Machine Learning Tutorial

Machine Learning

Supervised Learning

EDA (Exploratory Data Analysis), K-Nearest Neighbors (KNN), Regression, Cross Validation (CV), ROC Curve, Hyperparameter Tuning, Pre-processing Data

Unsupervised Learning

Kmeans Clustering, Evaluation of Clustering, Standardization, Hierarchy, T-Distributed Stochastic Neighbor Embedding (T-SNE), Principal Component Analysis (PCA)

Import Libraries

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# import warnings
import warnings
# ignore warnings
warnings.filterwarnings("ignore")
from subprocess import check_output
print(check_output(["ls", "../input"]).decode("utf8"))
```

```
column_2C_weka.csv
column_3C_weka.csv
```

In [2]:
```
# read csv (comma separated value) into data
data = pd.read_csv('../input/column_2C_weka.csv')
print(plt.style.available) # look at available plot styles
plt.style.use('ggplot')
```

```
['classic', 'seaborn-poster', 'dark_background', 'seaborn-ticks', 'seaborn-muted', 'seaborn-deep', 'fivethirt
yeight', 'grayscale', 'seaborn-notebook', 'ggplot', 'bmh', 'seaborn-paper', 'fast', 'seaborn-bright', 'seabor
n-colorblind', 'seaborn-pastel', 'seaborn', 'seaborn-talk', 'seaborn-white', 'seaborn-dark', 'seaborn-whitegr
id', 'seaborn-dark-palette', 'Solarize_Light2', '_classic_test', 'seaborn-darkgrid']
```

Machine Learning (ML)

In python there are some ML libraries like sklearn, keras or tensorflow. We will use sklearn.

Supervised Learning

Supervised learning: It involves labeled data. For instance, consider orthopedic patient data labeled as normal or abnormal. There are features (predictor variables) such as pelvic radius or sacral slope, and a target variable indicating normal or abnormal labels. The objective is to use the provided features (input) to predict whether the target variable (output) will be normal or abnormal.

Classification: In classification, the target variable comprises categories like normal or abnormal.

Regression: In regression, the target variable is continuous, as seen in phenomena like stock market data.

If these explanations aren't sufficient, feel free to search for more details online. However, it's important to be cautious about terminology: features can also be referred to as predictor variables, independent variables, columns, or inputs. The target variable might be termed as a response variable, class, dependent variable, output, or result.

Exploratory Data Analysis (EDA)

To refine something in data, you know the crucial step involves data exploration. My initial approach always involves using head() to observe features such as pelvic_incidence, pelvic_tilt numeric, lumbar_lordosis_angle, sacral_slope, pelvic_radius, degree_spondylolisthesis, and the target variable, class.

By default, head() displays the first five rows or samples, giving a snapshot of the dataset's initial structure. This method offers a quick glimpse before delving deeper into analysis or processing.

In [3]:
```
# features and target variable
data.head()
```

Out[3]:

	pelvic_incidence	pelvic_tilt numeric	lumbar_lordosis_angle	sacral_slope	pelvic_radius	degree_spondylolisthesis	class
0	63.027818	22.552586	39.609117	40.475232	98.672917	-0.254400	Abnormal
1	39.056951	10.060991	25.015378	28.995960	114.405425	4.564259	Abnormal
2	68.832021	22.218482	50.092194	46.613539	105.985135	-3.530317	Abnormal
