,max_leaf_nodes=4,min_samples_leaf= 1,
min_samples_split= 2,splitter='best',random_state=42)
```

[84]:
```
dc_final.fit(trainx_res,trainy_res)
dc_final_pred=dc_final.predict(testx)
```

[85]:
```
print('Accuracy_score -',accuracy_score(testy,dc_final_pred))
print('Mean_squared_error -',mean_squared_error(testy,dc_final_pred))
```

```
Accuracy_score - 0.6883116883116883
Mean_squared_error - 0.3116883116883117
```

[86]:
```
print((confusion_matrix(testy,dc_final_pred)))
```

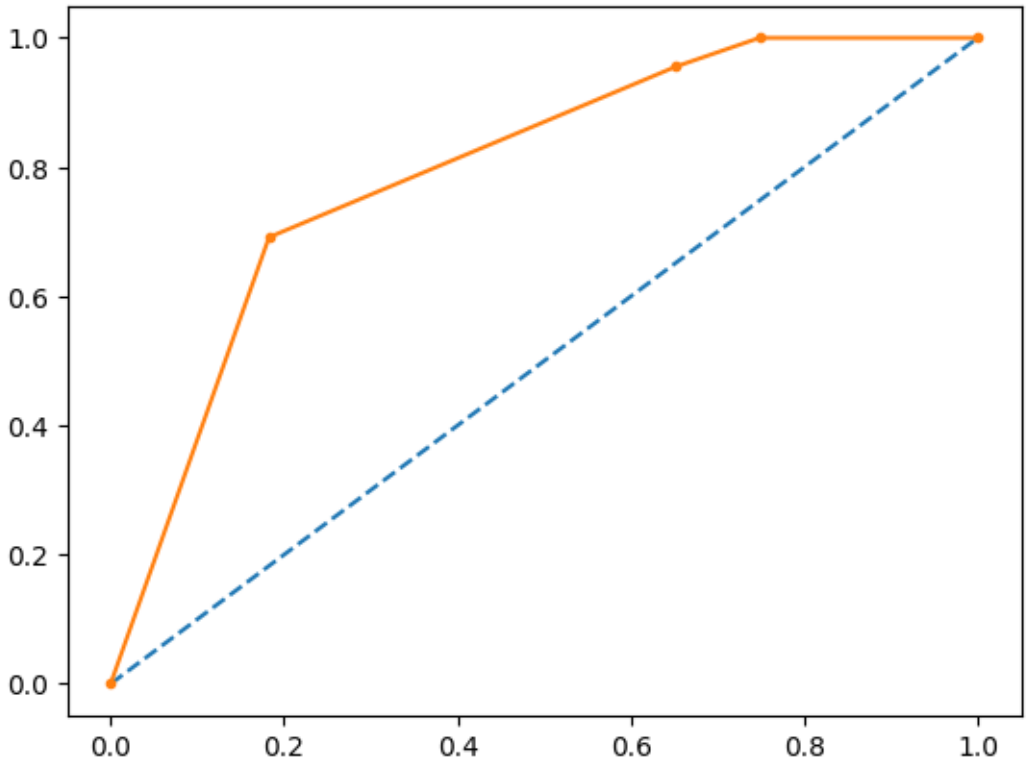
```
[[73 22]
 [26 33]]
```

[87]:
```
print((classification_report(testy,dc_final_pred)))
```

```
              precision    recall  f1-score   support

           0       0.74      0.77      0.75        95
           1       0.60      0.56      0.58        59

    accuracy                           0.69       154
   macro avg       0.67      0.66      0.67       154
weighted avg       0.68      0.69      0.69       154
```

[88]:
```
# Preparing ROC Curve (Receiver operating Characteristics Curve)
from sklearn.metrics import roc_curve
from sklearn.metrics import roc_auc_score

