PROBLEM STATEMENT

Prediction of abnormal blood pressure.

OBJECTIVE

Employing statistical techniques, conduct a preliminary prognosis of Hypertenstion / Hypotension, based on the all given factors.

LOADING LIBRARIES

```
import time
start_time = time.time()


from sklearn.metrics import accuracy_score, recall_score, precision_score, f1_score
from sklearn.metrics import roc_curve, roc_auc_score, classification_report, confusion_matrix
from sklearn.utils import class_weight

from sklearn.ensemble import ExtraTreesRegressor
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler

from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn import model_selection
from xgboost import XGBClassifier
from sklearn.svm import SVC

import pandas as pd
import numpy as np
import warnings

from pandas.plotting import scatter_matrix
import matplotlib.pyplot as plt
import seaborn as sns
```

DATA LOADING/DATA GATHERING

```
## Training Data
df = pd.read_csv(r"/content/BPA_DATA.csv")
df.head()
```

	Patient_Number	Blood_Pressure_Abnormality	Level_of_Hemoglobin	Genetic_Pedigree_Coefficient	Age	BMI	Sex	Pr
0	1	1	11.28	0.90	34	23	1	

```
## Getting basic statistical detail of columns
df.describe()
```

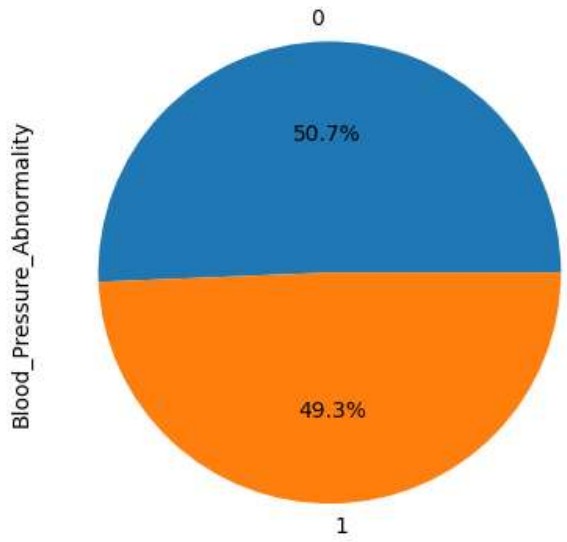
	Patient_Number	Blood_Pressure_Abnormality	Level_of_Hemoglobin	Genetic_Pedigree_Coefficient	Age
count	2000.000000	2000.000000	2000.000000	1908.000000	2000.000000
mean	1000.500000	0.493500	11.710035	0.494817	46.558500
std	577.494589	0.500083	2.186701	0.291736	17.107832
min	1.000000	0.000000	8.100000	0.000000	18.000000
25%	500.750000	0.000000	10.147500	0.240000	32.000000
50%	1000.500000	0.000000	11.330000	0.490000	46.000000
75%	1500.250000	1.000000	12.945000	0.740000	62.000000
max	2000.000000	1.000000	17.560000	1.000000	75.000000

```
## checking for the classes distribution in target column
df['Blood_Pressure_Abnormality'].value_counts()
```

```
0    1013
1     987
Name: Blood_Pressure_Abnormality, dtype: int64
```

```
df['Blood_Pressure_Abnormality'].value_counts().plot(kind='pie',autopct = '%1.1f%%')
#no imbalance in data 0 = 50.6%, 1 = 49.4%
```

```
<Axes: ylabel='Blood_Pressure_Abnormality'>
```




```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2000 entries, 0 to 1999
Data columns (total 15 columns):
 #   Column                        Non-Null Count  Dtype
---  ------                        --------------  -----
 0   Patient_Number                2000 non-null   int64
```

```
 1   Blood_Pressure_Abnormality     2000 non-null   int64
 2   Level_of_Hemoglobin            2000 non-null   float64
 3   Genetic_Pedigree_Coefficient   1908 non-null   float64
 4   Age                            2000 non-null   int64
 5   BMI                            2000 non-null   int64
 6   Sex                            2000 non-null   int64
 7   Pregnancy                      442 non-null    float64
 8   Smoking                        2000 non-null   int64
 9   Physical_activity              2000 non-null   int64
 10  salt_content_in_the_diet       2000 non-null   int64
 11  alcohol_consumption_per_day    1758 non-null   float64
 12  Level_of_Stress                2000 non-null   int64
 13  Chronic_kidney_disease         2000 non-null   int64
 14  Adrenal_and_thyroid_disorders  2000 non-null   int64
dtypes: float64(4), int64(11)
memory usage: 234.5 KB
```

CHEKING NULL PERCENTAGE

```
## Function for checking percentage of null values in dataframe columns

def null_value_check_in_dataframe(df):
    percent_missing = df.isnull().sum() * 100 / len(df)
    missing_value_df = pd.DataFrame({'column_name': df.columns,
                                     'percent_missing': percent_missing})
    return missing_value_df

df_null = null_value_check_in_dataframe(df)
df_null
```

	column_name	percent_missing
Patient_Number	Patient_Number	0.0
Blood_Pressure_Abnormality	Blood_Pressure_Abnormality	0.0
Level_of_Hemoglobin	Level_of_Hemoglobin	0.0
Genetic_Pedigree_Coefficient	Genetic_Pedigree_Coefficient	4.6
Age	Age	0.0
BMI	BMI	0.0
Sex	Sex	0.0
Pregnancy	Pregnancy	77.9
Smoking	Smoking	0.0
Physical_activity	Physical_activity	0.0
salt_content_in_the_diet	salt_content_in_the_diet	0.0
alcohol_consumption_per_day	alcohol_consumption_per_day	12.1
Level_of_Stress	Level_of_Stress	0.0
Chronic_kidney_disease	Chronic_kidney_disease	0.0
Adrenal_and_thyroid_disorders	Adrenal_and_thyroid_disorders	0.0

SPLITTING DATA INTO TRAIN AND TEST DATASET

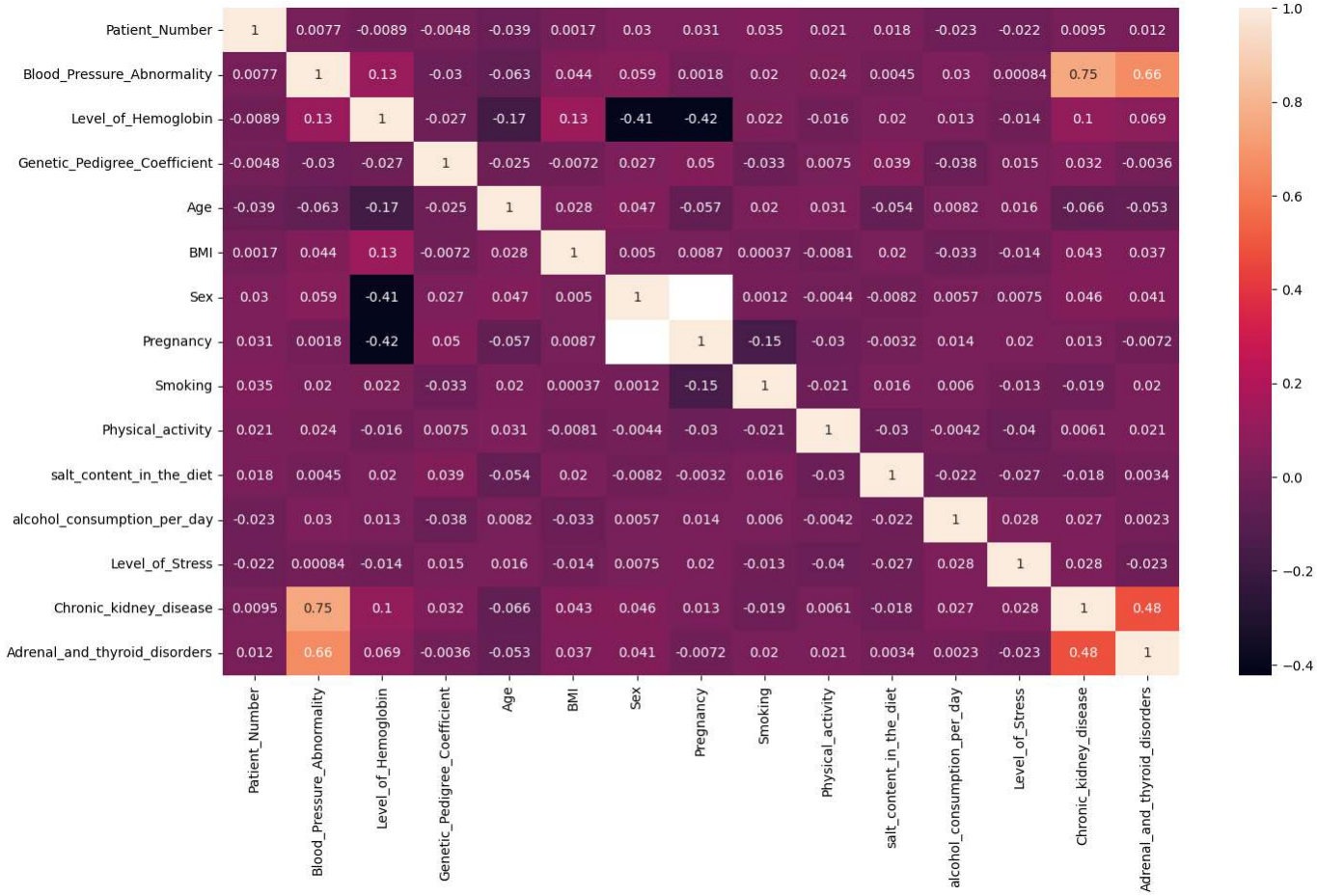
```
## Train-Test data split (10% Test data)

train,test = train_test_split(df, train_size=0.9 ,test_size = 0.1, random_state=50, stratify=df['Blood_Pressure_Abnorm
```