3	69.297008	24.652878	44.311238	44.644130	101.868495	11.211523	Abnormal
4	49.712859	9.652075	28.317406	40.060784	108.168725	7.918501	Abnormal

In [4]:
```
# Check NaN value and length of this data
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 310 entries, 0 to 309
Data columns (total 7 columns):
pelvic_incidence            310 non-null float64
pelvic_tilt numeric         310 non-null float64
lumbar_lordosis_angle       310 non-null float64
sacral_slope                310 non-null float64
pelvic_radius               310 non-null float64
degree_spondylolisthesis    310 non-null float64
class                       310 non-null object
dtypes: float64(6), object(1)
memory usage: 17.0+ KB
```

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

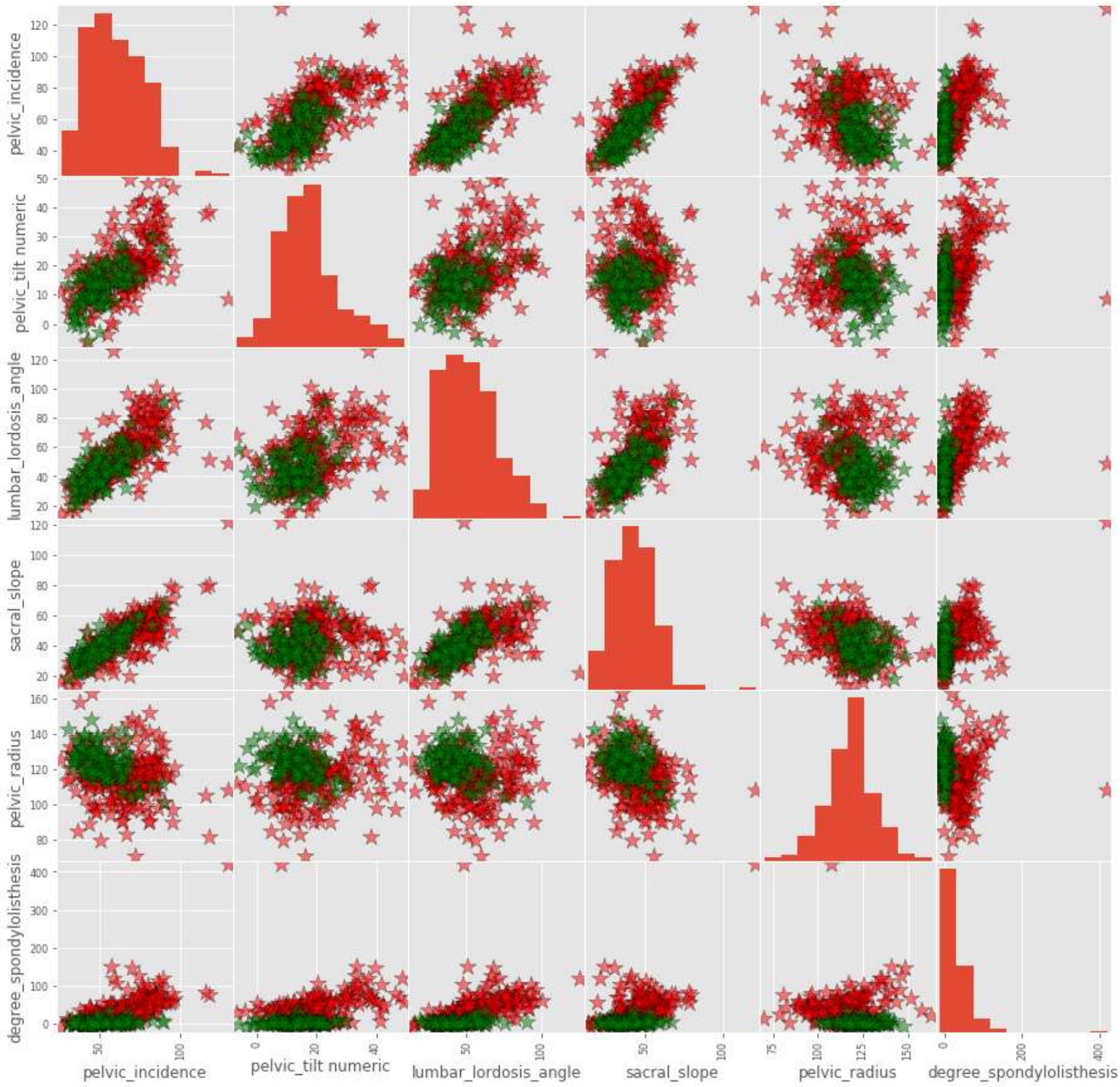
Out[5]:

	pelvic_incidence	pelvic_tilt numeric	lumbar_lordosis_angle	sacral_slope	pelvic_radius	degree_spondylolisthesis
count	310.000000	310.000000	310.000000	310.000000	310.000000	310.000000
mean	60.496653	17.542822	51.930930	42.953831	117.920655	26.296694
std	17.236520	10.008330	18.554064	13.423102	13.317377	37.559027
min	26.147921	-6.554948	14.000000	13.366931	70.082575	-11.058179
25%	46.430294	10.667069	37.000000	33.347122	110.709196	1.603727
50%	58.691038	16.357689	49.562398	42.404912	118.268178	11.767934
75%	72.877696	22.120395	63.000000	52.695888	125.467674	41.287352
max	129.834041	49.431864	125.742385	121.429566	163.071041	418.543082

pd.plotting.scatter_matrix:

- green: *normal* and red: *abnormal*
- c: color
- figsize: figure size
- diagonal: histogram of each features
- alpha: opacity
- s: size of marker
- marker: marker type

In [6]:
```
color_list = ['red' if i=='Abnormal' else 'green' for i in data.loc[:,'class']]
pd.plotting.scatter_matrix(data.loc[:, data.columns != 'class'],
                                       c=color_list,
                                       figsize= [15,15],
                                       diagonal='hist',
                                       alpha=0.5,
                                       s = 200,
                                       marker = '*',
                                       edgecolor= "black")
plt.show()
```



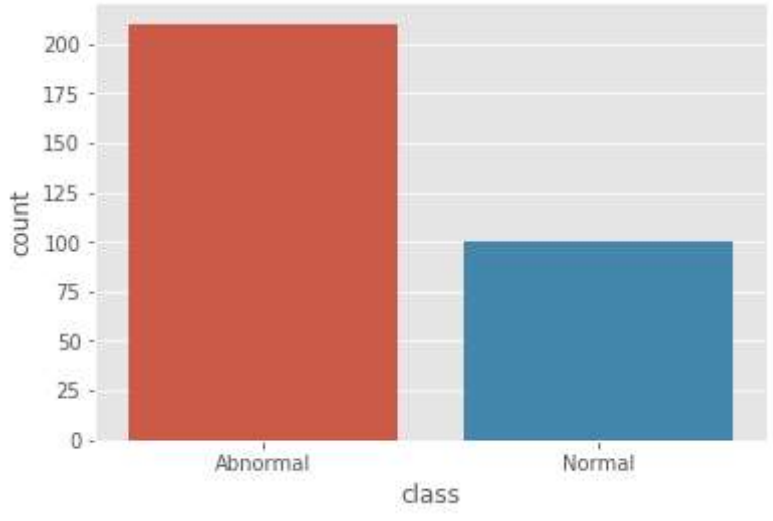
Check how many normal(green) and abnormal(red) classes are there.

- Searborn library has *countplot()* that counts number of classes
- Also you can print it with *value_counts()* method

This data looks like balanced. Actually there is no definiton or numeric value of balanced data but this data is balanced enough for us.
Now lets learn first classification method KNN

In [7]:
```
sns.countplot(x="class", data=data)
data.loc[:,'class'].value_counts()
```

Out[7]:
```
Abnormal    210
Normal      100
Name: class, dtype: int64
```



K-Nearest Neighbors (KNN)

- KNN: Look at the K closest labeled data points
- Classification method.
- First we need to train our data. Train = fit
- fit(): fits the data, train the data.
- predict(): predicts the data
- x: features
- y: target variables(normal, abnormal)
- n_neighbors: K. In this example it is 3. it means that Look at the 3 closest labeled data points

```python
In [8]: # KNN
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors = 3)
x,y = data.loc[:,data.columns != 'class'], data.loc[:,'class']
knn.fit(x,y)
prediction = knn.predict(x)
print('Prediction: {}'.format(prediction))
```