# predict probabilities
probs = dc_final.predict_proba(trainx_res)
# keep probabilities for the positive outcome only
probs = probs[:, 1]
# calculate AUC
auc = roc_auc_score(trainy_res, probs)
print('AUC: %.3f' % auc)
# calculate roc curve
fpr, tpr, thresholds = roc_curve(trainy_res, probs)
# plot no skill
plt.plot([0, 1], [0, 1], linestyle='--')
# plot the roc curve for the model
plt.plot(fpr, tpr, marker='.')
plt.show()
```

```
AUC: 0.795
```



0.12 Model-4 KNN

[89]:
```
from sklearn.neighbors import KNeighborsClassifier
```

[90]:
```
knn=KNeighborsClassifier(n_neighbors=4)
```

[91]:
```
knn.fit(trainx_res,trainy_res)
```

[91]:
```
KNeighborsClassifier(n_neighbors=4)
```

[92]:
```
knn_pred=knn.predict(testx)
```

[93]:
```
print('Accuracy_score -',accuracy_score(testy,knn_pred))
print('Mean_squared_error -',mean_squared_error(testy,knn_pred))
```

```
Accuracy_score - 0.6233766233766234
Mean_squared_error - 0.37662337662337664
```

[94]:
```
print((confusion_matrix(testy,knn_pred)))
```

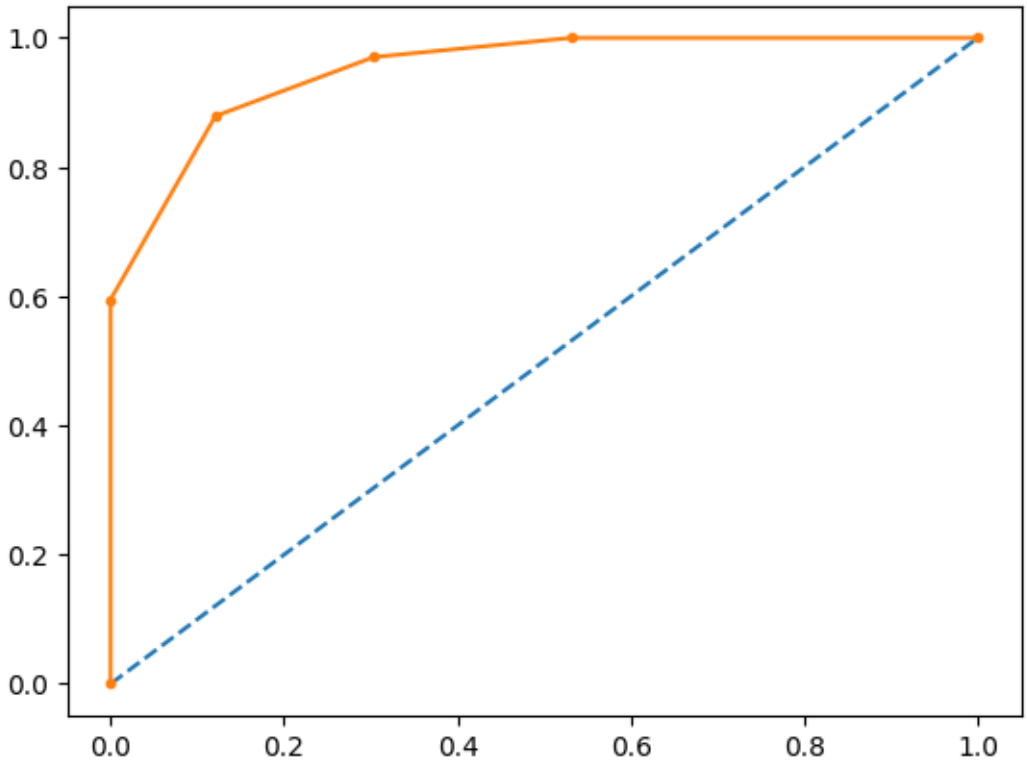
```
[[68 27]
 [31 28]]
```

[95]:
```
print((classification_report(testy,knn_pred)))
```