EXPLORATORY DATA ANALYSIS

1. PEARSON CORRELATION

```
# calculate the correlation matrix

plt.figure(figsize=(16,9))
sns.heatmap(train.corr(), annot = True)
```

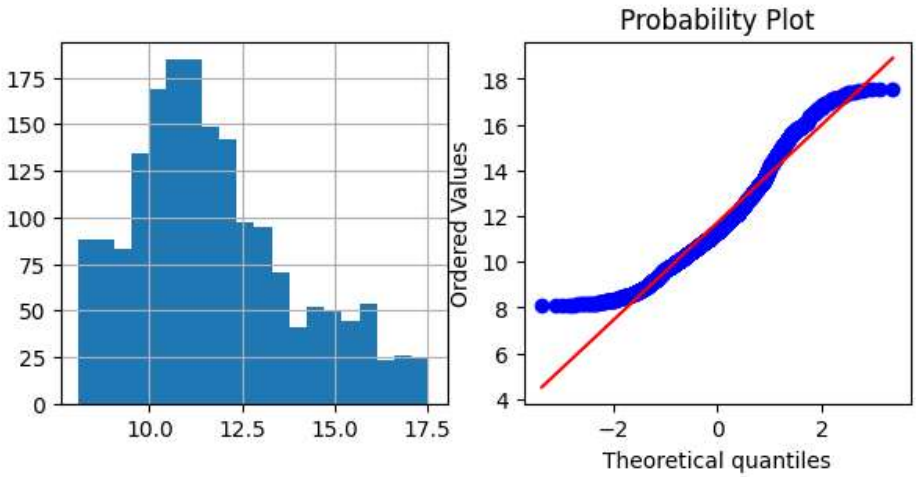
```
<Axes: >
```



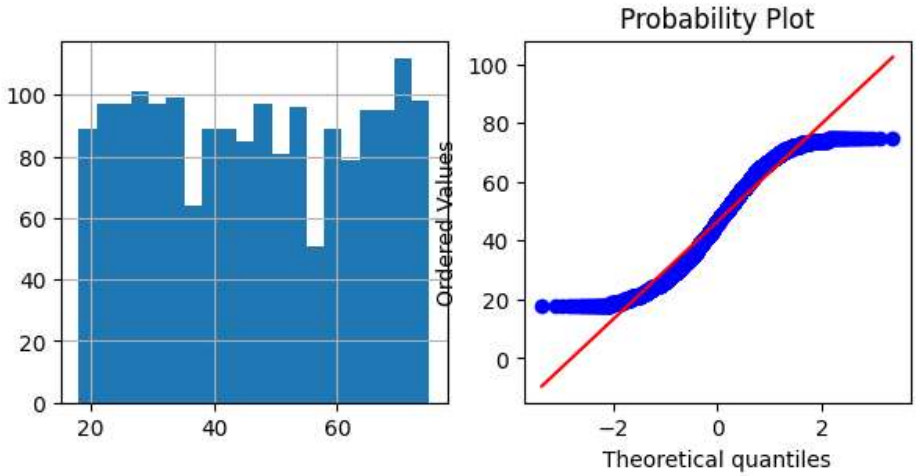
Unsupported Cell Type. Double-Click to inspect/edit the content.

2. HISTOGRAM & QQ PLOT

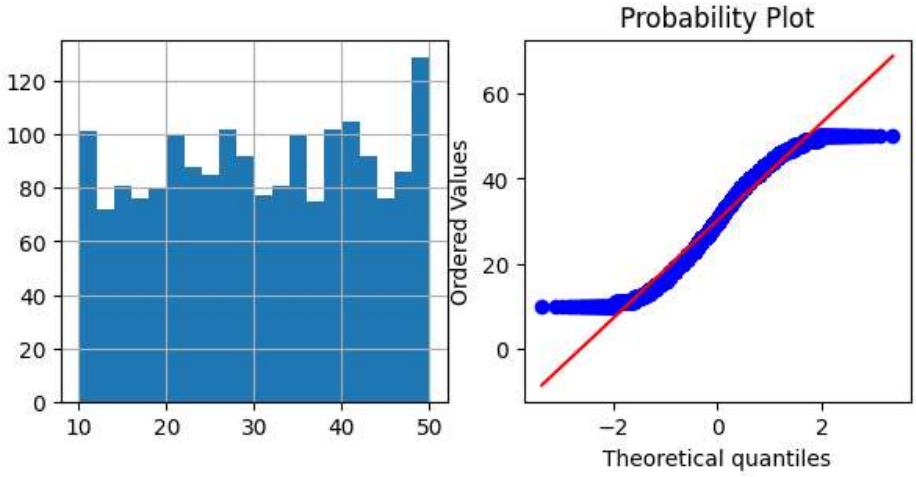
```
## Plotting histogram & QQ PLOT to check the values distribution in columns

import scipy.stats as stat
import pylab

def plot_data(train,features):
  plt.figure(figsize=(7,3))
  plt.subplot(1,2,1) # 1 row, 2 columns, and this plot is the first plot.
  train[features].hist(bins=20)
  plt.subplot(1,2,2) #1 row, 2 columns, and this plot is the second plot.
  stat.probplot(train[features],dist='norm',plot=pylab)
  plt.show()
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
from scipy.stats import norm
from scipy.stats import shapiro
def normal_Distibution_test(feature):
    my_data = norm.rvs(size=500)
    normal_test = shapiro(feature)
    return normal_test
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
print ("Distrtibution of level_of _hemoglobin", plot_data(train,'Level_of_Hemoglobin'))
print ("Test_result ofLevel_of_Hemoglobin :",normal_Distibution_test(train['Level_of_Hemoglobin']))
print ("Distrtibution of Age",plot_data(train,'Age'))
print ("Test_result ofLevel_of_Hemoglobin :",normal_Distibution_test(train['Age']))
print ("Distrtibution of BMI", plot_data(train,'BMI'))
print ("Test_result ofLevel_of_Hemoglobin :",normal_Distibution_test(train['BMI']))
print ("Distrtibution of Physical_activity",plot_data(train,'Physical_activity'))
print ("Test_result ofLevel_of_Hemoglobin :",normal_Distibution_test(train['Physical_activity']))
print ("Distrtibution of salt_content_in_the_diet", plot_data(train,'salt_content_in_the_diet'))
print ("Test_result ofLevel_of_Hemoglobin :",normal_Distibution_test(train['salt_content_in_the_diet']))

'''Interpretation

    If the P-Value of the Shapiro Test is larger than 0.05, we assume a normal distribution
    If the P-Value of the Shapiro Test is smaller than 0.05, we do not assume a normal distribution'''
```