```
Prediction: ['Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Normal' 'Normal'
 'Abnormal' 'Normal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal'
 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal'
 'Abnormal' 'Normal' 'Normal' 'Normal' 'Abnormal' 'Abnormal' 'Abnormal'
 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Normal' 'Abnormal' 'Normal'
 'Normal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal'
 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal'
 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal'
 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal'
 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal'
 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Normal'
 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal'
 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal'
 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal'
 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal'
 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal'
 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal'
 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal'
 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal'
 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal'
 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal'
 'Abnormal' 'Abnormal' 'Normal' 'Abnormal' 'Abnormal' 'Abnormal'
 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal'
 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal'
 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal'
 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal'
 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal'
 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal'
 'Normal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal'
 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal'
 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal'
 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal'
 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal'
 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal'
 'Abnormal' 'Abnormal' 'Abnormal' 'Abnormal' 'Normal' 'Normal' 'Normal'
 'Normal' 'Normal' 'Normal' 'Normal' 'Normal' 'Normal' 'Normal' 'Normal'
 'Abnormal' 'Normal' 'Normal' 'Normal' 'Normal' 'Normal' 'Normal' 'Normal'
 'Normal' 'Normal' 'Normal' 'Normal' 'Abnormal' 'Normal' 'Normal' 'Normal'
 'Normal' 'Normal' 'Normal' 'Abnormal' 'Normal' 'Normal' 'Abnormal'
 'Normal' 'Normal' 'Normal' 'Normal' 'Abnormal' 'Normal' 'Normal' 'Normal'
 'Normal' 'Normal' 'Abnormal' 'Normal' 'Normal' 'Normal' 'Normal' 'Normal'
 'Normal' 'Normal' 'Normal' 'Normal' 'Normal' 'Normal' 'Normal' 'Normal'
 'Normal' 'Normal' 'Normal' 'Normal' 'Normal' 'Normal' 'Normal' 'Normal'
 'Normal' 'Normal' 'Normal' 'Normal' 'Normal' 'Normal' 'Normal' 'Normal'
 'Normal' 'Normal' 'Normal' 'Normal' 'Normal' 'Normal' 'Normal' 'Normal'
 'Abnormal' 'Normal' 'Normal' 'Normal' 'Abnormal' 'Normal' 'Normal'
 'Normal' 'Normal' 'Normal' 'Normal' 'Normal' 'Abnormal' 'Normal'
 'Abnormal' 'Normal' 'Normal']
```

Fit the data and predict it using KNN. Now, let's address whether our predictions are correct and whether accuracy is the best metric to evaluate our results.

When it comes to measuring model performance, accuracy, which represents the fraction of correct predictions, is a commonly used metric. While we'll utilize it, there's another issue to consider.

You might notice that I trained the data using x (features) and then predicted using x (features) again.

Hence, we need to divide our data into train and test sets.

Train: Utilize the train set by fitting the model.

Test: Make predictions on the test set.

When employing train and test sets, the fitted data and tested data are entirely distinct.

To achieve this split, use train_test_split(x, y, test_size=0.3, random_state=1).

x: features

y: target variables (normal, abnormal)

test_size: denotes the percentage of the test size. For instance, test_size=0.3 implies a test size of 30% and a train size of 70%.

random_state: sets a seed. Using the same number here ensures that train_test_split() generates the exact same split each time. After splitting:

fit(x_train, y_train): Fit the model on the train sets.

score(x_test, y_test): Predict and determine the accuracy on the test sets.

In [9]:
```
# train test split
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 = 1)
knn = KNeighborsClassifier(n_neighbors = 3)
x,y = data.loc[:,data.columns != 'class'], data.loc[:,'class']
knn.fit(x_train,y_train)
prediction = knn.predict(x_test)
#print('Prediction: {}'.format(prediction))
print('With KNN (K=3) accuracy is: ',knn.score(x_test,y_test)) # accuracy
```