```
              precision    recall  f1-score   support

           0       0.69      0.72      0.70        95
           1       0.51      0.47      0.49        59

    accuracy                           0.62       154
   macro avg       0.60      0.60      0.60       154
weighted avg       0.62      0.62      0.62       154
```

[96]:
```
#Preparing ROC Curve (Receiver Operating Characteristics Curve)
from sklearn.metrics import roc_curve
from sklearn.metrics import roc_auc_score

# predict probabilities
probs = knn.predict_proba(trainx_res)
# keep probabilities for the positive outcome only
probs = probs[:, 1]
# calculate AUC
auc = roc_auc_score(trainy_res, probs)
print('AUC: %.3f' % auc)
# calculate roc curve
fpr, tpr, thresholds = roc_curve(trainy_res, probs)
# plot no skill
plt.plot([0, 1], [0, 1], linestyle='--')
# plot the roc curve for the model
plt.plot(fpr, tpr, marker='.')
plt.show()
```

```
AUC: 0.951
```



##Model Accuracy Comparision

[97]:
```
Algorithms=['KNN','RandomForest','Decisiontree','logreg']
Accuracy_Score=[accuracy_score(testy,knn_pred),accuracy_score(testy,rf_grid_predict),accuracy_
# Create a DataFrame
accuracy_df = pd.DataFrame({'Algorithm': Algorithms, 'Accuracy':␣
 ↪Accuracy_Score})

# Display the accuracy table
print(accuracy_df)
```

```
      Algorithm  Accuracy
0           KNN  0.623377
1  RandomForest  0.733766
2  Decisiontree  0.688312
3        logreg  0.733766
```

##Inferences from Model Accuracy Comparison **RandomForest Performs Well:**

Among the algorithms tested, RandomForest exhibits the highest accuracy at 73.38%.

Logistic Regression performs well: Among the algorithms tested, Logistic Regression exhibits the accuracy same as KNN 73.38%.

KNN Shows Lower Accuracy:

KNN has the lowest accuracy among the models, with a score of 62.34%. Consistent Performances:

Decision Tree Accuracy 68.83%.

Consideration for Model Selection:

The choice of the algorithm depends on various factors, including the specific requirements of the task, interpretability, and computational efficiency.

Further Evaluation:

Additional evaluation metrics, such as precision, recall, and F1 score, should be considered for a comprehensive assessment of model performance.

###Comparision of various models with the results from KNN algorithm

[98]:
```
#creating the objects
logreg_cv = LogisticRegression(solver='liblinear',random_state=123)
dt_cv=DecisionTreeClassifier(random_state=123)
knn_cv=KNeighborsClassifier()
rf_cv=RandomForestClassifier(random_state=123)
cv_dict = {0: 'Logistic Regression', 1: 'Decision Tree',2:'KNN',3:'Random␣
 ↪Forest'}
cv_models=[logreg_cv,dt_cv,knn_cv,rf_cv]

for i,model in enumerate(cv_models):
    print("{} Test Accuracy: {}".format(cv_dict[i],cross_val_score(model,␣
 ↪trainx, trainy, cv=10, scoring ='accuracy').mean()))
```

```
Logistic Regression Test Accuracy: 0.7768376520359598
Decision Tree Test Accuracy: 0.7053146483342146
KNN Test Accuracy: 0.7229772607086197
Random Forest Test Accuracy: 0.762083553675304
```

###Inferences from Model Comparison with KNN Algorithm Results Logistic Regression Outperforms: Among the models tested, Logistic Regression exhibits the highest test accuracy at 77.68%. Decisive Model Differences:

Decision Tree, and Random Forest show lower test accuracies compared to Logistic Regression, ranging from 70.53% to 76.21%. Consideration for Model Selection:

Logistic Regression and SVC might be preferred choices based on higher test accuracies, but other factors such as interpretability and computational efficiency should be considered. Cross-Validation Insights:

The use of cross-validation provides a robust estimate of model performance, reducing the impact of data partitioning on results. Further Exploration:

Evaluation metrics beyond accuracy, such as precision, recall, and F1 score, should be considered for a comprehensive understanding of model effectiveness.