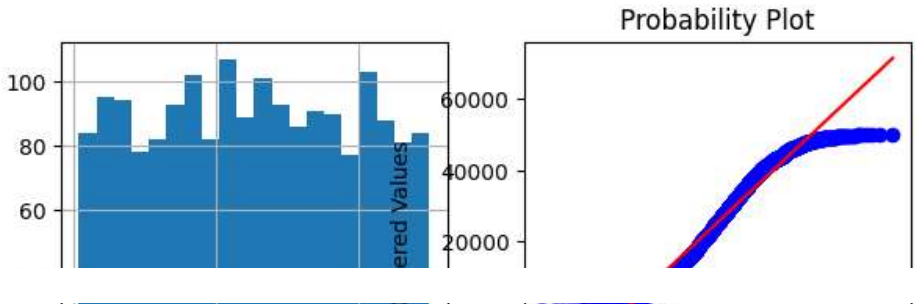

```
Distrtibution of level_of _hemoglobin None
Test_result ofLevel_of_Hemoglobin : ShapiroResult(statistic=0.9568712711334229, pvalue=1.152835016002745e-22)
```




```
Distrtibution of Age None
Test_result ofLevel_of_Hemoglobin : ShapiroResult(statistic=0.9491034150123596, pvalue=1.709260224867875e-24)
```




```
Distrtibution of BMI None
Test_result ofLevel_of_Hemoglobin : ShapiroResult(statistic=0.9562168121337891, pvalue=7.917267422263711e-23)
```



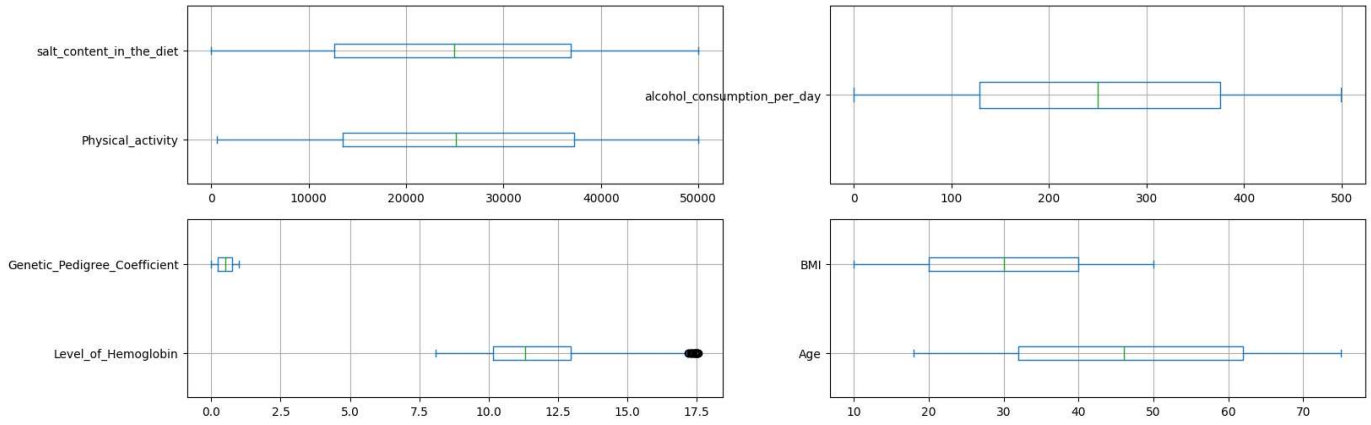
Note : For the normal distribution in alcohol_consumption_per_day and Genetic_Pedigree_Coefficient, we will see normal distribution after handling of missing values init.

4. BOXPLOTS (For checking Range of Values and Outliers)

```
Test result ofLevel of Hemoglobin : ShapiroResult(statistic=0.9583740234375, pvalue=2.7771445533699696e-22)
```

```
## Boxplot for checking the variance and outliers in data

fig, axes = plt.subplots(2, 2, figsize = (18,6))

boxplot_cols1 = ['Physical_activity','salt_content_in_the_diet']
boxplot_cols2 = ['alcohol_consumption_per_day']
boxplot_cols3 = ['Age','BMI']
boxplot_cols4 = ['Level_of_Hemoglobin','Genetic_Pedigree_Coefficient']

train[boxplot_cols2].plot.box(vert = False, grid = True, ax=axes[0][1])
train[boxplot_cols1].plot.box(vert = False, grid = True, ax=axes[0][0])
train[boxplot_cols4].plot.box(vert = False, grid = True, ax=axes[1][0])
train[boxplot_cols3].plot.box(vert = False, grid = True, ax=axes[1][1])
```

```
<Axes: >
```



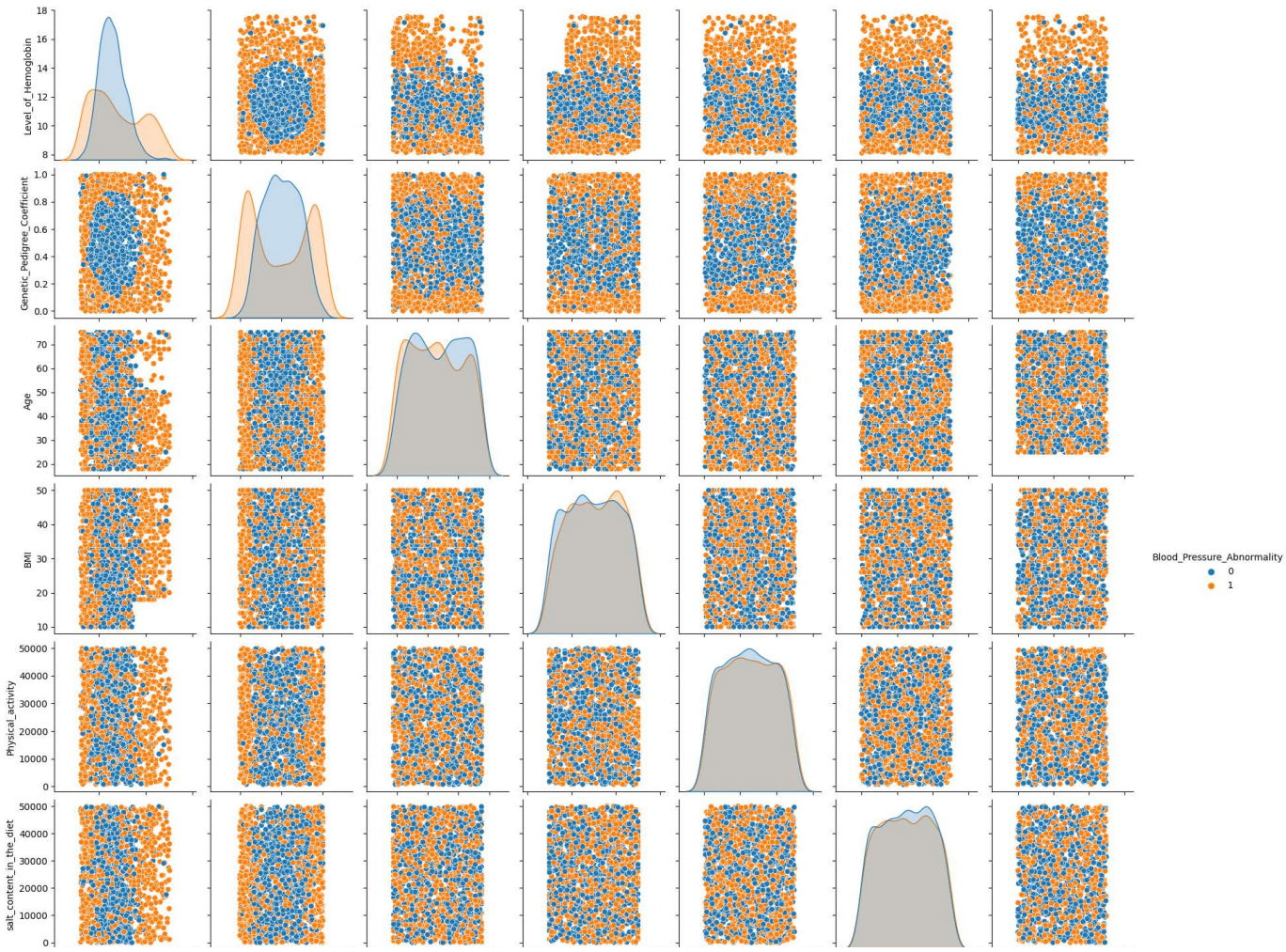

```
# train["Level_of_Hemoglobin"].min()
train["Level_of_Hemoglobin"].max()
```

```
17.56
```