```
With KNN (K=3) accuracy is:  0.8602150537634409
```

The accuracy stands at 86%. Is this a good result? I'm not entirely sure yet; we'll assess that at the end of the tutorial.

Now, the pivotal question: why did we choose K = 3, or what value should we choose for K? The answer lies in model complexity.

Model complexity:

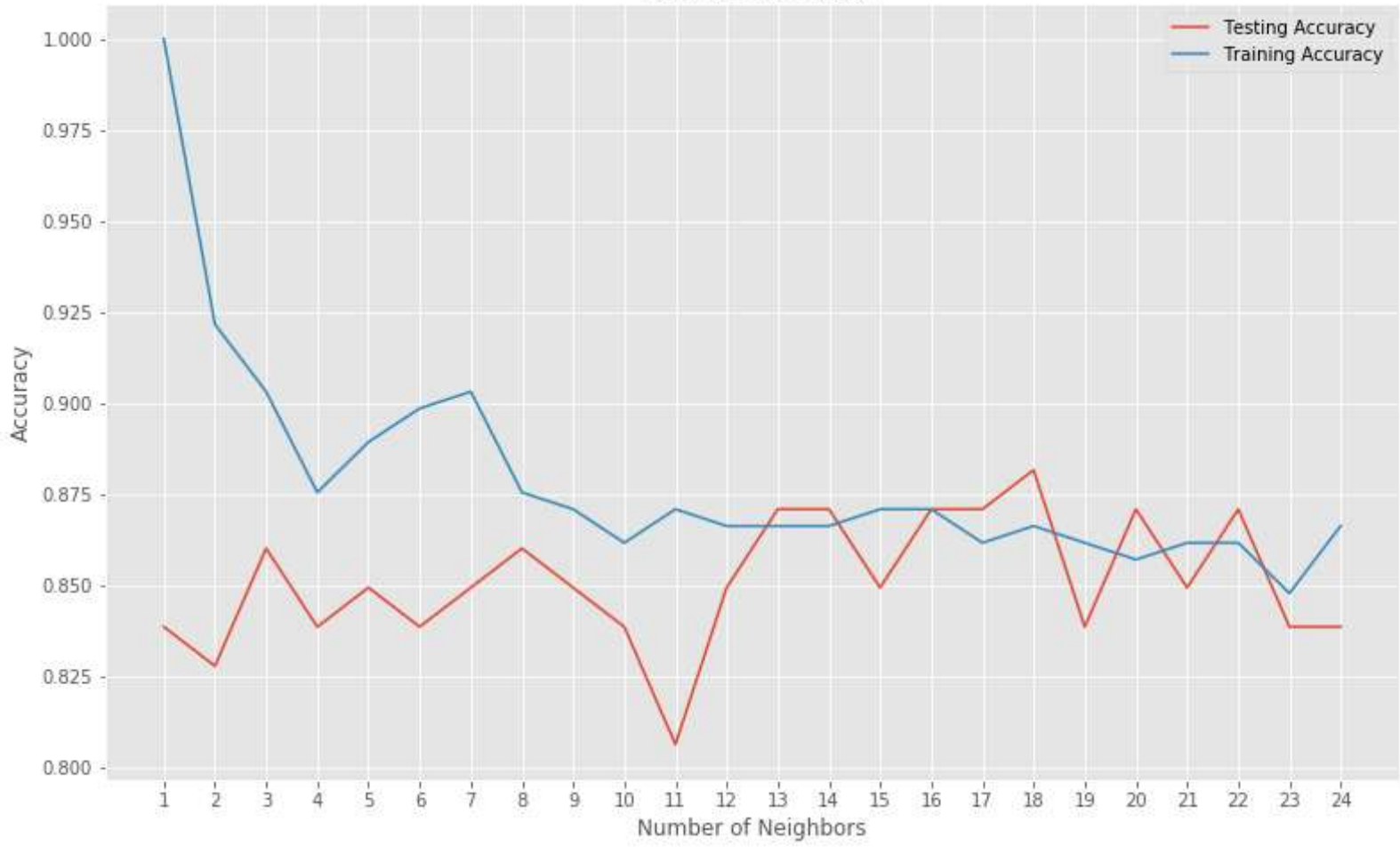
K represents a general parameter termed a hyperparameter. For now, understand that K is a hyperparameter, and we aim to select the value that yields the best performance.

According to literature, a small K leads to a complex model, potentially resulting in overfitting. This scenario involves the model memorizing the training sets and struggling to predict the test set accurately.

Conversely, a large K leads to a less complex model, potentially causing underfitting.

I experimented by ranging K values from 1 to 25 (excluding 25) and calculated accuracy for each K value. In the plot, when K is 1, the model memorizes the training sets (overfitting), resulting in poor accuracy on the test set. Similarly, with K at 18, the model tends to underfit, again resulting in inadequate accuracy. However, when K is 18 (showing the best performance), the accuracy reaches its peak at almost 88%.

In [10]:
```
# Model complexity
neig = np.arange(1, 25)
train_accuracy = []
test_accuracy = []
# Loop over different values of k
for i, k in enumerate(neig):
    # k from 1 to 25(exclude)
    knn = KNeighborsClassifier(n_neighbors=k)
    # Fit with knn
    knn.fit(x_train,y_train)
    #train accuracy
    train_accuracy.append(knn.score(x_train, y_train))
    # test accuracy
    test_accuracy.append(knn.score(x_test, y_test))

# Plot
plt.figure(figsize=[13,8])
plt.plot(neig, test_accuracy, label = 'Testing Accuracy')
plt.plot(neig, train_accuracy, label = 'Training Accuracy')
plt.legend()
plt.title('-value VS Accuracy')
plt.xlabel('Number of Neighbors')
plt.ylabel('Accuracy')
plt.xticks(neig)
plt.savefig('graph.png')
plt.show()
print("Best accuracy is {} with K = {}".format(np.max(test_accuracy),1+test_accuracy.index(np.max(test_accurac
```




