CPE019 - Prelim Examination

Names: QUEJADO, Jimlord M. | SERRANO, Jio A.
Course and Section: CPE019-CPE32S3
Date of Submission: March 6, 2024
Instructor: Engr. Roman Richard

About the data

Breast Cancer Wisconsin (Diagnostic)

Dataset Characteristics
Multivariate

Subject Area
Health and Medicine

Associated Tasks
Classification

Feature Type
Real

Number of Instances
569

Number of Features
30

DOI
10.24432/C5DW2B

Variable Information

Ten real-valued features are computed for each cell nucleus:

```
a) radius (mean of distances from center to points on the perimeter)
b) texture (standard deviation of gray-scale values)
c) perimeter
d) area
e) smoothness (local variation in radius lengths)
f) compactness (perimeter^2 / area - 1.0)
g) concavity (severity of concave portions of the contour)
h) concave points (number of concave portions of the contour)
i) symmetry
j) fractal dimension ("coastline approximation" - 1)
```

Variable	Description
1. ID	Unique identifier for each patient
2. Diagnosis	Diagnosis (M = malignant, B = benign)
3. radius1	
4. texture1	
5. perimeter1	
6. area1	
7. smoothness1	
8. compactness1	
9. concavity1	
10. concave_points1	
11. symmetry1	
12. fractal_dimension1	
13. radius2	
14. texture2	
15. perimeter2	
16. area2	
17. smoothness2	
18. compactness2	
19. concavity2	
20. concave_points2	
21. symmetry2	
22. fractal_dimension2	
23. radius3	
24. texture3	
25. perimeter3	
26. area3	
27. smoothness3	
28. compactness3	
29. concavity3	
30. concave_points3	
31. symmetry3	
32. fractal_dimension3	

Pre-Processing of Data

In order to prepare the data, we first checked the data set using pandas and performed the necessary data processing techniques such as checking for null entries, changing variables to suitable data for processing and adding column headers.

Loading the Data File

In [1]:

```
# Connecting to google drive to access the csv file
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

In [2]:

```
# Loading the CSV file
import pandas as pd
import numpy as np

data_names_path = "/content/drive/MyDrive/CPE019 - Prelims/Data.csv"   #creating string for path
namesDF = pd.read_csv(data_names_path)                                 #loadng the csv file
```

In [3]:

```
# Verifying if the csv was successfully loaded
namesDF.head()
```

Out[3]:

	842302	M	17.99	10.38	122.8	1001	0.1184	0.2776	0.3001	0.1471	...	25.38	17.33	184.6	2019	0.1622	0.6656	0.711
0	842517	M	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869	0.07017	...	24.99	23.41	158.80	1956.0	0.1238	0.1866	0.241
1	84300903	M	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974	0.12790	...	23.57	25.53	152.50	1709.0	0.1444	0.4245	0.450
2	84348301	M	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414	0.10520	...	14.91	26.50	98.87	567.7	0.2098	0.8663	0.686
3	84358402	M	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980	0.10430	...	22.54	16.67	152.20	1575.0	0.1374	0.2050	0.400
4	843786	M	12.45	15.70	82.57	477.1	0.12780	0.17000	0.1578	0.08089	...	15.47	23.75	103.40	741.6	0.1791	0.5249	0.535

5 rows × 32 columns

Adding Column Headers

In [4]:

```
# Adding the column headers based on the data descritpion

column_names = [                                                        #Column names
    'ID', 'Diagnosis',
    'radius1','texture1','perimeter1','area1','smoothness1','compactness1', 'concavity1','concave_points1','symmetry1', 'fractal_dimension1',
    'radius2','texture2','perimeter2','area2','smoothness2','compactness2','concavity2','concave_points2','symmetry2','fractal_dimension2',
    'radius3','texture3','perimeter3','area3','smoothness3','compactness3','concavity3','concave_points3','symmetry3','fractal_dimension3'
    ]

namesDF.to_csv("UpdatedData.csv", header=column_names, index=False)           #creating new csv file
```

In [5]:

```
#Updating the namesDF with the new file
import pandas as pd
import numpy as np

data_names_path = "/content/UpdatedData.csv"
namesDF = pd.read_csv(data_names_path)
```

In [6]:

```
# Verifying if the csv was successfully loaded
namesDF.head()
```

Out[6]:

	ID	Diagnosis	radius1	texture1	perimeter1	area1	smoothness1	compactness1	concavity1	concave_points1	...	radius3	te
0	842517	M	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869	0.07017	...	24.99	
1	84300903	M	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974	0.12790	...	23.57	
2	84348301	M	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414	0.10520	...	14.91	
3	84358402	M	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980	0.10430	...	22.54	
4	843786	M	12.45	15.70	82.57	477.1	0.12780	0.17000	0.1578	0.08089	...	15.47	

5 rows × 32 columns

Checking for Null Values

In [7]:

```
# Checking if there are NaN values in the table
namesDF.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 568 entries, 0 to 567
Data columns (total 32 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   ID                  568 non-null    int64
 1   Diagnosis           568 non-null    object
 2   radius1             568 non-null    float64
 3   texture1            568 non-null    float64
 4   perimeter1          568 non-null    float64
 5   area1               568 non-null    float64
 6   smoothness1         568 non-null    float64
 7   compactness1        568 non-null    float64
 8   concavity1          568 non-null    float64
 9   concave_points1     568 non-null    float64
 10  symmetry1           568 non-null    float64
 11  fractal_dimension1  568 non-null    float64
 12  radius2             568 non-null    float64
 13  texture2            568 non-null    float64
 14  perimeter2          568 non-null    float64
 15  area2               568 non-null    float64
 16  smoothness2         568 non-null    float64
 17  compactness2        568 non-null    float64
 18  concavity2          568 non-null    float64
 19  concave_points2     568 non-null    float64
 20  symmetry2           568 non-null    float64
 21  fractal_dimension2  568 non-null    float64
 22  radius3             568 non-null    float64
 23  texture3            568 non-null    float64
 24  perimeter3          568 non-null    float64
 25  area3               568 non-null    float64
 26  smoothness3         568 non-null    float64
 27  compactness3        568 non-null    float64
 28  concavity3          568 non-null    float64
 29  concave_points3     568 non-null    float64
 30  symmetry3           568 non-null    float64
 31  fractal_dimension3  568 non-null    float64
dtypes: float64(30), int64(1), object(1)
memory usage: 142.1+ KB
```

In [8]:

```
namesDF.iloc[15:20]
```

Out[8]:

	ID	Diagnosis	radius1	texture1	perimeter1	area1	smoothness1	compactness1	concavity1	concave_points1	...	radius3	t
15	848406	M	14.68	20.13	94.74	684.5	0.09867	0.07200	0.07395	0.05259	...	19.07	
16	84862001	M	16.13	20.68	108.10	798.8	0.11700	0.20220	0.17220	0.10280	...	20.96	
17	849014	M	19.81	22.15	130.00	1260.0	0.09831	0.10270	0.14790	0.09498	...	27.32	
18	8510426	B	13.54	14.36	87.46	566.3	0.09779	0.08129	0.06664	0.04781	...	15.11	
19	8510653	B	13.08	15.71	85.63	520.0	0.10750	0.12700	0.04568	0.03110	...	14.50	

5 rows × 32 columns

Converting the Diagnosis M and B to 0 and 1

In this section, in order to process the categorical values, we changed the (M)alignant and (B)enign to 0 and 1, respectively. Such conversion will allow us to perform our model training later on more seamlessly.

In [9]:

```
#Changing Diagnosis Column:  0 = (M)malignant, 1 = (B)benign

namesDF["Diagnosis"] = namesDF["Diagnosis"].apply(lambda toLabel: 0 if toLabel=='M' else 1)
```

In [10]:

```
#verifying if the values are changed
namesDF.iloc[15:20]
```

Out[10]:

	ID	Diagnosis	radius1	texture1	perimeter1	area1	smoothness1	compactness1	concavity1	concave_points1	...	radius3
15	848406	0	14.68	20.13	94.74	684.5	0.09867	0.07200	0.07395	0.05259	...	19.07
16	84862001	0	16.13	20.68	108.10	798.8	0.11700	0.20220	0.17220	0.10280	...	20.96
17	849014	0	19.81	22.15	130.00	1260.0	0.09831	0.10270	0.14790	0.09498	...	27.32
18	8510426	1	13.54	14.36	87.46	566.3	0.09779	0.08129	0.06664	0.04781	...	15.11
19	8510653	1	13.08	15.71	85.63	520.0	0.10750	0.12700	0.04568	0.03110	...	14.50

5 rows × 32 columns

Linear Regression

Singular Linear Regression

Identifying the Dependent and Independent Vairables

The first step that we did on singular linear regression is to indentify the dependent (y) and independent variables (x). Dependent variables are used interchangeably with *target variables* in this paper while independent variables are used interchangeably with *input features*. We analyzed each variable through correlation analysis to check the varaibles we can use the simple linear regression. In the data set that was given, all values are continous except for the `Diagnosis` which is changed to either 1 or 0 thus, it is excluded along with the `ID` .

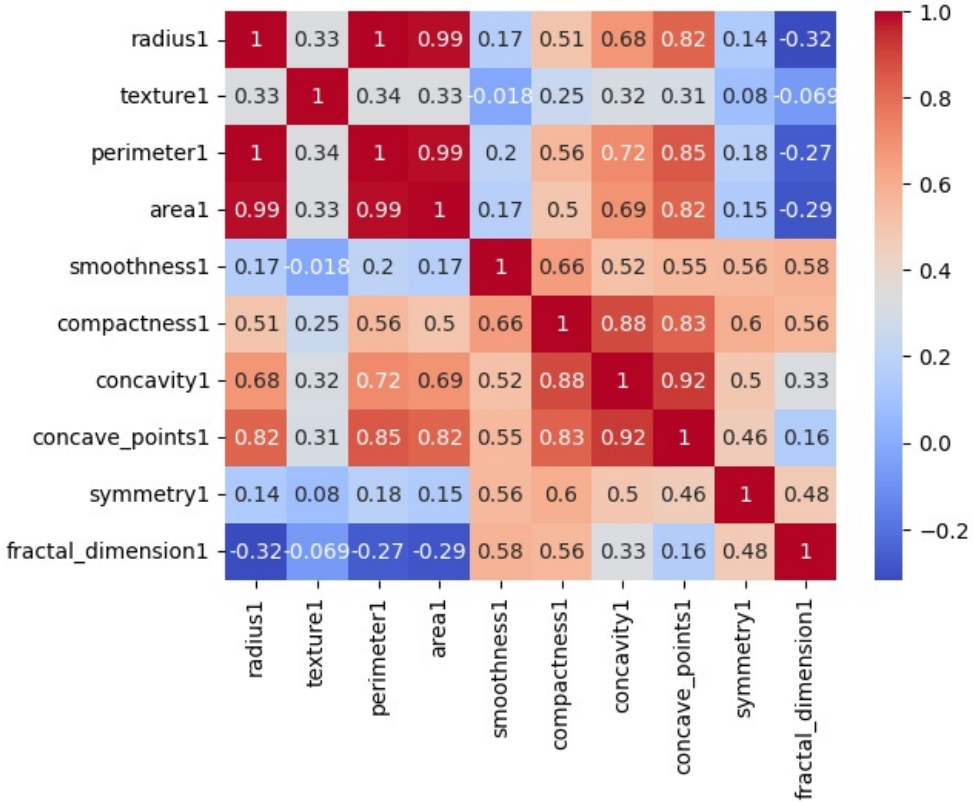
We performed the simple correlation analysis on all variables of the first set of variables (Cell 1) as shown below.

In [ ]:

```
#performing a simple correlation analysis
import seaborn as sns

cell1 = namesDF[['radius1','texture1','perimeter1','area1','smoothness1','compactness1', 'concavity1','concave_po
ints1','symmetry1', 'fractal_dimension1']]
resultsCorr = cell1.corr()
sns.heatmap(resultsCorr,cmap='coolwarm',annot=True)
```

Out[ ]:

<Axes: >



The chosen dependent and independet variables are `concavity1` and `compactness1` , respectively. Although there are other variables that have positive strong correlation, these two variables are not directly related at all. Therefore, we aim to check the relationship of these two features of a nuclei cell that is a factor of identifying the breast cancer cell type. </br></br> After picking, we then proceeded on creating objects and creating a test and train data sets.

In [ ]:

```
# Creating objects for the model

X = namesDF[["concavity1"]]      #input Variable
y = namesDF[["compactness1"]]    #target variable
```

Splitting the Dataset

In this section, we split the data set into two: the training set and test set. The test size would be 30% of the orginal dataset(171) and the random state would be set to 0

In [ ]:

```
#splitting the data to test and train using sklearn
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3,
                                                    random_state = 0)
```

Creating the model

In this section, we created the model for linear regression with by using the library sklearn

In [ ]:

```
#importing LinearRegression on sklearn
from sklearn.linear_model import LinearRegression

#model creation
model = LinearRegression()
```

In [ ]:

```
#fitting the model with input and target variables
model.fit(X_train, y_train)
```

Out[ ]:

```
▾ LinearRegression
LinearRegression()
```

Evaluating the model

In order to evaluate the model, we used a scatter plot to visualize the model and `r2_score` from sklearn.metrics.

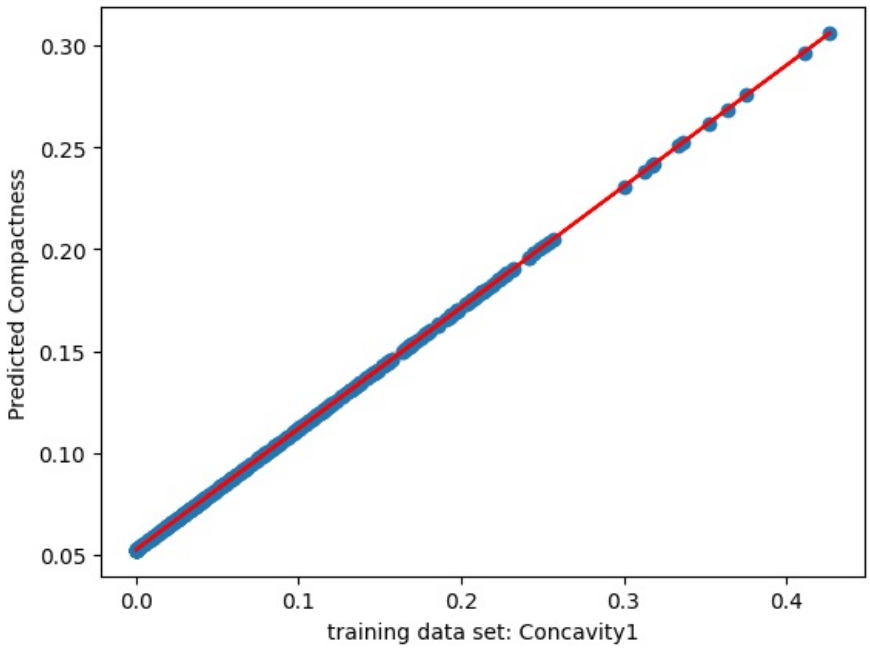
In [ ]:

```
#Creating predictions of train and test
Train_y_pred = model.predict(X_train)
Test_y_pred = model.predict(X_test)
```

Training Dataset: Visualizing the Model

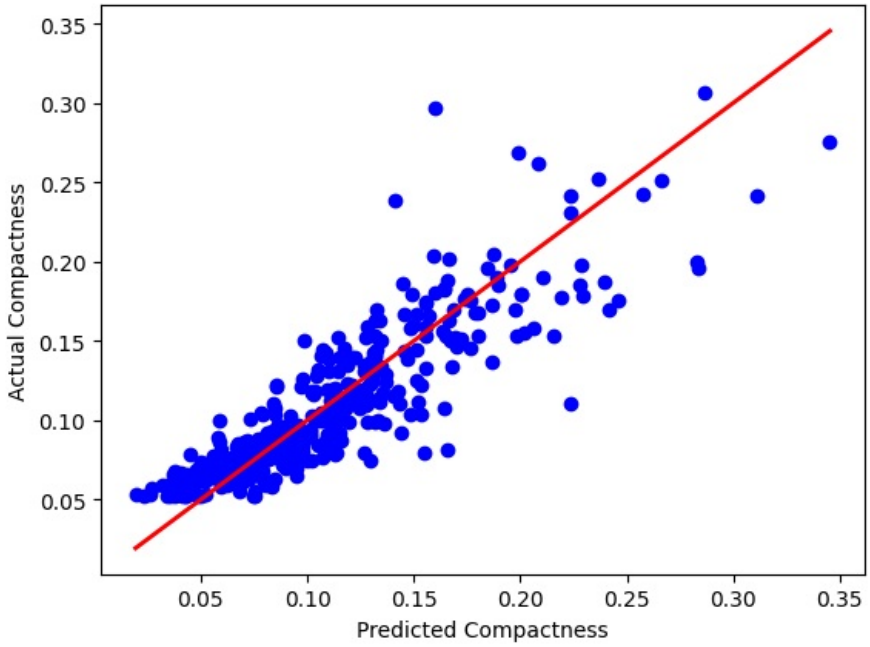
In [ ]:

```
# Visualizing the training dataset and prediction
import matplotlib.pyplot as plt

#plotting
plt.scatter(X_train, Train_y_pred)
plt.plot(X_train, Train_y_pred, color='red')

#labels
plt.xlabel("training data set: Concavity1")
plt.ylabel("Predicted Compactness")

plt.show()
```



In [ ]:

```
# Visualizing the model prediction against the actual values

#plotting
plt.scatter(y_train, Train_y_pred,color='blue')
plt.plot([y_train.min(), y_train.max()], [y_train.min(), y_train.max()], color='red', linestyle='-', linewidth=2,
label='Regression Line')

#labels
plt.xlabel("Predicted Compactness")
plt.ylabel("Actual Compactness")

plt.show()
```



Training DataSet: Using `r2_score`

In [ ]:

```
from sklearn.metrics import r2_score
r2_score(y_train, Train_y_pred)
```

Out[ ]:

```
0.7831333872965175
```

Remarks: The scatter plot shows that there are clusters near the 0.05 to 0.20 compactness. This can mean that the prediction on these levels are higher compared to values higher than this. There are some outliers that can be seen in the scatterplot. Additionaly, the model's r2_score is decent with a value of 78.31%. The result almost reached 80% which can be considered a good score for predicting values. Nevertheless, the model using the training data set is close to the regression line.

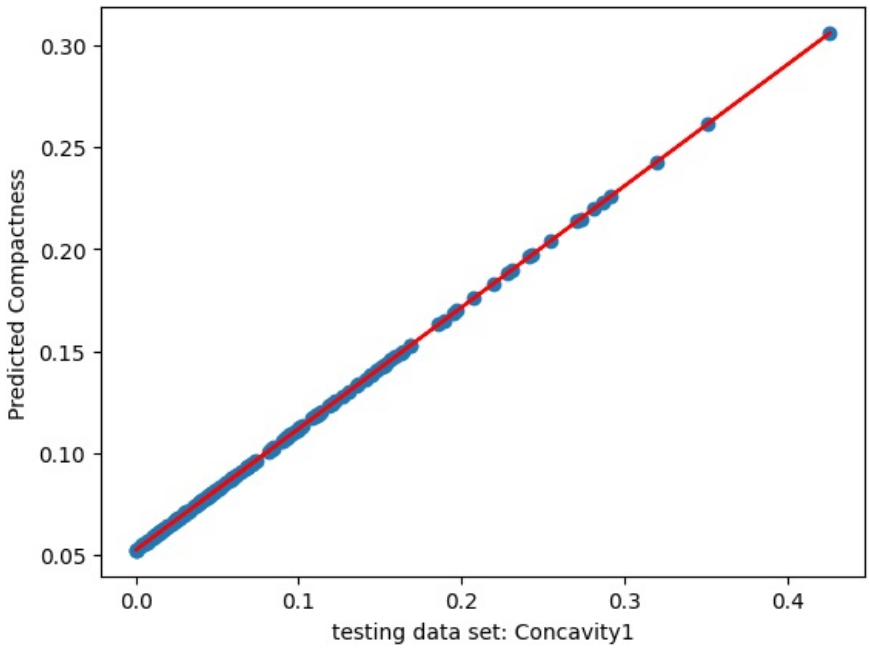
Test Dataset: Visualizing the model

In [ ]:

```
# Visualizing the training dataset and prediction

#plotting
plt.scatter(X_test, Test_y_pred)
plt.plot(X_test, Test_y_pred, color='red')