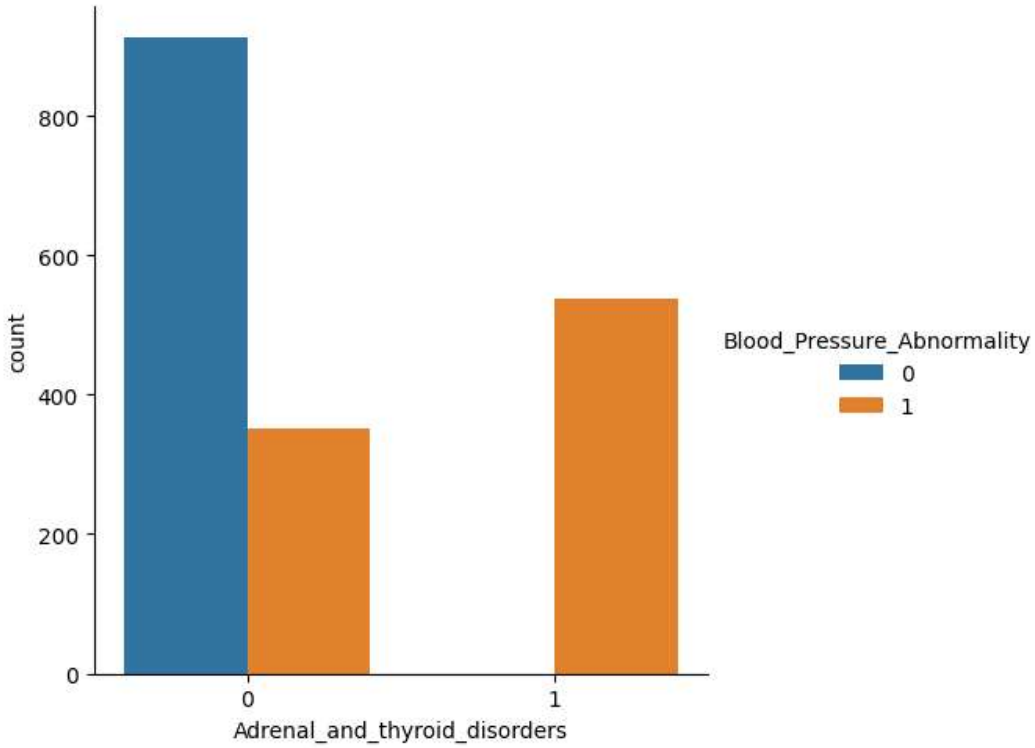
5. PAIR PLOT (Scatter Matrix Plot)

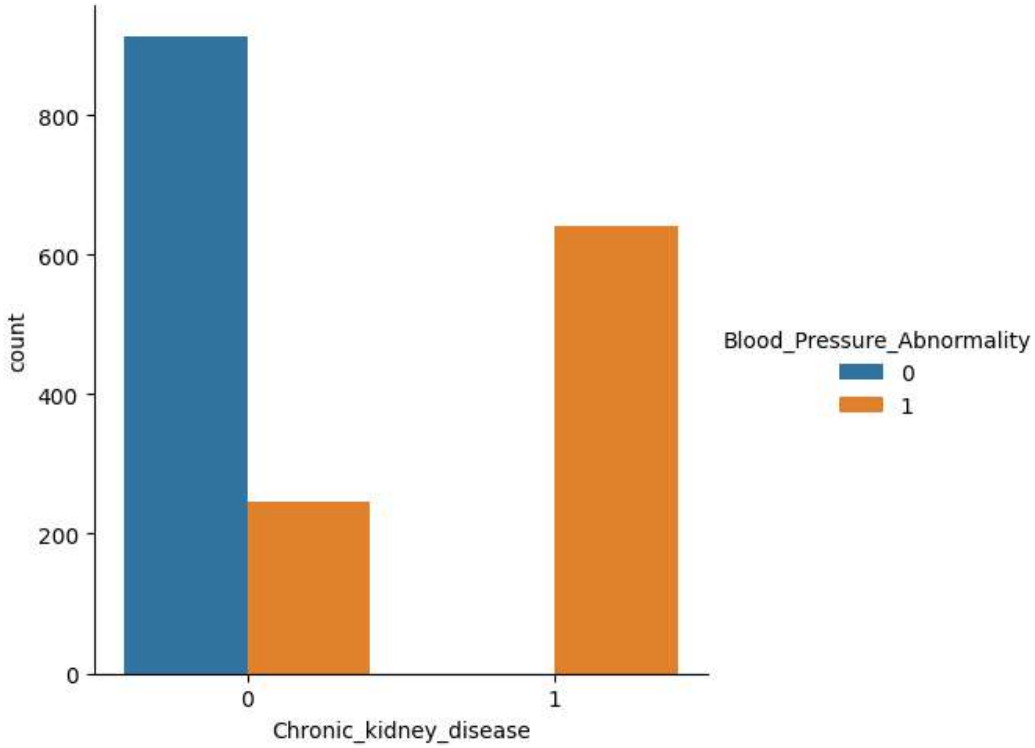
```
## Checking for the relationship between numerical variables through scatter plot
df_pairlot =  train[['Level_of_Hemoglobin','Genetic_Pedigree_Coefficient', 'Age', 'BMI', 'Physical_activity',
                     'salt_content_in_the_diet','alcohol_consumption_per_day',"Blood_Pressure_Abnormality" ]]
sns.pairplot(df_pairlot, hue = 'Blood_Pressure_Abnormality')
# plt.figure.autofmt_xdate()
```

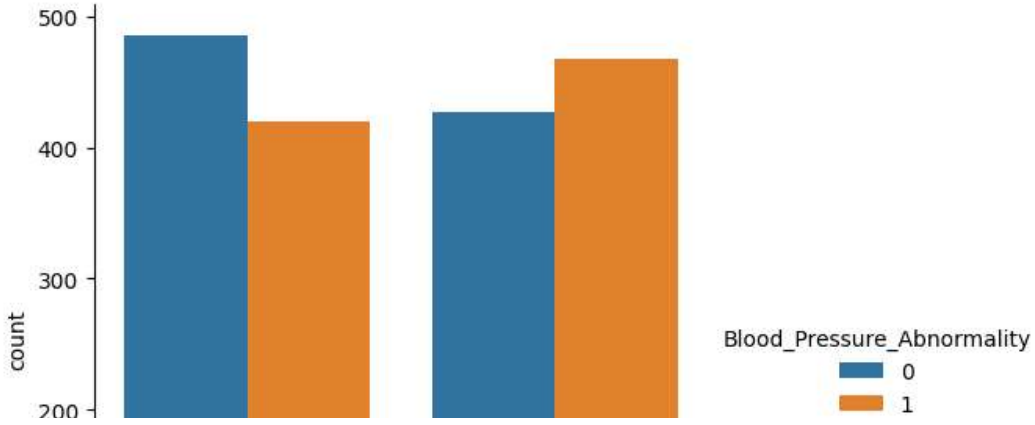
```
<seaborn.axisgrid.PairGrid at 0x786470056ec0>
```