```
Best accuracy is 0.8817204301075269 with K = 18
```

Regression

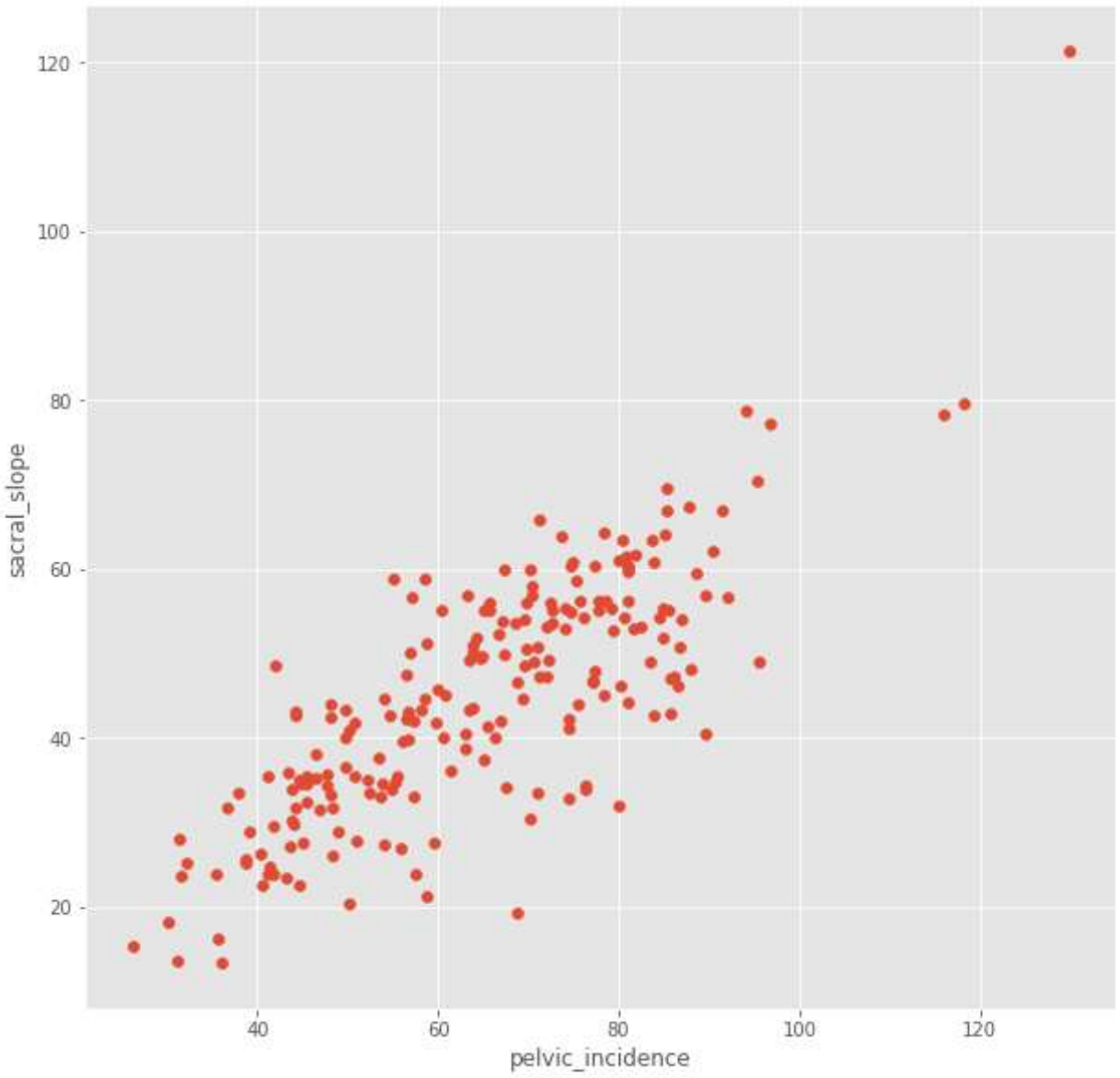
Supervised learning

We'll delve into linear and logistic regressions. However, the orthopedic patient data isn't suitable for regression analysis, so I'm utilizing only two features: sacral_slope and pelvic_incidence for abnormalities.

I'm considering pelvic_incidence as the feature and sacral_slope as the target variable. Let's visualize this relationship using a scatter plot to gain better insights.

The function reshape(-1,1) is necessary. If not used, the shape of x or y becomes (210,), which cannot be employed in sklearn. By using reshape(-1,1), the shape of x or y becomes (210, 1), making it compatible with sklearn.

In [11]:
```
# create data1 that includes pelvic_incidence that is feature and sacral_slope that is target variable
data1 = data[data['class'] =='Abnormal']
x = np.array(data1.loc[:,'pelvic_incidence']).reshape(-1,1)
y = np.array(data1.loc[:,'sacral_slope']).reshape(-1,1)
# Scatter
plt.figure(figsize=[10,10])
plt.scatter(x=x,y=y)
plt.xlabel('pelvic_incidence')
plt.ylabel('sacral_slope')
plt.show()
```



We're now equipped with our data to conduct regression. In regression problems, the target value represents a continuously varying variable, such as the price of a house or sacral_slope. Our aim is to fit a line to these data points.

Linear regression:

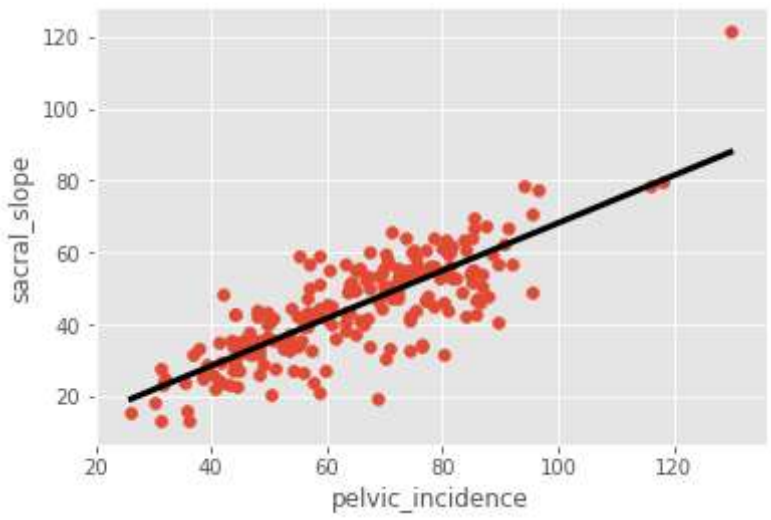
The equation y = ax + b is fundamental, where y represents the target, x denotes the feature, and a signifies the parameter of the model. We determine the model's parameter (a) based on minimizing the error function, known as the loss function.

For linear regression, we employ Ordinary Least Squares (OLS) as the loss function. OLS involves summing all residuals, considering that positive and negative residuals may offset each other. Hence, we square the residuals and sum them, hence the term OLS.

When evaluating the model, the 'Score' utilizes the R^2 method, calculated as ((y_pred - y_mean)^2)/(y_actual - y_mean)^2.

In [12]:
```
# LinearRegression
from sklearn.linear_model import LinearRegression
reg = LinearRegression()
# Predict space
predict_space = np.linspace(min(x), max(x)).reshape(-1,1)
# Fit
reg.fit(x,y)
# Predict
predicted = reg.predict(predict_space)
# R^2
print('R^2 score: ',reg.score(x, y))
# Plot regression line and scatter
plt.plot(predict_space, predicted, color='black', linewidth=3)
plt.scatter(x=x,y=y)
plt.xlabel('pelvic_incidence')
plt.ylabel('sacral_slope')
plt.show()
```