#labels
plt.xlabel("testing data set: Concavity1")
plt.ylabel("Predicted Compactness")
```

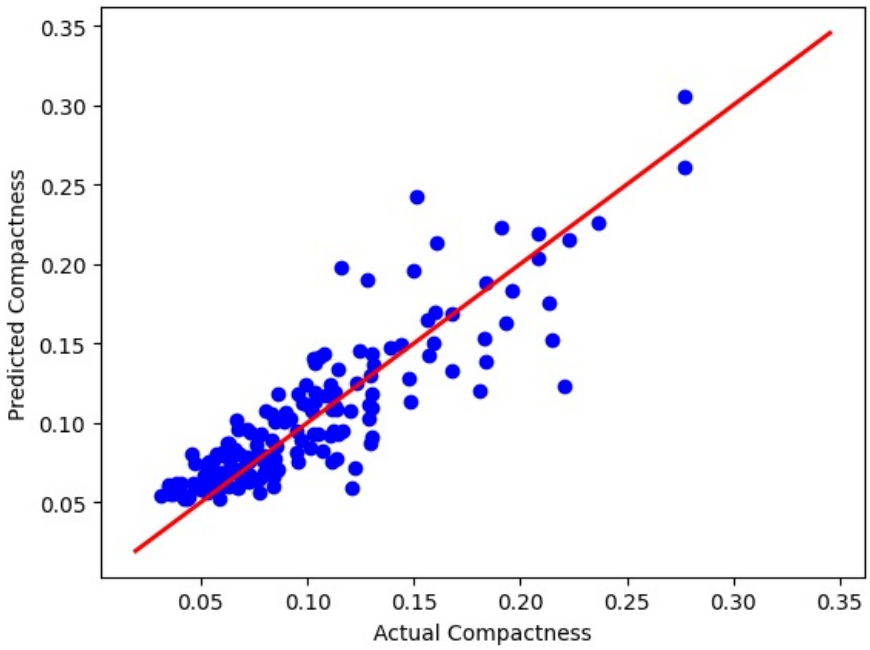
Out[ ]:

```
Text(0, 0.5, 'Predicted Compactness')
```



In [ ]:

```
# Visualizing the model prediction against the actual values

plt.scatter(y_test, Test_y_pred,color='blue')
plt.plot([y_train.min(), y_train.max()], [y_train.min(), y_train.max()], color='red', linestyle='-', linewidth=2,
label='Regression Line')
plt.xlabel("Actual Compactness")
plt.ylabel("Predicted Compactness")
plt.show()
```



Test Dataset: Using `r2_score()`

In [ ]:

```
from sklearn.metrics import r2_score
r2_score(y_test, Test_y_pred)
```

Out[ ]:

```
0.7543855412323663
```

Remarks: The scatterplot shows that the values of actual and predicted is close to the regression line. There are outliers that can be observed too. The model has r2_score of the test dataset resutled to 0.7543855412323663 or approximately 75.43%. The score is not that far in the train value results. This means that the model has a good prediction when faced against an unknown data.

Using mean_squared_error and mean_absolute_error

In [ ]:

```
from sklearn.metrics import mean_squared_error

# Calculate MSE
mse = mean_squared_error(y_test, Test_y_pred)
print("Mean Squared Error (MSE):", mse)
```

```
Mean Squared Error (MSE): 0.0005873862208731265
```

In [ ]:

```
from sklearn.metrics import mean_absolute_error


# Calculate MAE
mae = mean_absolute_error(y_test, Test_y_pred)
print("Mean Absolute Error (MAE):", mae)
```

```
Mean Absolute Error (MAE): 0.017982966204007467
```

Remarks : As shown, the mean squared error and absolute mean error of the model is low against an unknown value. This means that the tendency of the data producing an error in its prediciton is small.

Multiple Linear Regression

Defining the Independent (Input Features) and Dependent (Target) Variables

In this step, we will now create separate dataframes for our independent (x) and dependent (y) variables. For multiple linea regression, we will be using multiple independent variables to predict a single dependent variable.

Specifically for this dataset, we have decided to use the columns `radius1` until the column `symmetry1` as the independent variables or the input features. These dependent variable or the target variable chosen is the `fractal_dimension1` . By doing so, we aim to identify the extent of the relationship among the chosen independent variables on the fractal dimension or the tightness of the branches of a cell cluster [1].

We will also separate the dataset into two: *testing* and *training*. Splitting the dataset provides us better insights about how good our model is.

Reference:

[1] “Research shows human cells assembling into fractal-like clusters,” Brown University, 2019. https://www.brown.edu/news/2019-08-12/fractals#:~:text=Further%20analysis%20showed%20that%20the%20fractal%20dimension%20of,predicted%20theoretically%20for%20the%20diffusion-limited%20aggregation%20of%20particles (https://www.brown.edu/news/2019-08-12/fractals#:~:text=Further%20analysis%20showed%20that%20the%20fractal%20dimension%20of,predicted%20theoretically%20for%20the%20diffusion-limited%20aggregation%20of%20particles). (accessed Mar. 03, 2024).

In [ ]:

```
# Create training features

X = namesDF.loc[:, "radius1":"symmetry1"]
X
```

Out[ ]:

	radius1	texture1	perimeter1	area1	smoothness1	compactness1	concavity1	concave_points1	symmetry1
0	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.08690	0.07017	0.1812
1	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.19740	0.12790	0.2069
2	11.42	20.38	77.58	386.1	0.14250	0.28390	0.24140	0.10520	0.2597
3	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.19800	0.10430	0.1809
4	12.45	15.70	82.57	477.1	0.12780	0.17000	0.15780	0.08089	0.2087
...	...	...	...	...	...	...	...	...	...
563	21.56	22.39	142.00	1479.0	0.11100	0.11590	0.24390	0.13890	0.1726
564	20.13	28.25	131.20	1261.0	0.09780	0.10340	0.14400	0.09791	0.1752
565	16.60	28.08	108.30	858.1	0.08455	0.10230	0.09251	0.05302	0.1590
566	20.60	29.33	140.10	1265.0	0.11780	0.27700	0.35140	0.15200	0.2397
567	7.76	24.54	47.92	181.0	0.05263	0.04362	0.00000	0.00000	0.1587

568 rows × 9 columns

Remarks: The code above allows us to store the values of the columns in the `namesDF` dataframe starting from `radius1` until `symmetry` to the variable `X`. These columns will serve as the independent variables in our multiple linear regression. These variables are the variables used to predict the value of the dependent variable [4]. As we can see, when `X` was called, the specified columns were listed.

[4] S. Taylor, "Multiple Linear Regression," Corporate Finance Institute, Nov. 21, 2023. https://corporatefinanceinstitute.com/resources/data-science/multiple-linear-regression/#:~:text=It%20is%20sometimes%20known%20simply%20as%20multiple%20regression%2C,variable%20are%20known%20as%20independent' (https://corporatefinanceinstitute.com/resources/data-science/multiple-linear-regression/#:~:text=It%20is%20sometimes%20known%20simply%20as%20multiple%20regression%2C,variable%20are%20known%20as%20independent' (accessed Mar. 04, 2024).

In [ ]:

```
# Create target variable (dependent variable)
y = namesDF["fractal_dimension1"]
```

Remarks: The aboce code on the other hand, stores the values of the `fractal_dimension1` column to `y`. This will be used as the dependent variable in the multiple linear regression model.

Splitting the Dataset

In [ ]:

```
# Split the dataset so a portion of it can be used for training
# and a portion for testing

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3,
                                                    random_state = 0)
```

Remarks: Before proceeding, we split the data into two different portions—*training* and *testing* data. This will be used to train and evaluate the model later on. It is noted that the dataframe we are splitting are the selected columns we have defined in the earlier parts, `X` which contains the independent variables, and `y` which contains the dependent variable.

Creating and Training the Model

In this step, we will now create the model to be used for the multiple linear regression.

In [ ]:

```
from sklearn.linear_model import LinearRegression
lr = LinearRegression()

lr.fit(X_train, y_train)
```

Out[ ]:

```
▾ LinearRegression
LinearRegression()
```

Remarks: To perform the multiple linear regression, we will first import the necessary module needed, for linear regression models, we will use the `LinearRegression` module from the `sklearn.linear_model` library.

In [ ]:

```
# Try to predict fractal_dimension1 using the independent variables (x)

y_pred_train = lr.predict(X_train)
```

Remarks: By using the `predict()` method, we are now performing the prediction of our dependent variable, `fractal_dimension1` using the defined independent variables.

Evaluating the Model

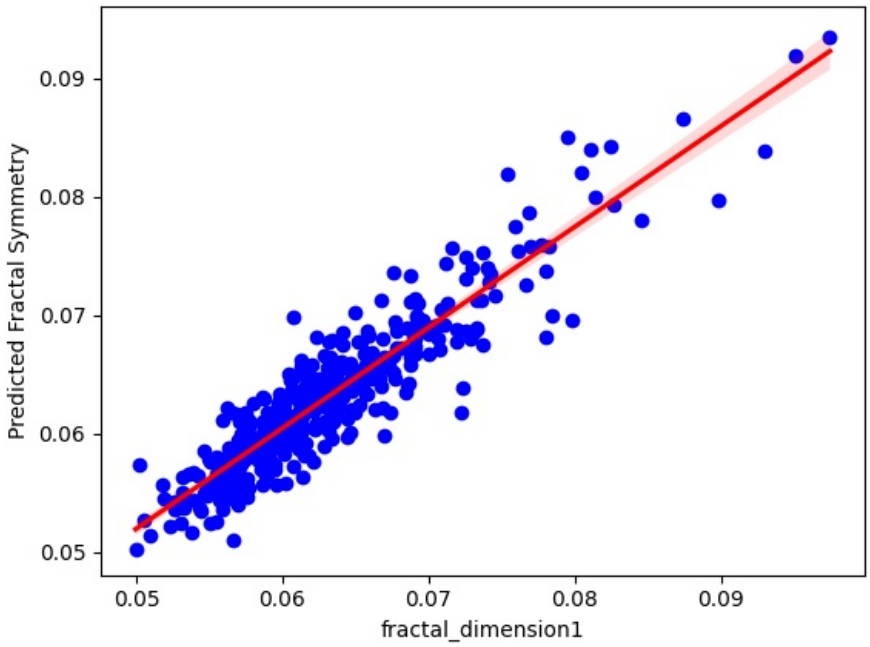
Several methods were used to evaluate the multiple linear regression model. The first one is through visualizing the actual and predicted values by plotting them in a scatterplot.

Through visualization

In this part, we plotted the actual fractal symmetry values as well as the predicted fracal symmetry values.

In [ ]:

```
# How good is the prediction?

import matplotlib.pyplot as plt
import seaborn as sns

plt.scatter(y_train, y_pred_train, c="blue")
plt.xlabel("Actual Fractal Symmetry")
plt.ylabel("Predicted Fractal Symmetry")
sns.regplot(x=y_train, y=y_pred_train, scatter=False, color="red")
plt.show()
```



Remarks: In the above graph, there are regions where the data points are closer to the diagonal line. These data points represent those that are correctly predicted by the model. On the other hand, data points away from the diagonal line are those that are incorrectly predicted, above the diagonal line are overestimated values while below are underestimated values.

Using r2_score

We also implement the `r2_score` method from the `sklearn.metrics` library. This returns the regression score which indicates how good does the model predict something, a score of 1 is the highest score [5].

[5] “sklearn.metrics.r2_score,” scikit-learn, 2024. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.r2_score.html (https://scikit-learn.org/stable/modules/generated/sklearn.metrics.r2_score.html) (accessed Mar. 04, 2024).

In [ ]:

```
from sklearn.metrics import r2_score
```

In [ ]:

```
r2_score(y_train, y_pred_train)
```

Out[ ]:

```
0.8501783056922954
```

Remarks: From the above results, we can see that there is a regression score of approximately 0.85. This value ranges from 0 to 1, and a value closer to 1 suggests a better model.

Using mean_squared_error and mean_absolute_error

Mean squared error (MSE) and mean absolute error (MAE) are machine learning metrics used to assess the performance of regression models such as multiple linear regression. Both give us the idea about the errors of a model with MSE giving more focus on the error since it uses squared error unlike MAE which uses absolute error in its calculations [9].

[9] S. Allwright, “MSE vs MAE, which is the better regression metric?,” Stephen Allwright, Jul. 07, 2022. https://stephenallwright.com/mse-vs-mae/ (https://stephenallwright.com/mse-vs-mae/) (accessed Mar. 05, 2024).

In [ ]:

```
# Compute mean squared error and mean absolute error

from sklearn.metrics import mean_squared_error, mean_absolute_error

print(f"Mean Squared Error: {mean_squared_error(y_train, y_pred_train)}")
print(f"Mean Absolute Error: {mean_absolute_error(y_train, y_pred_train)}")
```

```
Mean Squared Error: 7.728043432022982e-06
Mean Absolute Error: 0.0020971449089168194
```

Remarks: Based on the results obtained above, both the mean squared error and the mean absolute error of the results using training data are low values, indicating that the model's predictions are not that far from the actual values.

Evaluating the model using the testing data.

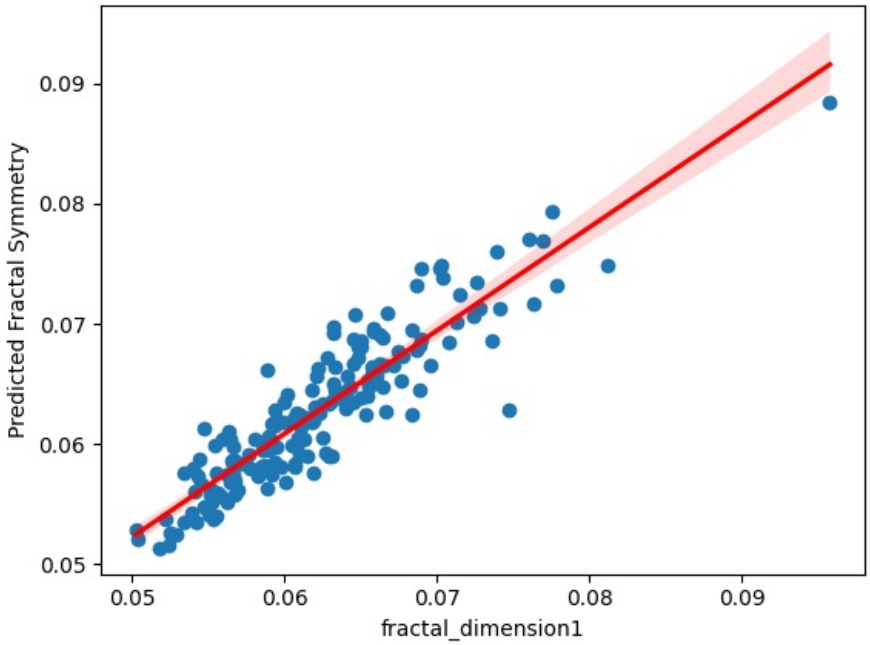
In the next part, we will evaluate the model based on the testing data. It is to be noted that we have separated the dataset into two: training and testing dataset. The testing done in the prior tabs were done in the training dataset. This time, we will use the testing dataset, a dataset which the model has not seen yet.

In [ ]:

```
# Evaluating the model with the testing data:

y_pred_test = lr.predict(X_test)
```

In [ ]:

```
# How good is the prediction?

import matplotlib.pyplot as plt
plt.scatter(y_test, y_pred_test)
plt.xlabel("Actual Fractal Symmetry")
plt.ylabel("Predicted Fractal Symmetry")
sns.regplot(x=y_test, y=y_pred_test, scatter=False, color="red")

plt.show()
```



In [ ]:

```
r2_score(y_test, y_pred_test)
```

Out[ ]:

```
0.8202033201548747
```

Remarks: From the above result, we can see a slight decrease with the regression score, which resulted to approximately 0.82. Nevertheless, since the value is still close to 1, this suggests that the model still is pretty good.

Using mean_squared_error and mean_absolute_error

In [ ]:

```
# Compute mean squared error and mean absolute error

from sklearn.metrics import mean_squared_error, mean_absolute_error

print(f"Mean Squared Error: {mean_squared_error(y_test, y_pred_test)}")
print(f"Mean Absolute Error: {mean_absolute_error(y_test, y_pred_test)}")
```

```
Mean Squared Error: 7.909105645438319e-06
Mean Absolute Error: 0.00211133788425646
```

Remarks: The MSE and MAE for the testing values were also small further confirming that the model is performing well in predicting the target variable values.

Showing the Difference Between Actual Values and Predicted Values (Testing Values)

To further see the model's accuracy, we can try printing the actual values of the fractal symmetry and the predicted values given by the multiple linear regression model.

In [ ]:

```
pred_y_df = pd.DataFrame({'Actual Value' : y_test, 'Predicted Value' : y_pred_test, 'Difference' : y_test - y_pre
d_test})
pred_y_df
```

Out[ ]:

	Actual Value	Predicted Value	Difference
512	0.05640	0.060395	-0.003995
457	0.05449	0.056720	-0.002230
439	0.06640	0.068818	-0.002418
298	0.06556	0.064755	0.000805
37	0.05504	0.054100	0.000940
...	...	...	...
7	0.07389	0.076020	-0.002130
408	0.05597	0.060466	-0.004496
523	0.06891	0.068153	0.000757
361	0.06183	0.061790	0.000040
552	0.06576	0.065778	-0.000018

171 rows × 3 columns

In [ ]:

```
mean_difference = pred_y_df["Difference"].mean()
print(mean_difference)
```

```
-0.0005314655160501634
```

Remarks: As we can see from the results above, the mean difference between the actual and predicted `fractal_symmetry1` values is around -0.0005.

Polynomial Linear Regression

Defining the Independent (Input Features) and Dependent (Target) Variables

In [11]:

```
# Define input features and target variables

X2 = namesDF.loc[:, "radius2":"symmetry2"]
y2 = namesDF["fractal_dimension2"]
```

For the first part, we will define the input features and the dependent variables The input features are those variables that will be used to predict the target variable. Specifically on this model, the columns starting from `radius2` until `symmetry2` will be used as the input features and the target variable is the `fractal_symmetry2` .

Splitting the Dataset

In [12]:

```
# Split the dataset so a portion of it can be used for training
# and a portion for testing

from sklearn.model_selection import train_test_split
X_train2, X_test2, y_train2, y_test2 = train_test_split(X2, y2, test_size = 0.3,
                                                        random_state = 20)
```

Just like in the previous models, we will still split the dataset into two: *testing* and *training*. The training dataset will be the only thing that is visible to the model while the testing dataset should be left "unseen" by the model. This allows us to evaluate the model accordingly.

Data Scaling

This step involves scaling the data so that all of them are within the same ranges. While not required, data scaling is considered to be a good practice when it comes to polynomial linear regression [2]. For this activity, we will test the model with a) scaled values, and b) unscaled values.

In [13]:

```
# Bring the features to the same range (Data Scaling)

from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_train2_scaler = scaler.fit_transform(X_train2)
X_test2_scaler = scaler.fit_transform(X_test2)
```

Remarks: The code above utilizes the `StandardScaler` from `sklearn.preprocessing` to scale the data in our defined independent and dependent variables.

Creating and Training the Model

In [14]:

```
# Model Development

from sklearn.linear_model import LinearRegression
PolyModel = LinearRegression()
```

For unscaled values:

First, we train the model on unscaled values.

In [15]:

```
from sklearn.preprocessing import PolynomialFeatures
poly = PolynomialFeatures(degree=6)
UnscaledXPolyTrain = poly.fit_transform(X_train2)
UnscaledXTestPoly = poly.fit_transform(X_test2)
poly.fit(UnscaledXPolyTrain, y_train2)
PolyModel.fit(UnscaledXPolyTrain, y_train2)
```

Out[15]:

▾ LinearRegression

LinearRegression()

In [16]:

```
y_pred2 = PolyModel.predict(UnscaledXTestPoly)
```

Evaluating the Model

Using `mean_absolute_error`

The mean absolute error tells us the average of all the absolute errors in calculations done by the model [8]. This suits our polynomial model as there are multiple predictor variables present. A lower MAE value indicates better results.

[8] L. Peng, S. Liu, R. Liu, and L. Wang, “Effective long short-term memory with differential evolution algorithm for electricity price prediction,” Energy, vol. 162, pp. 1301–1314, Nov. 2018, doi: https://doi.org/10.1016/j.energy.2018.05.052 (https://doi.org/10.1016/j.energy.2018.05.052).

In [20]:

```
from sklearn.metrics import mean_absolute_error
y_pred_train = PolyModel.predict(UnscaledXPolyTrain)
print(f"Mean Absolute Error (Train Values): {mean_absolute_error(y_train2, y_pred_train)}")
```

```
Mean Absolute Error (Train Values): 0.0006744630685052068
```

In [18]:

```
print(f"Mean Absolute Error (Test Values): {mean_absolute_error(y_test2, y_pred2)}")
```

```
Mean Absolute Error (Test Values): 0.6051931699175404
```

Remarks: As we can see from the results above, the mean absolute error of the testing and training data, have a big gap between them. This indicates that the model may have been overfitted.

Overfitting, while it may seem ideal since at first, the model kinds of predicting the values very accurately, it is not robust to new or unknown values.

Hence, we have to adjust some of our parameters that were used in the model like the `degree` parameter in `PolynomialFeatures` .

Updating `degree` parameter

In [21]:

```
from sklearn.preprocessing import PolynomialFeatures
poly = PolynomialFeatures(degree=2)
X_poly_train = poly.fit_transform(X_train2)
X_test_poly = poly.fit_transform(X_test2)
poly.fit(X_poly_train, y_train2)
PolyModel.fit(X_poly_train, y_train2)
```

Out[21]:

▾ LinearRegression

LinearRegression()

Evaluate using `mean_absolute_error`

In [22]:

```
y_pred_train = PolyModel.predict(X_poly_train)
print(f"Mean Absolute Error (Train Values): {mean_absolute_error(y_train2, y_pred_train)}")
```

```
Mean Absolute Error (Train Values): 0.0006631773700665347
```

In [23]:

```
y_pred2 = PolyModel.predict(X_test_poly)
print(f"Mean Absolute Error (Test Values): {mean_absolute_error(y_test2, y_pred2)}")
```

```
Mean Absolute Error (Test Values): 0.0011538656726622124
```

Remarks: When we changed the `degree` parameter into 2, we can notice that the gap between the MAE results from training data and testing data went down significantly. The `degree` parameter, according to sklearn's documentation for PolynomialFeatures, is responsible for the complexity of the polynomial features generated for the model [6].

[6] “sklearn.preprocessing.PolynomialFeatures,” scikit-learn, 2024. https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.PolynomialFeatures.html (https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.PolynomialFeatures.html) (accessed Mar. 04, 2024).

For scaled values:

Next, we repeat the steps done above on scaled values to see whether the model performs better.

In [ ]:

```
poly = PolynomialFeatures(degree=6)
ScaledXPolyTrain = poly.fit_transform(X_train2_scaler)
ScaledXTestPoly = poly.fit_transform(X_test2_scaler)
poly.fit(ScaledXPolyTrain, y_train2)
PolyModel.fit(ScaledXPolyTrain, y_train2)
```

Out[ ]:

▾ LinearRegression

LinearRegression()

In [ ]:

```
y_pred2 = PolyModel.predict(ScaledXTestPoly)
```

Evaluating the model using `mean_absolute_error`

In [ ]:

```
from sklearn.metrics import mean_absolute_error
print(f"MAE (Test Values): {mean_absolute_error(y_test2, y_pred2)}")
```

```
MAE (Test Values): 2.423287425194479
```

In [ ]:

```
y_pred_train = PolyModel.predict(ScaledXPolyTrain)
print(f"MAE (Train Values): {mean_absolute_error(y_train2, y_pred_train)}")
```

```
MAE (Train Values): 2.2864820998984436e-14
```

Remarks: Similar to the case with the unscaled values, there is a huge gap between the mean absolute error values when testing the model in testing data and training data.

Updating `degree` parameter

In [ ]:

```
poly = PolynomialFeatures(degree=2)
ScaledXPolyTrain = poly.fit_transform(X_train2_scaler)
ScaledXTestPoly = poly.fit_transform(X_test2_scaler)
poly.fit(ScaledXPolyTrain, y_train2)
PolyModel.fit(ScaledXPolyTrain, y_train2)
```

Out[ ]:

▾ LinearRegression

LinearRegression()

Remarks: In the code above, we updated the `degree` parameter by reducing it to 2 to see whether the model will be able to perform better.

Evaluate using `mean_absolute_error`

In [ ]:

```
y_pred2 = PolyModel.predict(ScaledXTestPoly)
print(f"MAE (Test Values): {mean_absolute_error(y_test2, y_pred2)}")
```

```
MAE (Test Values): 0.001037535710653574
```

In [ ]:

```
y_pred_train = PolyModel.predict(ScaledXPolyTrain)
print(f"MAE (Train Values): {mean_absolute_error(y_train2, y_pred_train)}")
```

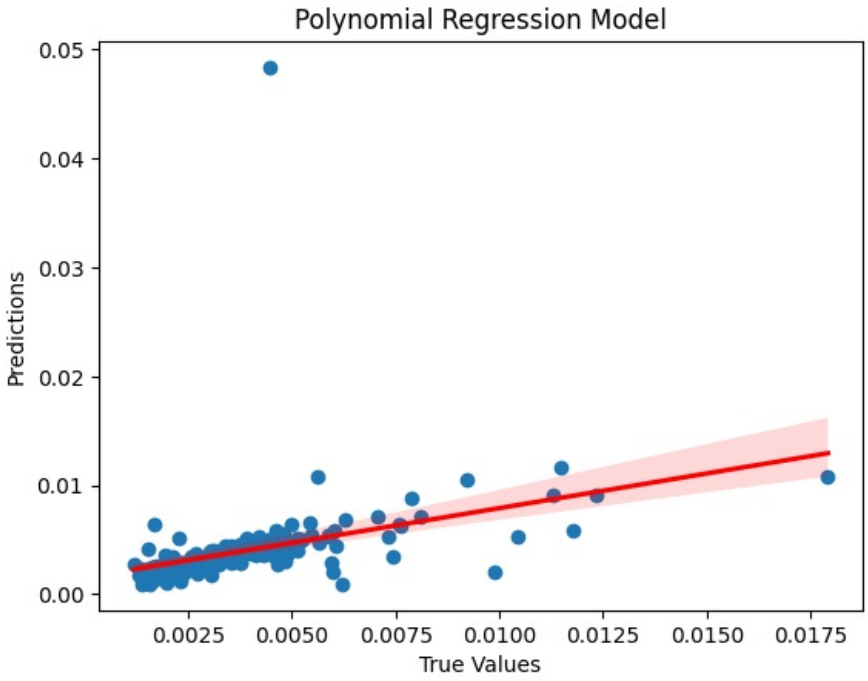
```
MAE (Train Values): 0.0006631773699074196
```

Remarks: Adjusting the degree value significantly reduced the gap between the two mean absolute error values.

Visualizing the Model

In [27]:

```
# Plotting

from matplotlib import pyplot as plt
import seaborn as sns

plt.scatter(y_test2, y_pred2)
sns.regplot(x=y_test2, y=y_pred2, scatter=False, color="red")
plt.xlabel('True Values')
plt.ylabel('Predictions')
plt.title('Polynomial Regression Model')
plt.show()
```



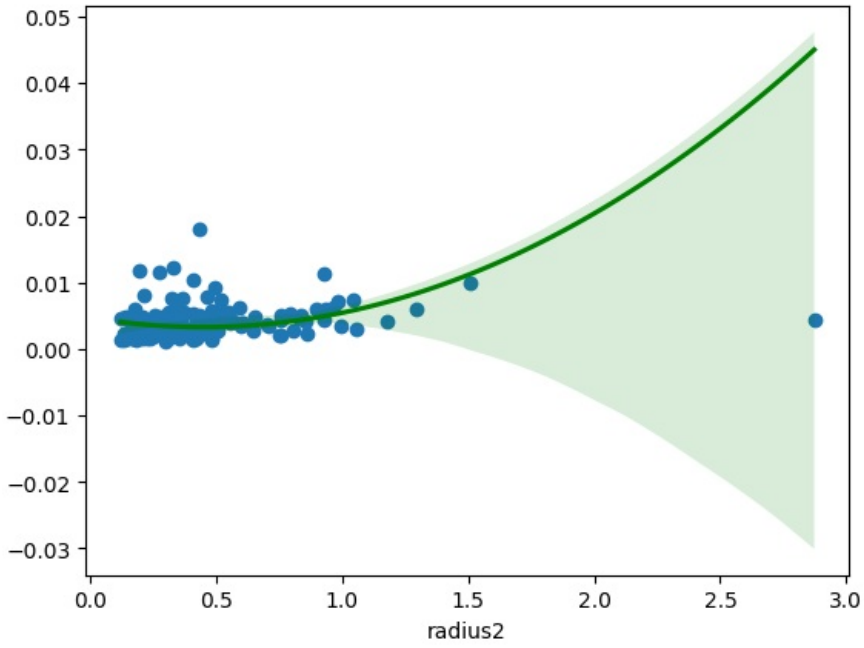
Remarks: The graph above shows the scatterplot for the actual values and predicted values given by the model. Similarly with the previous scatterplots from other models, the data points closer to the diagonal line represent those that were correctly predicted by the model. On the other hand, data points away from the regression line were either overestimated (when located above) or underestimated (when located below) by the model.

In [59]:

```
# Plotting Regression Curve between one of the input features, radius2 and the
# target variable, fractal_dimension2

plt.scatter(X_test2['radius2'], y_test2)
sns.regplot(x=X_test2['radius2'], y=y_pred2, color="g", scatter=False, order=2)
```

Out[59]:

```
<Axes: xlabel='radius2'>
```



Remarks: To visualize the polynomial regression model, we have chosen one variable from the input features, `radius2` and the target variable, `fractal_dimension2` . We plotted the data points first using `plt.scatter` then graphed the regression curve using `sns.regplot` .

Similar to the degree value used in the polynomial regression model, we have used the `order=2` as one of the parameters in the `regplot` function, indicating a quadratic regression model.

Logistic Regression

Identifying the target and input variables

In this section, the input variables are all the continous variables that was taken from three different nuclei cell. The dependent or target variable would be the diagnosis, whether it is malignant or benign.

In [ ]:

```
#creating object

X = namesDF[['radius1','texture1','perimeter1','area1','smoothness1','compactness1', 'concavity1','concave_points
1','symmetry1', 'fractal_dimension1',
    'radius2','texture2','perimeter2','area2','smoothness2','compactness2','concavity2','concave_points2','symmet
ry2','fractal_dimension2',
    'radius3','texture3','perimeter3','area3','smoothness3','compactness3','concavity3','concave_points3','symmet
ry3','fractal_dimension3'
    ]]  #input Variable
y = namesDF[["Diagnosis"]]    #target variable
```

Splitting the Dataset

In this section, we split the data set into two training and test. The test size of the data would still be the 30% of the data set while the remaining will be the training set.

In [ ]:

```
#splitting the data to test and train

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3,
                                                    random_state = 0)
```

Creating the Model

In order to apply logistic regression, we used the library sklearn. This allowed us to create a model for our prediction

In [ ]:

```
from sklearn.linear_model import LogisticRegression
#model creation
model = LogisticRegression()
```

In [ ]:

```
#fitting the model with input and target variables
model.fit(X_train, y_train)
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/utils/validation.py:1143: DataConversionWarning: A c
olumn-vector y was passed when a 1d array was expected. Please change the shape of y to (n_samples,
), for example using ravel().
  y = column_or_1d(y, warn=True)
/usr/local/lib/python3.10/dist-packages/sklearn/linear_model/_logistic.py:458: ConvergenceWarning: l
bfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
```

Out[ ]:

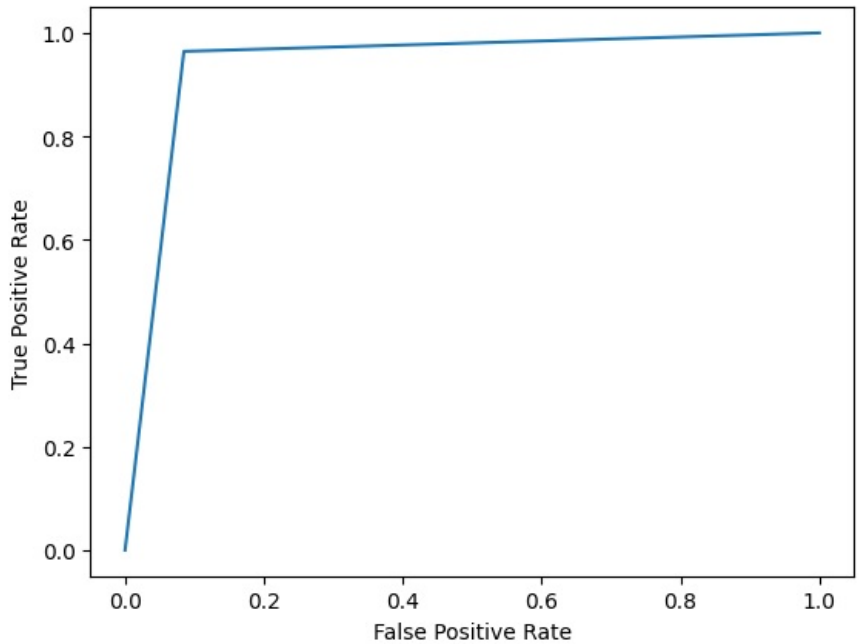
```
▾ LogisticRegression
LogisticRegression()
```

Visualizing the ROC curve

ROC curve is used to assess the overall diagnostic performance of a test and to compare the performance of two or more diagnostic tests.

In [ ]:

```
from sklearn import metrics
#define metrics
y_pred = model.predict(X_test)

#Applying the roc_curve
fpr, tpr, _ = metrics.roc_curve(y_test,  y_pred)

#create ROC curve
plt.plot(fpr,tpr)
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.show()
```



Remarks : The location of the ROC curve is at the upper left of the diagram. This implies that the model has an excellent performance in predicting the diagnosis on an unknown dataset.

Evaluating the Model

Using confusion_matrix()

In this section, to evaluate the model we used a confusion matrix to determine the true positive and negative and the false positive and negatives on the given testing data set.

In [ ]:

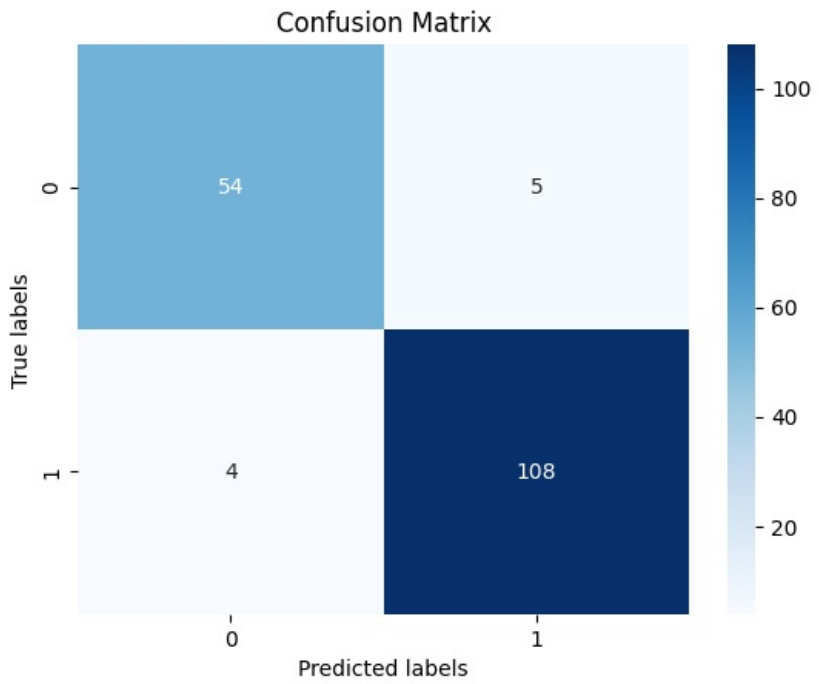
```
from sklearn.metrics import confusion_matrix
#predicting using the data set for testing
y_pred = model.predict(X_test)


#applying the confusion_matrix()
conf_m = confusion_matrix(y_test, y_pred)
```

In [ ]:

```
#creating a visualization of confusion matrix
import seaborn as sns
sns.heatmap(conf_m,annot =True,fmt='d', cmap='Blues',
            xticklabels=np.unique(y_test), yticklabels=np.unique(y_test))

#labels
plt.xlabel('Predicted labels')
plt.ylabel('True labels')
plt.title('Confusion Matrix')
```

Out[ ]:

```
Text(0.5, 1.0, 'Confusion Matrix')
```



Remarks: The confusion matrix shows that the values of the true positive and negative is significanlty higher than the false negatives and positives of the test data set. The number of false negatives is 5 while 4 for false positives. These tells that the model has a low tendency of producing an error on a 171 different test values.

Using classification_report()

In this section, we used the classification_report function of sklearn.metrics to get the summary the precision and accuracy of the model especially on the two classes that we need to classify: Malignant and Benign

In [ ]:

```
from sklearn.metrics import classification_report

report = classification_report(y_test, y_pred,target_names=['malignant', 'benign'])
print(report)
```

```
              precision    recall  f1-score   support

   malignant       0.93      0.92      0.92        59
      benign       0.96      0.96      0.96       112

    accuracy                           0.95       171
   macro avg       0.94      0.94      0.94       171
weighted avg       0.95      0.95      0.95       171
```

Remarks: The classification_report returns that the two classes has a high precision, recall and f1-score. Additionally the overall accuracy of the model is approximately 95%. Thus, the model is performing excellent even on unknown test values.

Decision Tree

Identifying the target and input variables

In this section, the input variables are all the continous variables that was taken from three different nuclei cell. The dependent or target variable would be the diagnosis, whether it is malignant or benign.

In [ ]:

```
# Define independent variables / input features and target/dependent variable

from sklearn.model_selection import train_test_split
X = namesDF.drop(columns = ["Diagnosis", "ID"])
y = namesDF['Diagnosis']
```

Splitting the Dataset

Splitting the dataset is a good practice when training models. When we split the dataset, we are actually preparing two datasets, one for the consumption of the model for its training, and one for verifying the model's capabalities through testing never before seen values—testing values.

In [ ]:

```
# Split the dataset into training and testing datasets

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
                                                    random_state = 25)
```

Creating and Training the Model

In [ ]:

```
# Create model

from sklearn.tree import DecisionTreeClassifier
dtc = DecisionTreeClassifier()
dtc2 = DecisionTreeClassifier(criterion="entropy", max_depth=3)
```

Remarks: In the above code, we define two different models, one with no added parameters and one with two additional parameters: `criterion` and `max_depth`. The `criterion` parameter can be defined as `gini`, `entropy`, or `log_loss`. While `max_depth` defines the maximum depth of the tree. If none is specified, the tree will grow until all leaves are pure or contains less than min_samples_split samples [3].

[3] “sklearn.tree.DecisionTreeClassifier,” scikit-learn, 2024. https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html (https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html) (accessed Mar. 03, 2024).

In [ ]:

```
# Fit the models into the training data.

dtc = dtc.fit(X_train, y_train)
dtc2 = dtc2.fit(X_train, y_train)
```

In [ ]:

```
predictions = dtc.predict(X_test)
print("Predictions using unparametrized model: ")
print(predictions)
print("\n")

print("Predictions using parametrized model: ")
predictions2 = dtc2.predict(X_test)
print(predictions2)
```

```
Predictions using unparametrized model:
[0 1 1 1 1 0 1 1 1 1 1 0 1 1 1 0 1 0 1 1 0 0 1 1 1 0 0 1 1 1 1 0
 1 1 1 1 0 0 0 1 1 1 1 1 0 1 1 1 0 1 1 1 1 1 0 1 1 1 0 1 1 1 0 1 1 1 1
 1 1 0 0 1 0 1 1 1 1 1 0 1 1 1 0 1 1 1 0 1 0 1 1 1 0 1 0 0 1 0 0 1 0 1 0 0
 1 0 1 1 1 1 1 0 0 0 1 0 1 1 1 1 1 1 1 0 0 1 0 1 1 0 1 1 0 1 1 1 1 1 1 1 1 0
 0 1 1 0 1 1 1 1 1 1 1 1 1 0 1 0 1 1 1 0 1 0 1 1]


Predictions using parametrized model:
[0 1 1 1 1 0 1 1 1 1 1 1 0 1 1 1 0 1 0 1 1 0 0 1 1 1 0 0 1 1 1 1 0 0 1 1 1 1 0
 1 1 1 1 0 0 0 1 1 1 1 1 0 1 1 1 0 0 1 1 1 1 0 1 1 0 1 0 1 0 1 1 1 0 1 1 1 1
 1 1 0 1 1 0 1 1 1 1 0 0 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 0 1 1 1 0 0 1 0 1 0 1
 1 0 1 1 1 1 1 1 0 0 0 1 0 1 1 1 1 1 1 1 0 0 1 0 1 1 1 1 1 0 1 1 1 1 1 1 0 1 1
 0 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 0 1 1 0 1 0 1 1]
```

Evaluating Model

The `sklearn` library provides us with multiple ways to evaluate our decision tree model. In this part, we will evaluate the model using some of the pre-existing performance metrics.

Using `score`

In [ ]:

```
# Determining model score based on training data

score1 = dtc.score(X_train, y_train)
print(f"Model 1 score (without parameters): {score1}")

score2 = dtc2.score(X_train, y_train)
print(f"Model 2 score (with parameters): {score2}")
```

```
Model 1 score (without parameters): 1.0
Model 2 score (with parameters): 0.9848866498740554
```

Remarks: From the performance metric used above, the model without defined parameters achieved a perfect score of 1, meaning, it is correct 100% of the time.

On the other hand, the model with defined parameters had 98% correct rate. The difference of the values may be explained because of the differences in their parameters.

Since the first model had no defined parameters for `criterion` and `max_depth` , the tree expands exhaustively. On the other hand, the second model has restrictions, not allowing the tree to expand fully.

In [ ]:

```
# Determining model score based on testing data

score1 = dtc.score(X_test, y_test)
print(f"Model 1 score (without parameters): {score1}")

score2 = dtc2.score(X_test, y_test)
print(f"Model 2 score (with parameters): {score2}")
```