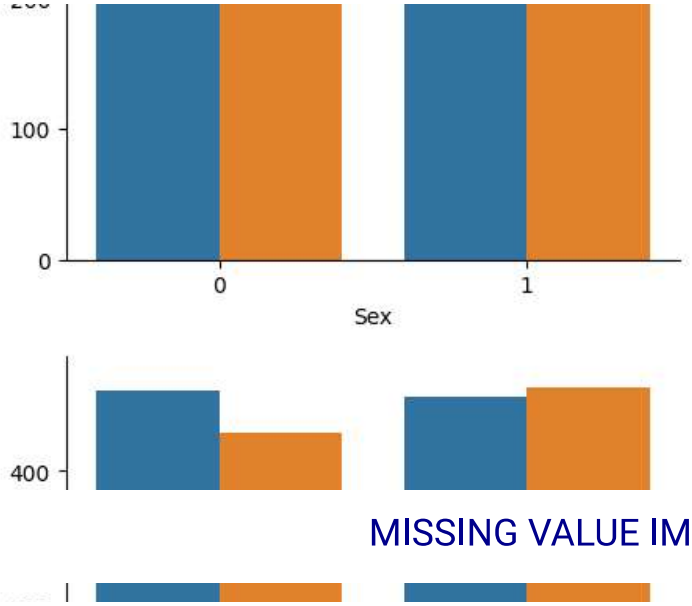



```
## Barplots for various categorical variable to check the effect on Target variable (or is there any association)

sns.catplot(x="Adrenal_and_thyroid_disorders", kind="count", hue="Blood_Pressure_Abnormality", data=train)
sns.catplot(x="Chronic_kidney_disease", kind="count", hue="Blood_Pressure_Abnormality", data=train)
sns.catplot(x="Sex", kind="count", hue="Blood_Pressure_Abnormality", data=train)
sns.catplot(x="Smoking", kind="count", hue="Blood_Pressure_Abnormality", data=train)
plt.show()
```


800
600
400
200
0
count
0
1
Adrenal_and_thyroid_disorders
Blood_Pressure_Abnormality
0
1


800
600
400
200
0
count
0
1
Chronic_kidney_disease
Blood_Pressure_Abnormality
0
1


500
400
300
200
count
Blood_Pressure_Abnormality
0
1



MISSING VALUE IMPUTATION

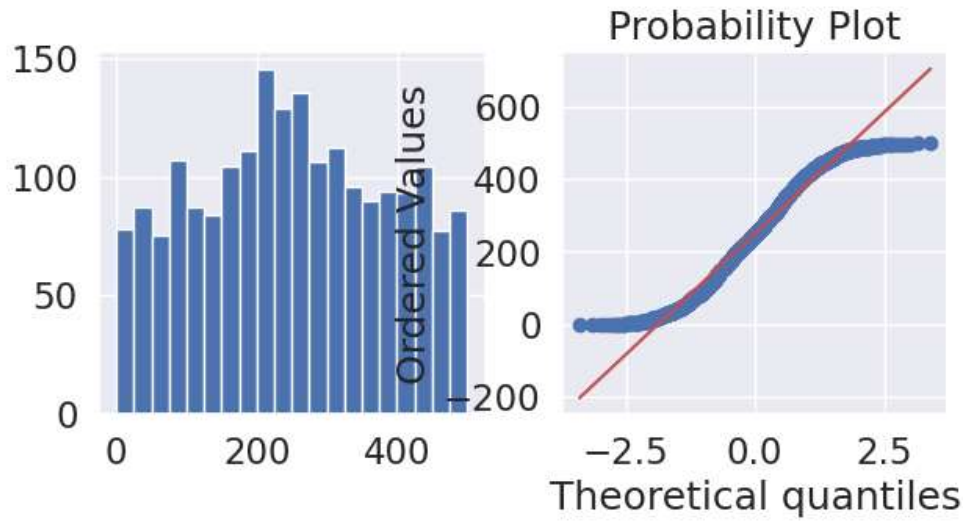
```
## Fuction for Missing value imputation using IterativeImputer

def missing_value_imputation(df):

    ## Creating copy of a dataframe
    df_imputed = df.copy()

    ## Imputing 'Pregnancy' column values, putting Pregnancy=0 where Sex=0
    #train['Pregnancy'] = np.where(train['Sex'] == 1, train['Pregnancy'], 0)
    mask = df_imputed['Sex'] == 0
    df_imputed.loc[mask, 'Pregnancy'] = 0

    ## Remaining 'NaN' value is replaced with -1 (left Nan if for Sex=1, so created a separated category for those)
    df_imputed['Pregnancy'].fillna(-1, inplace=True)

    ## Changing the column type to 'int', it was 'float' earlier
    df_imputed['Pregnancy'] = df_imputed['Pregnancy'].astype(int)

    #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    ## Imputing missing values using Iterative Imputer for columns 'Genetic_Pedigree_Coefficient' and 'alcohol_consumpti

    # Define modeling pipeline
    model = ExtraTreesRegressor(n_estimators=20, random_state=0)
    imputer = IterativeImputer(estimator=model)

    # Fitting the model
    imputer.fit(df_imputed)

    imputed_values = pd.DataFrame(imputer.transform(df_imputed), columns=df_imputed.columns)

    imputed_values['Patient_Number'] = imputed_values['Patient_Number'].astype(int)
    imputed_values_subset = imputed_values[['Patient_Number','Genetic_Pedigree_Coefficient', 'alcohol_consumption_per_da


    df_imputed.drop(['Genetic_Pedigree_Coefficient', 'alcohol_consumption_per_day'], axis = 1, inplace=True)
    df_imputed = pd.merge(df_imputed,imputed_values_subset, on=['Patient_Number'], how='left')

    return df_imputed
normality_test = missing_value_imputation(df)
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/impute/_iterative.py:785: ConvergenceWarning: [IterativeImputer] E
  warnings.warn(
```

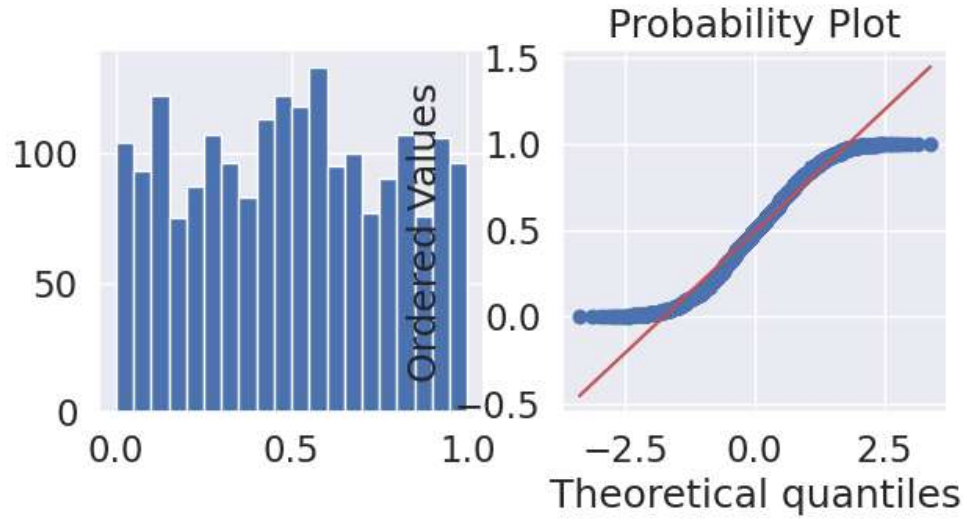
```
normality_test.to_csv('/content/cleaned_data1.csv')
```

Unsupported Cell Type. Double-Click to inspect/edit the content.

```
print ("Distrtibution of alcohol_consumption_per_day", plot_data(normality_test,'alcohol_consumption_per_day'))
print ("Test_result of alcohol_consumption_per_day :",normal_Distibution_test(normality_test['alcohol_consumption_per_da
print ("Distrtibution of Genetic_Pedigree_Coefficient",plot_data(normality_test,'Genetic_Pedigree_Coefficient'))
print ("Test_result of Genetic_Pedigree_Coefficient :",normal_Distibution_test(normality_test['Genetic_Pedigree_Coeffici
```




```
Distrtibution of alcohol_consumption_per_day None
Test_result of alcohol_consumption_per_day : ShapiroResult(statistic=0.9689673781394958, pvalue=2.3247206719812466
```




```
Distrtibution of Genetic_Pedigree_Coefficient None
Test_result of Genetic_Pedigree_Coefficient : ShapiroResult(statistic=0.9618586301803589, pvalue=1.559498366224949
```