```
R^2 score:  0.6458410481075871
```



An R^2 score of 0.645 indicates that the model explains approximately 64.5% of the variance in the dependent variable. This score measures how well the regression predictions approximate the actual data points. A higher R^2 score closer to 1 suggests that the model better fits the data.

Cross Validation (CV)

In the KNN method, using the train-test split with a fixed random_state ensures consistent splits each time. However, without specifying random_state, the data splits differently, affecting the accuracy. Consequently, the model's performance relies on the train-test split.

Consider this: if you split, fit, and predict the data five times, resulting in accuracies of 0.89, 0.9, 0.91, 0.92, and 0.93, respectively, which accuracy do you rely on? Additionally, predicting the accuracy for the sixth split remains uncertain.

The solution? Employing cross-validation (CV) helps determine a more acceptable accuracy estimate.

Cross Validation (CV):

CV aids in evaluating the model's performance by iteratively splitting the data into multiple train-test sets. This method provides a more reliable and stable estimate of the model's accuracy, minimizing the dependency on a single train-test split.

- K folds = K fold CV.
- Look at this image it defines better than me :)
- When K is increase, computationally cost is increase
- cross_val_score(reg,x,y,cv=5): use reg(linear regression) with x and y that we define at above and K is 5. It means 5 times(split, train,predict)

In [13]:
```
# CV
from sklearn.model_selection import cross_val_score
reg = LinearRegression()
k = 5
cv_result = cross_val_score(reg,x,y,cv=k) # uses R^2 as score
print('CV Scores: ',cv_result)
print('CV scores average: ',np.sum(cv_result)/k)
```

```
CV Scores:  [0.32924233 0.61683991 0.53117056 0.1954798  0.29299864]
CV scores average:  0.39314625028848676
```

Regularized Regression

In linear regression, parameter selection involves minimizing the loss function. When linear regression identifies a feature as significant, it assigns a higher coefficient to that feature. However, this emphasis on specific features can lead to overfitting, akin to memorization in KNN. To prevent overfitting, regularization is employed, penalizing large coefficients.

- Ridge regression: First regularization technique. Also it is called L2 regularization.
  - Ridge regression lost fuction = OLS + alpha * sum(parameter^2)
  - Alpha is a parameter crucial for fitting and predicting. Choosing alpha resembles selecting K in KNN. As you've grasped, alpha is a hyperparameter pivotal for achieving the best accuracy and managing model complexity. This process is commonly referred to as hyperparameter tuning.

    - What if alpha is zero? lost function = OLS so that is linear rigression
    - If alpha is small that can cause overfitting
    - If alpha is big that can cause underfitting. But do not ask what is small and big. These can be change from problem to problem.
- Lasso regression: Second regularization technique. Also it is called L1 regularization.
    - Lasso regression lost fuction = OLS + alpha * sum(absolute_value(parameter))
    - It can be used to select important features within the data because Lasso regression chooses features whose values are not shrunk to zero.
    - In order to choose feature, I add new features in our regression data

Linear vs Ridge vs Lasso

First impression: Linear

Feature Selection: 1.Lasso 2.Ridge

Regression model: 1.Ridge 2.Lasso 3.Linear

In [14]:
```
# Ridge
from sklearn.linear_model import Ridge
x_train,x_test,y_train,y_test = train_test_split(x,y,random_state = 2, test_size = 0.3)
ridge = Ridge(alpha = 0.1, normalize = True)
ridge.fit(x_train,y_train)
ridge_predict = ridge.predict(x_test)
print('Ridge score: ',ridge.score(x_test,y_test))
```

```
Ridge score:  0.5608287918841997
```

In [15]:
```
# Lasso
from sklearn.linear_model import Lasso
x = np.array(data1.loc[:,['pelvic_incidence','pelvic_tilt numeric','lumbar_lordosis_angle','pelvic_radius']])
x_train,x_test,y_train,y_test = train_test_split(x,y,random_state = 3, test_size = 0.3)
lasso = Lasso(alpha = 0.1, normalize = True)
lasso.fit(x_train,y_train)
ridge_predict = lasso.predict(x_test)
print('Lasso score: ',lasso.score(x_test,y_test))
print('Lasso coefficients: ',lasso.coef_)
```

```
Lasso score:  0.9640334804327547
Lasso coefficients:  [ 0.82498243 -0.7209057   0.         -0.        ]
```

As you can observe, pelvic_incidence and pelvic_tilt numeric are considered important features, whereas the others are deemed less significant.

Now, let's delve into accuracy. Is it sufficient as a model selection metric? Consider a dataset comprising 95% normal and 5% abnormal samples, where our model utilizes accuracy for measurement. In this scenario, if our model predicts 100% normal for all samples, achieving a 95% accuracy, it misclassifies all abnormal samples. Thus, in cases of imbalanced data, relying solely on accuracy becomes inadequate. Instead, employing a confusion matrix serves as a more appropriate model measurement matrix.

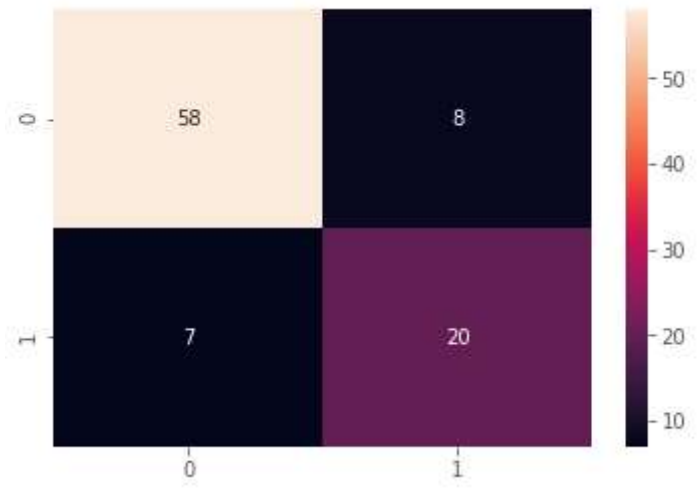
When utilizing the confusion matrix, let's employ the Random Forest classifier to diversify classification methods.

- tp = true positive(20), fp = false positive(7), fn = false negative(8), tn = true negative(58)
- tp = Prediction is positive(normal) and actual is positive(normal).
- fp = Prediction is positive(normal) and actual is negative(abnormal).
- fn = Prediction is negative(abnormal) and actual is positive(normal).
- tn = Prediction is negative(abnormal) and actual is negative(abnormal)
- precision = tp / (tp+fp)
- recall = tp / (tp+fn)
- f1 = 2 * precision * recall / (precision + recall)

In [16]:
```
# Confusion matrix with random forest
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.ensemble import RandomForestClassifier
x,y = data.loc[:,data.columns != 'class'], data.loc[:,'class']
x_train,x_test,y_train,y_test = train_test_split(x,y,test_size = 0.3,random_state = 1)
rf = RandomForestClassifier(random_state = 4)
rf.fit(x_train,y_train)
y_pred = rf.predict(x_test)
cm = confusion_matrix(y_test,y_pred)
print('Confusion matrix: \n',cm)
print('Classification report: \n',classification_report(y_test,y_pred))
```