```
Model 1 score (without parameters): 0.9064327485380117
Model 2 score (with parameters): 0.9181286549707602
```

Remarks: According to the output above, the model without parameters is correct approximately 90.6% of the time. On the other hand, the model with defined parameters is correct approximately 91.8% of the time.

It is noticeable how the model with defined parameters received better scores compared to the model without parameters when tested against testing data.

Using confusion_matrix

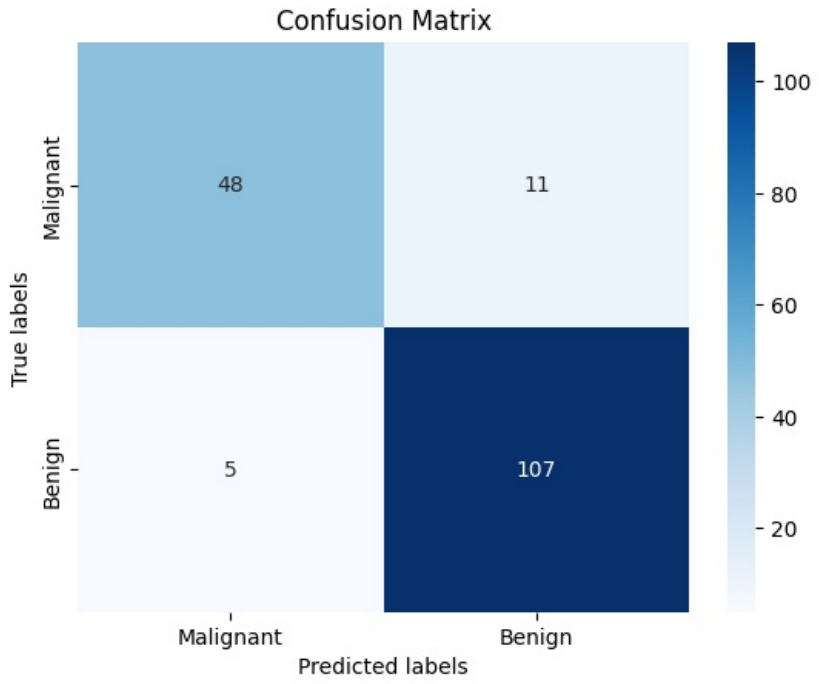
The confusion matrix allows us to see how many times the model correctly or incorrectly predicted the results.

In [ ]:

```
from sklearn.metrics import confusion_matrix
import seaborn as sns

cm = confusion_matrix(y_test, predictions, labels=[0, 1]) # for model without parameters

sns.heatmap(cm, annot =True, fmt='d', cmap='Blues',
            xticklabels=['Malignant', 'Benign'], yticklabels=['Malignant', 'Benign'])

#labels
plt.xlabel('Predicted labels')
plt.ylabel('True labels')
plt.title('Confusion Matrix')
```

Out[ ]:

```
Text(0.5, 1.0, 'Confusion Matrix')
```



Remarks: From the above results, the following can be said:

- 48 data points were correctly classified as malignant (true positive).
- 11 data points were actually malignant but predicted as benign (false negative).
- 5 data points were incorrectly classified as malignant (false positive).
- 107 data points were correctly classified as benign (true negative).

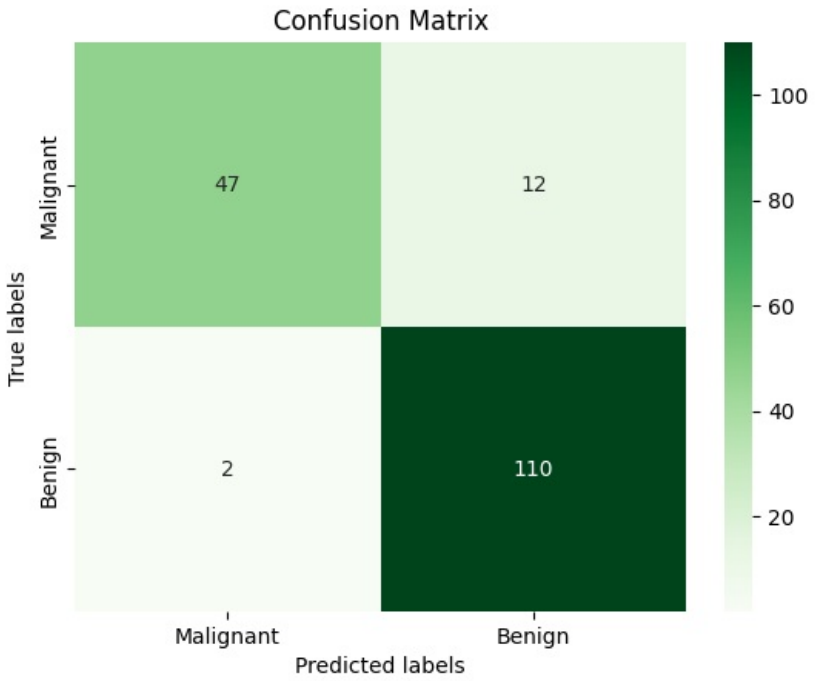
Based on the results, it can also be said that the model is able to correctly identify class 1 better with only 5 mistakes and 107 correct predictions.

In [ ]:

```
cm2 = confusion_matrix(y_test, predictions2, labels=[0, 1]) # for model with parameters

sns.heatmap(cm2, annot =True, fmt='d', cmap='Greens',
            xticklabels=['Malignant', 'Benign'], yticklabels=['Malignant', 'Benign'])

#labels
plt.xlabel('Predicted labels')
plt.ylabel('True labels')
plt.title('Confusion Matrix')
```

Out[ ]:

```
Text(0.5, 1.0, 'Confusion Matrix')
```



Remarks: From the confusion matrix above, the following can be said:

- 47 data points were correctly classified as malignant (true positive).
- 12 data points were actually malignant but predicted as benign (false negative).
- 2 data points were incorrectly classified as malignant (false positive).
- 110 data points were correctly classified as benign (true negative).

Using `classification_report`

This allows us to see a summary table containing the precision scores of the classifications done by the model for each label that we have. In our case, we can see the precision scores for malignant and benign predictions

In [ ]:

```
from sklearn.metrics import classification_report
print(classification_report(y_test, predictions, target_names=['malignant', 'benign']))
```

```
              precision    recall  f1-score   support

   malignant       0.91      0.81      0.86        59
      benign       0.91      0.96      0.93       112

    accuracy                           0.91       171
   macro avg       0.91      0.88      0.89       171
weighted avg       0.91      0.91      0.91       171
```

In [ ]:

```
print(classification_report(y_test, predictions2, target_names=['malignant', 'benign']))
```

```
              precision    recall  f1-score   support

   malignant       0.96      0.80      0.87        59
      benign       0.90      0.98      0.94       112

    accuracy                           0.92       171
   macro avg       0.93      0.89      0.91       171
weighted avg       0.92      0.92      0.92       171
```

Visualizing the Tree

For Model without Parameters

In [ ]:

```
import six
import sys
sys.modules['sklearn.externals.six'] = six
```

In [ ]:

```
from sklearn.externals.six import StringIO
from sklearn import tree

with open("breastcancer.dot", 'w') as f:
  f = tree.export_graphviz(dtc, out_file=f, feature_names=X_train.columns)
```

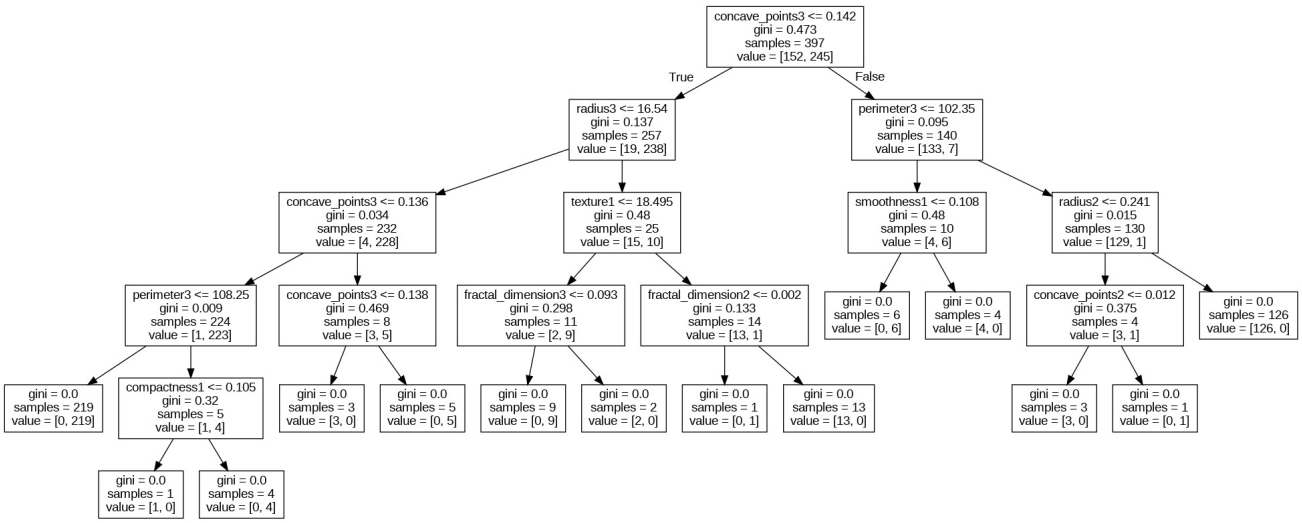
In [ ]:

```
# Convert immediate file to a graphic

!dot -Tpng breastcancer.dot -o breastcancer.png
```

In [ ]:

```
# Display the image

from IPython.display import Image
Image("breastcancer.png")
```

Out[ ]:



Remarks: In the decision tree generated above, we can notice different parameters inside a node.

One of the parameters is one of the features that we have fed the model. These features include `concave_points3`, `radius3`, `perimeter3`, and among others.

`samples` indicates the number of data points there are. In the root node, or the topmost node, we can see that there are 397 samples.

Lastly, we have `value` which indicates a class split [9].

Next, we can see the variable, `gini`, this indicates the probability from a random element to be incorrectly classified [8].

Now, if a random sample data satisfies the condition specified in the node, it will proceed to the next node to the left, otherwise, it will proceed on the node to the right.

[8] V. Richer, "Understanding Decision Trees (once and for all!)," Medium, Mar. 02, 2019. https://towardsdatascience.com/understanding-decision-trees-once-and-for-all-2d891b1be579 (https://towardsdatascience.com/understanding-decision-trees-once-and-for-all-2d891b1be579) (accessed Mar. 05, 2024).

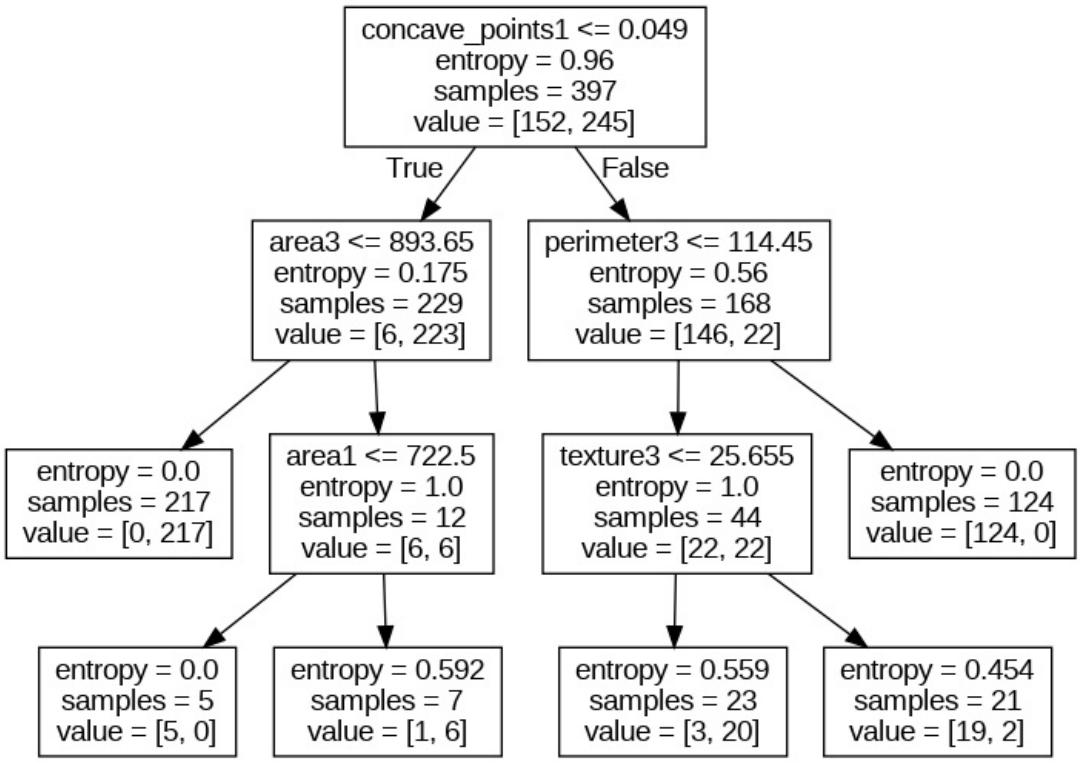
[9] LBoss, "What is the meaning of 'value' in a node in sklearn decisiontree plot_tree," Stack Overflow, 2024. https://stackoverflow.com/questions/65717850/what-is-the-meaning-of-value-in-a-node-in-sklearn-decisiontree-plot-tree (https://stackoverflow.com/questions/65717850/what-is-the-meaning-of-value-in-a-node-in-sklearn-decisiontree-plot-tree) (accessed Mar. 05, 2024).

For Model with Parameters

In [ ]:

```
with open("breastcancer2.dot", 'w') as f:
  f = tree.export_graphviz(dtc2, out_file=f, feature_names=X_train.columns)

!dot -Tpng breastcancer2.dot -o breastcancer2.png

from IPython.display import Image
Image("breastcancer2.png")
```

Out[ ]:



Remarks: By plotting the decision tree, we can now clearly see the difference it takes when we define the parameters of the model. The first image is for the model without parameters, as we can see, the tree expanded exhastively until all the leaves are pure. In the second diagram, the tree is a bit limited when it comes to its levels.

We can also observe that one of the variables inside a leaf in the first diagram is `gini` while `entropy` is in the second diagram.

Random Forest

Identifying the input and target variable

To create a prediction using random forest, we used all the variables except the `ID` and `Diagnosis` for the input variable while the targer variable is the Diagnosis of the patient whether it is benign or malignant.

In [ ]:

```
#creating object

X = namesDF[['radius1','texture1','perimeter1','area1','smoothness1','compactness1', 'concavity1','concave_points
1','symmetry1', 'fractal_dimension1',
    'radius2','texture2','perimeter2','area2','smoothness2','compactness2','concavity2','concave_points2','symmet
ry2','fractal_dimension2',
    'radius3','texture3','perimeter3','area3','smoothness3','compactness3','concavity3','concave_points3','symmet
ry3','fractal_dimension3'
    ]]  #input Variable
y = namesDF[["Diagnosis"]]    #target variable
```

Splitting the Dataset

In this part, we created two data sets by splitting the main dataset for testing and training. The test dataset will be a random 30% of the main dataset while the rest will be used for the training.

In [ ]:

```
#splitting the data to test and train

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3,
                                                    random_state = 0)
```

Creating the model fo Random Forest Classifier

To create a model for random classifier, we imported the RandomForestClassifier from sklearn.ensemble. This library helps us to apply a random forest classfier.

In [ ]:

```
from sklearn.ensemble import RandomForestClassifier
#model creation
model = RandomForestClassifier()
```

Remarks: We opted using the default parameters of the function as it gives a good model performance on an unknown data set as shown in the latter part of the activity.

In [ ]:

```
#fitting the model with input and target variables
model.fit(X_train, y_train)
```

```
<ipython-input-103-5528435b03c6>:2: DataConversionWarning: A column-vector y was passed when a 1d ar
ray was expected. Please change the shape of y to (n_samples,), for example using ravel().
  model.fit(X_train, y_train)
```

Out[ ]:

▾ RandomForestClassifier

RandomForestClassifier()

Evaluating the model

Using accuracy_score

In [ ]:

```
from sklearn.metrics import accuracy_score

#Creating a prediction
y_predTrain = model.predict(X_train)
y_predTest = model.predict(X_test)

accuracyTrain = accuracy_score(y_train, y_predTrain)
accuracyTest = accuracy_score(y_test, y_predTest)

print(f"The accuracy of the model is: {accuracyTrain} \n")
print(f"The accuracy of the model is: {accuracyTest} ")
```

```
The accuracy of the model is: 1.0

The accuracy of the model is: 0.9473684210526315
```

Remarks: The training data set is naturally 100 percent as it was used to create the prediction model. The parameters were not changed thus the maximum depth of the tree was expanded until the end. On the other hand, The accuracy of the model when faced with an unknown value shows a 94.73% accuracy implying that the performance of the model is good.

Using confusion_matrix

In this section, to evaluate the model we used a confusion matrix to determine the true positive and negative and the false positive and negatives on the given testing data set.

In [ ]:

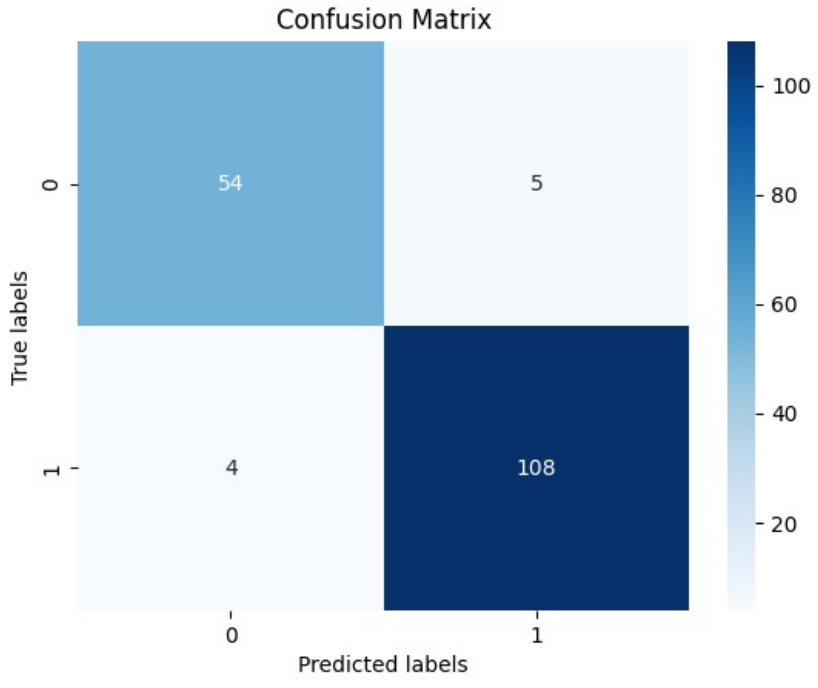
```
from sklearn.metrics import confusion_matrix

y_pred = model.predict(X_test)
conf_m = confusion_matrix(y_test, y_pred)
```

In [ ]:

```
#creating a visualization of confusion matrix
import seaborn as sns
sns.heatmap(conf_m,annot =True,fmt='d', cmap='Blues',
            xticklabels=np.unique(y_test), yticklabels=np.unique(y_test))

#labels
plt.xlabel('Predicted labels')
plt.ylabel('True labels')
plt.title('Confusion Matrix')
```

Out[ ]:

```
Text(0.5, 1.0, 'Confusion Matrix')
```



Remarks: The heat map shows the results of the 171 test values that was tested in the model. It showed that there are 54 true positves, 108 true negatives, while there are 4 and 5 false positives and negatives, respectively. This means that the model is performing good as it has low tendency to produce an error in its prediction.

Using `classification_report`

In this section, we used the `classification_report` function of sklearn.metrics to get the summary of the precision scores of the model.

In [ ]:

```
from sklearn.metrics import classification_report

report = classification_report(y_test, y_pred, target_names=['malignant', 'benign'])
```

In [ ]:

```
print(report)
```

```
              precision    recall  f1-score   support

   malignant       0.93      0.92      0.92        59
      benign       0.96      0.96      0.96       112

    accuracy                           0.95       171
   macro avg       0.94      0.94      0.94       171
weighted avg       0.95      0.95      0.95       171
```

Remarks: As shown in the summary using `classification_report` , the model shown a high level of precision, recall, and f1 score for all the classes and the overall accuracy of it is 95% on an unknown dataset.

Summary, Conclusion, and Lessons Learned

This activity is about performing different algorithms such as linear regression, logistic regression, decision tree, and random forest. The first step in doing the activity involves scouring the internet for dataset to be used. It is important to take note that *there is no such thing as a perfect dataset*. Just like what we did in the first stages, we have to pre-process the dataset we have chosen.

Pre-processing is very important in machine learning. This allows us to scrutinize the data we are feeding the models to ensure that they are suitable for the algorithms.

The process of creating and training the models is similar all the models used in this activity. Another thing that we have learned while doing this activity is the importance of splitting the dataset into training and testing values. Doing so allows us to make more valid evaluation regarding the accuracy of the model created.

Lastly, as data analysts, it is also important to evaluate the models we have created. Libraries are available to aid us in this matter. It is just a matter of researching whether the evaluation metrics values indicate positive or negative model performance.