SCALING

```
## Function for normalizing data using MinMaxScaler

def data_normalization(df):

    # Normalization - MinMaxScaler Transform
    cols_to_transform = ['Level_of_Hemoglobin','Age','BMI','Physical_activity','salt_content_in_the_diet','alcohol_con
    df_to_transform = df[cols_to_transform]

    trans = MinMaxScaler()
    scaled_features = trans.fit_transform(df_to_transform)

    # convert the array back to a dataframe
    df_transformed = pd.DataFrame(scaled_features, index=df_to_transform.index, columns=df_to_transform.columns)

    df_subset = df.drop(cols_to_transform, axis=1)
    df_train_transformed = pd.concat([df_transformed,df_subset], axis=1)

    return df_train_transformed
```

ENCODING

```
## Function for performing the one-hot encoding of categorical variables

def oneHotEncoding(df):

    ## Creating one-hot encoding of categorical columns
    df = pd.get_dummies(df, columns=['Pregnancy','Level_of_Stress'],drop_first = True)
    5
    ## Dropping column 'Patient_Number' as it is unique id of patient, so not useful for analysis
    df.drop(['Patient_Number'], axis=1, inplace=True)

    return df


## Function for data preparation (which does data cleaning, normalization and column encoding)

def data_preparation(df):

    df_missing_value_imputation = missing_value_imputation(df)
    df_data_normalized = data_normalization(df_missing_value_imputation)
    df_onehot_encoding = oneHotEncoding(df_data_normalized)

    return df_onehot_encoding
```

TRAIN AND TEST DATA PREPERATION

```
## Data Preparation for Training dataframe
df_train_cleaned = data_preparation(train)

## Data Preparation for Test dataframe
df_test_cleaned = data_preparation(test)
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/impute/_iterative.py:785: ConvergenceWarning: [IterativeImputer] E
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/impute/_iterative.py:785: ConvergenceWarning: [IterativeImputer] E
  warnings.warn(
```

SEPERATING OUT TARGET AND DEPENDENT VARIABLE

```
df_train_cleaned.shape
```

```
(1800, 16)
```

```
df_train_cleaned.to_csv('cleaned_data.csv')
```

```
# Putting feature variable to X_train and X_test
X_train = df_train_cleaned.drop(['Blood_Pressure_Abnormality'], axis=1)
X_test = df_test_cleaned.drop(['Blood_Pressure_Abnormality'], axis=1)

# Putting response variable to y_train and y_test
y_train = df_train_cleaned['Blood_Pressure_Abnormality']
y_test = df_test_cleaned['Blood_Pressure_Abnormality']
```

BULDING RANDOM FOREST MODEL

```
# Creating a Random Forest Classifier
clf=RandomForestClassifier(n_estimators=100)

# Training the model using the training sets y_pred=clf.predict(X_test)
clf.fit(X_train,y_train)

# predicting probabilities
rf_probs = clf.predict_proba(X_test)

# keeping probabilities for the positive outcome only
rf_probs = rf_probs[:, 1]

# predicting labels
y_pred=clf.predict(X_test)
```

EVALUATION METRIX

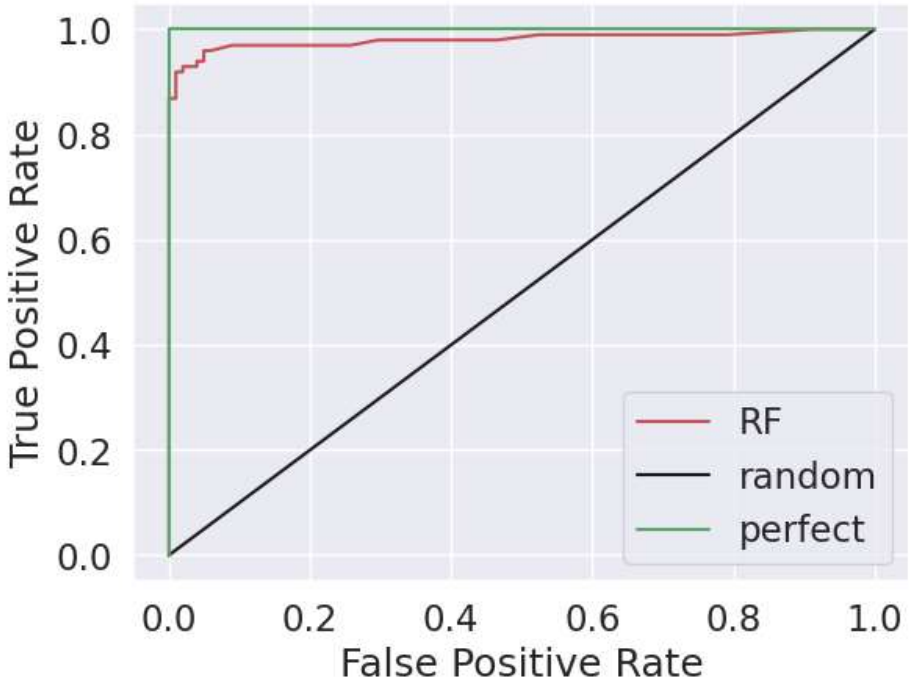
```
# Calculating different metrics to check the model performance on Test data
```

```
## Calculating different model metrics
print("Accuracy  :: %.3f"% accuracy_score(y_test, y_pred))
print("Recall    :: %.3f"% recall_score(y_test, y_pred))
print("Precision :: %.3f"% precision_score(y_test, y_pred))
print("F1 Score  :: %.3f"% f1_score(y_test, y_pred))
print("AUC Score :: %.3f"% roc_auc_score(y_test, rf_probs))

print("\nClassification Report ::")
print("\n",classification_report(y_test, y_pred))
```

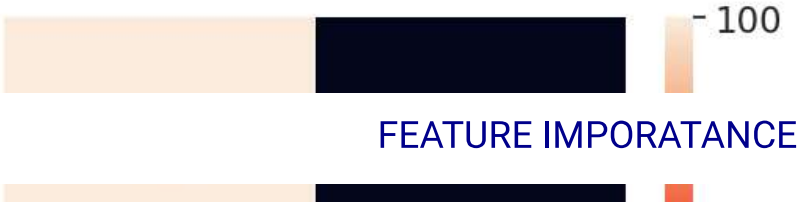
```
Accuracy  :: 0.950
Recall    :: 0.909
Precision :: 0.989
F1 Score  :: 0.947
AUC Score :: 0.981

Classification Report ::

              precision    recall  f1-score   support

           0       0.92      0.99      0.95       101
           1       0.99      0.91      0.95        99

    accuracy                           0.95       200
   macro avg       0.95      0.95      0.95       200
```

```
weighted avg       0.95      0.95      0.95       200
```

```
## Plotting ROC curve
fpr_RF, tpr_RF, thresholds_RF = roc_curve(y_test, rf_probs)

plt.plot(fpr_RF, tpr_RF,'r-',label = 'RF')
plt.plot([0,1],[0,1],'k-',label='random')
plt.plot([0,0,1,1],[0,1,1,1],'g-',label='perfect')
plt.legend()
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.show()
```