```
Confusion matrix:
 [[58  8]
 [ 7 20]]
Classification report:
              precision    recall  f1-score   support

   Abnormal       0.89      0.88      0.89        66
     Normal       0.71      0.74      0.73        27

avg / total       0.84      0.84      0.84        93
```

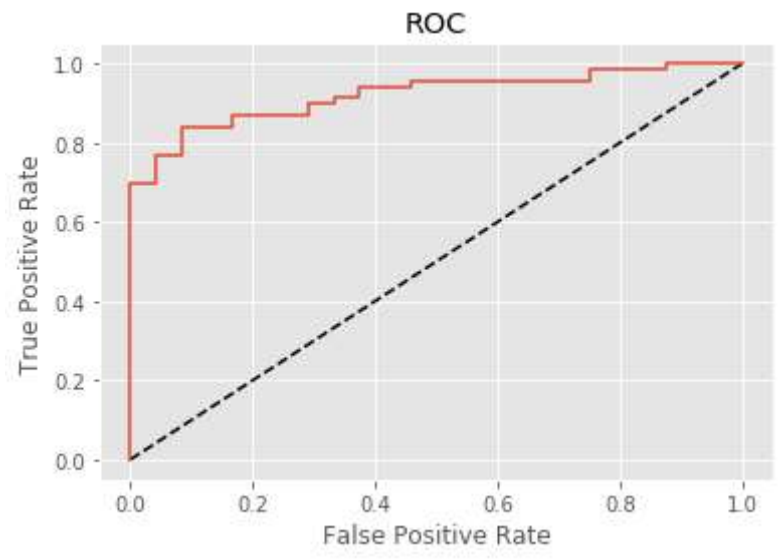
In [17]:
```
# visualize with seaborn library
sns.heatmap(cm,annot=True,fmt="d")
plt.show()
```



ROC Curve with Logistic Regression

- logistic regression output is probabilities
- If probability is higher than 0.5 data is labeled 1(abnormal) else 0(normal)
- By default logistic regression threshold is 0.5
- ROC is receiver operationg characteristic. In this curve x axis is false positive rate and y axis is true positive rate
- If the curve in plot is closer to left-top corner, test is more accurate.
- Roc curve score is auc that is computation area under the curve from prediction scores
- We want auc to closer 1
- fpr = False Positive Rate
- tpr = True Positive Rate

In [18]:
```
# ROC Curve with logistic regression
from sklearn.metrics import roc_curve
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix, classification_report
# abnormal = 1 and normal = 0
data['class_binary'] = [1 if i == 'Abnormal' else 0 for i in data.loc[:,'class']]
x,y = data.loc[:,(data.columns != 'class') & (data.columns != 'class_binary')], data.loc[:,'class_binary']
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.3, random_state=42)
logreg = LogisticRegression()
logreg.fit(x_train,y_train)
y_pred_prob = logreg.predict_proba(x_test)[:,1]
fpr, tpr, thresholds = roc_curve(y_test, y_pred_prob)
# Plot ROC curve
plt.plot([0, 1], [0, 1], 'k--')
plt.plot(fpr, tpr)
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC')
plt.show()
```



Hyperparameter Tuning

As I mention at KNN there are hyperparameters that are need to be tuned

- For example:
  - k at KNN
  - alpha at Ridge and Lasso
  - Random forest parameters like max_depth
  - linear regression parameters(coefficients)
- Hyperparameter tuning:
  - try all of combinations of different parameters
  - fit all of them
  - measure prediction performance
  - see how well each performs
  - finally choose best hyperparameters

- We only need is one line code that is GridSearchCV
  - grid: K is from 1 to 50(exclude)
  - GridSearchCV takes knn and grid and makes grid search. It means combination of all hyperparameters. Here it is k.

In [19]:
```
# grid search cross validation with 1 hyperparameter
from sklearn.model_selection import GridSearchCV
grid = {'n_neighbors': np.arange(1,50)}
knn = KNeighborsClassifier()
knn_cv = GridSearchCV(knn, grid, cv=3) # GridSearchCV
knn_cv.fit(x,y)# Fit

# Print hyperparameter
print("Tuned hyperparameter k: {}".format(knn_cv.best_params_))
print("Best score: {}".format(knn_cv.best_score_))
```