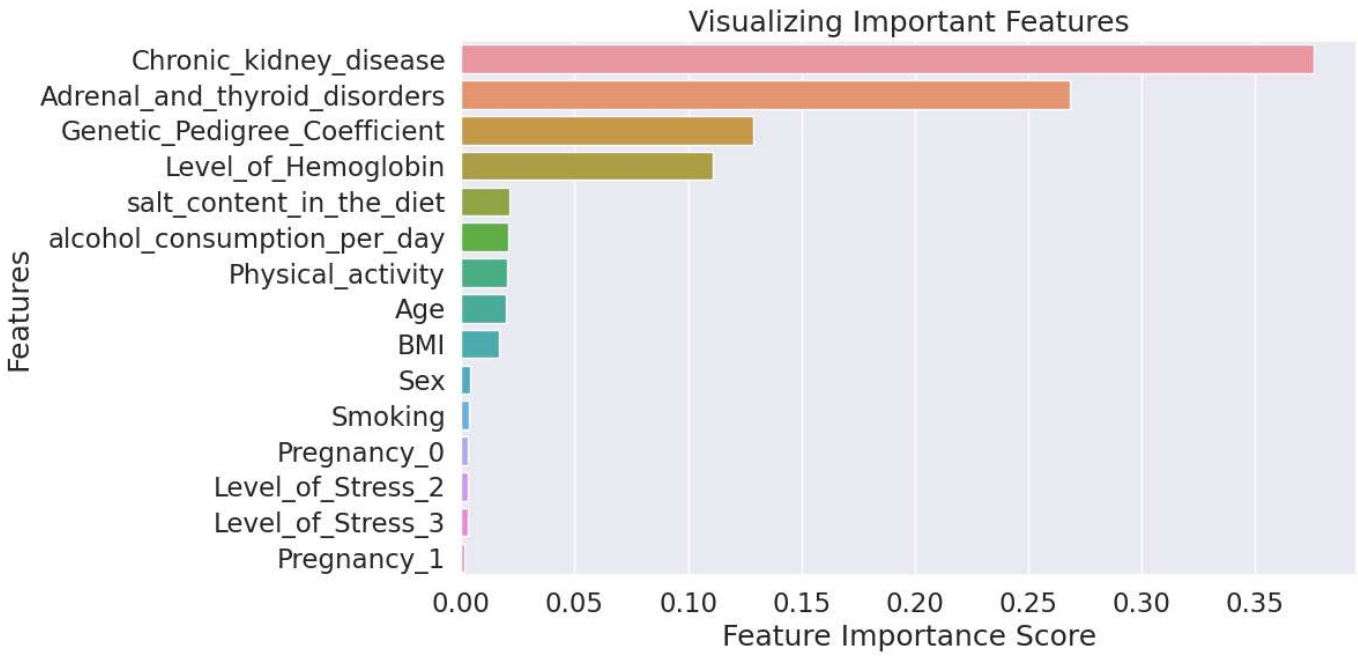



```
## Plotting confusion matrix

cf_matrix = confusion_matrix(y_test, y_pred)
sns.heatmap(cf_matrix, annot=True, fmt="d")
plt.show()
```



FEATURE IMPORATANCE

```
## Calculating Feature Importance
feature_imp = pd.Series(clf.feature_importances_,index=X_train.columns).sort_values(ascending=False)

# Creating a bar plot for Feature Importance
fig = plt.figure(figsize=(10,6))
sns.barplot(x=feature_imp, y=feature_imp.index)

# Add labels to your graph
plt.xlabel('Feature Importance Score')
plt.ylabel('Features')
plt.title("Visualizing Important Features")
plt.show()
```



Based on the Feature Importance we can select the subset of features, lets say Top 10 features or the features which account for 95% of the importance. The same number of features must be used in the training and testing sets.

Then We have build the model gain on Train dataset and prediction on Test dataset Testing out multiple models

TESTING OUT MULTIPLE MODEL

The Idea here is to try out multiple classification models and choose the best one based on the evaluation metrics and Training/Testing time.

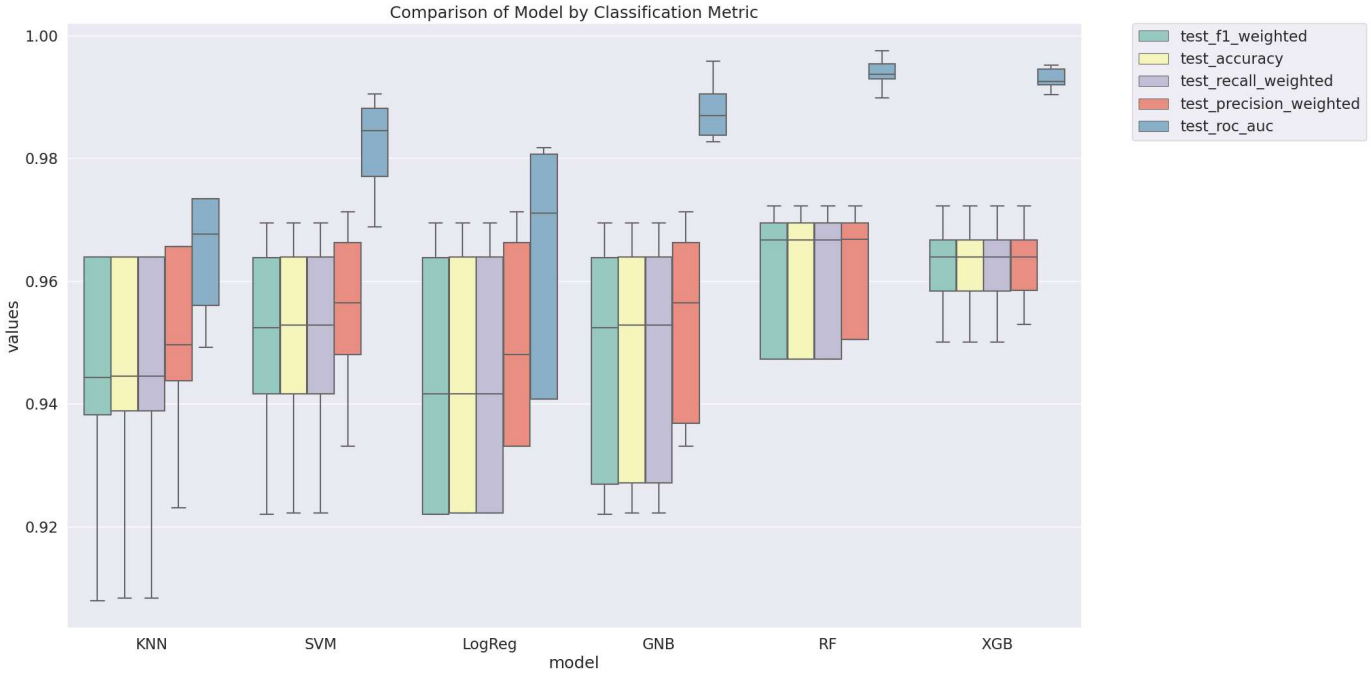
```
## Trying out different classification models and then we'll choose the best based on the different metrics saved

## Function for running different experiments or different models and caturing their respective metrices
def run_exps(X_train: pd.DataFrame , y_train: pd.DataFrame, X_test: pd.DataFrame, y_test: pd.DataFram
    '''
    Lightweight script to test many models and find winners
    :param X_train: training split
    :param y_train: training target vector
    :param X_test: test split
    :param y_test: test target vector
    :return: DataFrame of predictions
    '''

    # variable to hold all of the datasets that will be created from the application of k-fold cross validation on the
    dfs = []

    # list of tuples holding the name and class for each classifier to be tested
    models = [('LogReg', LogisticRegression()),
              ('RF', RandomForestClassifier()),
              ('KNN', KNeighborsClassifier()),
              ('SVM', SVC()),
              ('GNB', GaussianNB()),
              ('XGB', XGBClassifier(eval_metric='logloss'))
             ]

    results = []
    names = []
    scoring = ['accuracy', 'precision_weighted', 'recall_weighted', 'f1_weighted', 'roc_auc']

    for name, model in models:

      kfold = model_selection.KFold(n_splits=5, shuffle=True, random_state=90210)
      cv_results = model_selection.cross_validate(model, X_train, y_train, cv=kfold, scoring=scoring)

      clf = model.fit(X_train, y_train)
      y_pred = clf.predict(X_test)

      print(name)
      print(classification_report(y_test, y_pred))

      results.append(cv_results)
      names.append(name)

      this_df = pd.DataFrame(cv_results)
      this_df['model'] = name
      dfs.append(this_df)
      final = pd.concat(dfs, ignore_index=True)

    return final


## Calling model experimentation function
final = run_exps(X_train, y_train, X_test, y_test)
```

```
              0       0.88      1.00      0.94       101
              1       1.00      0.86      0.92        99

       accuracy                           0.93       200
      macro avg       0.94      0.93      0.93       200
   weighted avg       0.94      0.93      0.93       200

   RF
                precision    recall  f1-score   support
```

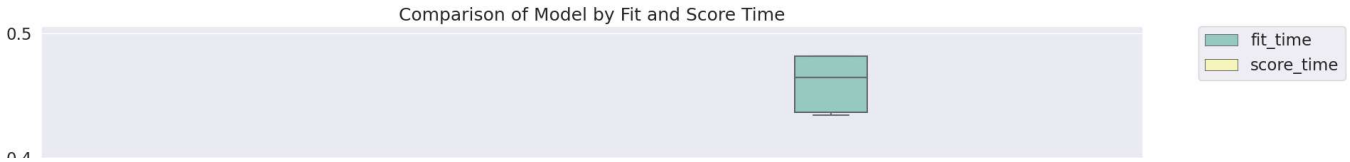
```
          accuracy                           0.94       200
         macro avg       0.95      0.94      0.94       200
      weighted avg       0.95      0.94      0.94       200

      KNN
                    precision    recall  f1-score   support

                0       0.87      1.00      0.93       101
                1       1.00      0.85      0.92        99

          accuracy                           0.93       200
         macro avg       0.94      0.92      0.92       200
      weighted avg       0.93      0.93      0.92       200

      SVM
                    precision    recall  f1-score   support

                0       0.88      1.00      0.94       101
                1       1.00      0.86      0.92        99

          accuracy                           0.93       200
         macro avg       0.94      0.93      0.93       200
      weighted avg       0.94      0.93      0.93       200

      GNB
                    precision    recall  f1-score   support

                0       0.88      1.00      0.94       101
                1       1.00      0.86      0.92        99

          accuracy                           0.93       200
         macro avg       0.94      0.93      0.93       200
      weighted avg       0.94      0.93      0.93       200

      XGB
                    precision    recall  f1-score   support

                0       0.90      0.99      0.94       101
                1       0.99      0.89      0.94        99

          accuracy                           0.94       200
         macro avg       0.94      0.94      0.94       200
      weighted avg       0.94      0.94      0.94       200
```

```
## To obtain better estimates of the distribution of metrics from each model, ran empirical bootstrapping at 30 samples
## Additionally, partitioned the data into two sorts: performance metrics and fit-time metrics.

bootstraps = []
for model in list(set(final.model.values)):
    model_df = final.loc[final.model == model]
    bootstrap = model_df.sample(n=30, replace=True)
    bootstraps.append(bootstrap)

bootstrap_df = pd.concat(bootstraps, ignore_index=True)
results_long = pd.melt(bootstrap_df,id_vars=['model'],var_name='metrics', value_name='values')

# fit time metrics
time_metrics = ['fit_time','score_time']

## PERFORMANCE METRICS
results_long_nofit = results_long.loc[~results_long['metrics'].isin(time_metrics)] # get df without fit data
results_long_nofit = results_long_nofit.sort_values(by='values')

## TIME METRICS
results_long_fit = results_long.loc[results_long['metrics'].isin(time_metrics)] # df with fit data
results_long_fit = results_long_fit.sort_values(by='values')
```

```
## Plotting performance metrics from the 5-fold cross validation.

plt.figure(figsize=(20, 12))
sns.set(font_scale=1.5)
g = sns.boxplot(x="model", y="values", hue="metrics", data=results_long_nofit, palette="Set3")
plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
plt.title('Comparison of Model by Classification Metric')
plt.show()
```




```
## Training and Scoring time comparison by plotting

plt.figure(figsize=(20, 12))
sns.set(font_scale=1.5)
g = sns.boxplot(x="model", y="values", hue="metrics", data=results_long_fit, palette="Set3")
plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
plt.title('Comparison of Model by Fit and Score Time')
plt.show()
```




```
## Evaluation metrics details for all the trained models

metrics = list(set(results_long_nofit.metrics.values))
bootstrap_df.groupby(['model'])[metrics].agg([np.std, np.mean])
```

	test_precision_weighted		test_accuracy		test_recall_weighted		test_f1_weighted		test_roc_auc	
	std	mean	std	mean	std	mean	std	mean	std	mean
model										
GNB	0.015174	0.954374	0.018873	0.949074	0.018873	0.949074	0.018934	0.948937	0.003942	0.987421
KNN	0.015516	0.948693	0.020133	0.942870	0.020133	0.942870	0.020296	0.942627	0.009692	0.964334
LogReg	0.014670	0.950731	0.018367	0.944630	0.018367	0.944630	0.018416	0.944468	0.017165	0.963389
RF	0.008531	0.962780	0.009994	0.961759	0.009994	0.961759	0.009996	0.961757	0.002354	0.993924
SVM	0.012725	0.957656	0.015703	0.953333	0.015703	0.953333	0.015755	0.953177	0.007355	0.983406
XGB	0.006831	0.962645	0.007862	0.961944	0.007862	0.961944	0.007857	0.961942	0.001671	0.992940

```
## Training and prediction time calculation for all the trained models

time_metrics = list(set(results_long_fit.metrics.values))
bootstrap_df.groupby(['model'])[time_metrics].agg([np.std, np.mean])
```

	fit_time		score_time	
	std	mean	std	mean
model				
GNB	0.000141	0.003109	0.000190	0.009914
KNN	0.001484	0.008528	0.004849	0.077588
LogReg	0.006850	0.032349	0.011314	0.024647
RF	0.018798	0.458927	0.002817	0.045094
SVM	0.002441	0.063923	0.002272	0.046374
XGB	0.002699	0.076625	0.001064	0.017527

Based on the Evaluation Metrics (Accuracy, Precision Recall, F1-score etc.) Random Forest(RF) isary (or oth by far the best model. But it is having slightly higher model training time (fit_time) compared to other models (except XGBoost, which is even having higer training time).

Either we can choose the Random Forest (as a best model) or go for second best model (in terms of prediction metrics and training time which is GNB(Gaussian Naive Bayes) else XGB is also a good option in case of accuracy and less training time.

So, this is basically an trade-of between the Accuracy(or other metrics for that matter) and the Training time, so we have to decide which one to choose if we have comparable model(slightly high/low prediction metrics and high/low mdel building/training time) because there is cost involve in that (Time + resources) and every business is having different context to look into this.

Once the model is finalized, then we'll go for the Hyperparameter tuning in oder to improve the model Accuracy further (We'll have to decide based on the business objective whethere we really need this or not).

HYPERPARAMETER TUNING

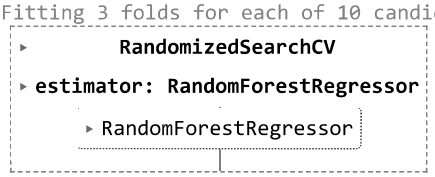
1. RANDOM FOREST

```
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV# Number of trees in random forest
n_estimators = [int(x) for x in np.linspace(start = 100, stop = 2000, num = 10)]
# Number of features to consider at every split
max_features = ['auto', 'sqrt']
# Maximum number of levels in tree
max_depth = [int(x) for x in np.linspace(10, 110, num = 11)]
max_depth.append(None)
# Minimum number of samples required to split a node
min_samples_split = [2, 5, 10]
# Minimum number of samples required at each leaf node
min_samples_leaf = [1, 2, 4]
# Method of selecting samples for training each tree
bootstrap = [True, False]# Create the random grid
grid_grid = {'n_estimators': n_estimators,
               'max_features': max_features,
               'max_depth': max_depth,
               'min_samples_split': min_samples_split,
               'min_samples_leaf': min_samples_leaf,
               'bootstrap': bootstrap}
print(grid_grid)
```

```
{'n_estimators': [100, 311, 522, 733, 944, 1155, 1366, 1577, 1788, 2000], 'max_features': ['auto', 'sqrt'], 'max_d
```

```
from sklearn.ensemble import RandomForestRegressor
# Use the random grid to search for best hyperparameters
# First create the base model to tune
rf = RandomForestRegressor()
# Random search of parameters, using 3 fold cross validation,
# search across 100 different combinations, and use all available cores
rf_grid = RandomizedSearchCV(estimator = rf, param_distributions = grid_grid, cv = 3, verbose=2, n_jobs = -1)
# Fit the random search model
rf_grid.fit(X_train,y_train)
```

```
Fitting 3 folds for each of 10 candidates, totalling 30 fits
```




```
rf_grid.best_params_
```

```
{'n_estimators': 1577,
 'min_samples_split': 2,
 'min_samples_leaf': 1,
 'max_features': 'sqrt',
 'max_depth': 90,
 'bootstrap': False}
```