```
Tuned hyperparameter k: {'n_neighbors': 4}
Best score: 0.7548387096774194

/opt/conda/lib/python3.6/site-packages/sklearn/model_selection/_search.py:735: DeprecationWarning: The defaul
t of the `iid` parameter will change from True to False in version 0.22 and will be removed in 0.24. This wil
l change numeric results when test-set sizes are unequal.
  DeprecationWarning)
```

Other grid search example with 2 hyperparameter

- First hyperparameter is C:logistic regression regularization parameter
  - If C is high: overfit
  - If C is low: underfit
- Second hyperparameter is penalty(lost function): l1 (Lasso) or l2(Ridge) as we learnt at linear regression part.

In [20]:
```
# grid search cross validation with 2 hyperparameter
# 1. hyperparameter is C:logistic regression regularization parameter
# 2. penalty l1 or l2
# Hyperparameter grid
param_grid = {'C': np.logspace(-3, 3, 7), 'penalty': ['l1', 'l2']}
x_train, x_test, y_train, y_test = train_test_split(x,y,test_size = 0.3,random_state = 12)
logreg = LogisticRegression()
logreg_cv = GridSearchCV(logreg,param_grid,cv=3)
logreg_cv.fit(x_train,y_train)

# Print the optimal parameters and best score
print("Tuned hyperparameters : {}".format(logreg_cv.best_params_))
print("Best Accuracy: {}".format(logreg_cv.best_score_))
```

```
Tuned hyperparameters : {'C': 100.0, 'penalty': 'l2'}
Best Accuracy: 0.8525345622119815

/opt/conda/lib/python3.6/site-packages/sklearn/model_selection/_search.py:735: DeprecationWarning: The defaul
t of the `iid` parameter will change from True to False in version 0.22 and will be removed in 0.24. This wil
l change numeric results when test-set sizes are unequal.
  DeprecationWarning)
```

Pre-processing Data

In real-life datasets, categorical variables or objects often exist. To utilize them in sklearn, it's necessary to encode these variables into numerical data.

In our dataset, the 'class' variable contains 'abnormal' and 'normal' labels. We've converted them into numeric values previously (in the logistic regression section) by creating two distinct features: 'class_Abnormal' and 'class_Normal.' However, it's essential to drop one of these columns since they're duplicates.

In [21]:
```
# Load data
data = pd.read_csv('../input/column_2C_weka.csv')
# get_dummies
df = pd.get_dummies(data)
df.head(10)
```

Out[21]:

	pelvic_incidence	pelvic_tilt numeric	lumbar_lordosis_angle	sacral_slope	pelvic_radius	degree_spondylolisthesis	class_Abnormal	class_Norn
0	63.027818	22.552586	39.609117	40.475232	98.672917	-0.254400	1	
1	39.056951	10.060991	25.015378	28.995960	114.405425	4.564259	1	
2	68.832021	22.218482	50.092194	46.613539	105.985135	-3.530317	1	
3	69.297008	24.652878	44.311238	44.644130	101.868495	11.211523	1	
4	49.712859	9.652075	28.317406	40.060784	108.168725	7.918501	1	
5	40.250200	13.921907	25.124950	26.328293	130.327871	2.230652	1	
6	53.432928	15.864336	37.165934	37.568592	120.567523	5.988551	1	
7	45.366754	10.755611	29.038349	34.611142	117.270068	-10.675871	1	
8	43.790190	13.533753	42.690814	30.256437	125.002893	13.289018	1	
9	36.686353	5.010884	41.948751	31.675469	84.241415	0.664437	1	

In [22]:
```
# drop one of the feature
df.drop("class_Normal",axis = 1, inplace = True)
df.head(10)
# instead of two steps we can make it with one step pd.get_dummies(data,drop_first = True)
```

Out[22]:

	pelvic_incidence	pelvic_tilt numeric	lumbar_lordosis_angle	sacral_slope	pelvic_radius	degree_spondylolisthesis	class_Abnormal
0	63.027818	22.552586	39.609117	40.475232	98.672917	-0.254400	1
1	39.056951	10.060991	25.015378	28.995960	114.405425	4.564259	1
2	68.832021	22.218482	50.092194	46.613539	105.985135	-3.530317	1
3	69.297008	24.652878	44.311238	44.644130	101.868495	11.211523	1
4	49.712859	9.652075	28.317406	40.060784	108.168725	7.918501	1
5	40.250200	13.921907	25.124950	26.328293	130.327871	2.230652	1
6	53.432928	15.864336	37.165934	37.568592	120.567523	5.988551	1
7	45.366754	10.755611	29.038349	34.611142	117.270068	-10.675871	1
8	43.790190	13.533753	42.690814	30.256437	125.002893	13.289018	1
9	36.686353	5.010884	41.948751	31.675469	84.241415	0.664437	1

In [23]:
```
# SVM, pre-process and pipeline
from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
steps = [('scalar', StandardScaler()),
         ('SVM', SVC())]
pipeline = Pipeline(steps)
parameters = {'SVM__C':[1, 10, 100],
              'SVM__gamma':[0.1, 0.01]}
x_train, x_test, y_train, y_test = train_test_split(x,y,test_size=0.2,random_state = 1)
cv = GridSearchCV(pipeline,param_grid=parameters,cv=3)
cv.fit(x_train,y_train)

y_pred = cv.predict(x_test)

print("Accuracy: {}".format(cv.score(x_test, y_test)))
print("Tuned Model Parameters: {}".format(cv.best_params_))
```

```
Accuracy: 0.8548387096774194
Tuned Model Parameters: {'SVM__C': 100, 'SVM__gamma': 0.01}

/opt/conda/lib/python3.6/site-packages/sklearn/model_selection/_search.py:735: DeprecationWarning: The defaul
t of the `iid` parameter will change from True to False in version 0.22 and will be removed in 0.24. This wil
l change numeric results when test-set sizes are unequal.
  DeprecationWarning)
```

Unsupervised Learning

Unsupervised learning involves working with unlabeled data to uncover hidden patterns within it. For instance, consider orthopedic patient data without any labels. In this dataset, distinguishing between normal and abnormal orthopedic patients is unknown.

Typically, orthopedic patient data is labeled (supervised), containing target variables indicating patient status. To transition to unsupervised learning, we must remove these target variables. For visualization purposes, let's focus on two specific features: pelvic_radius and degree_spondylolisthesis.

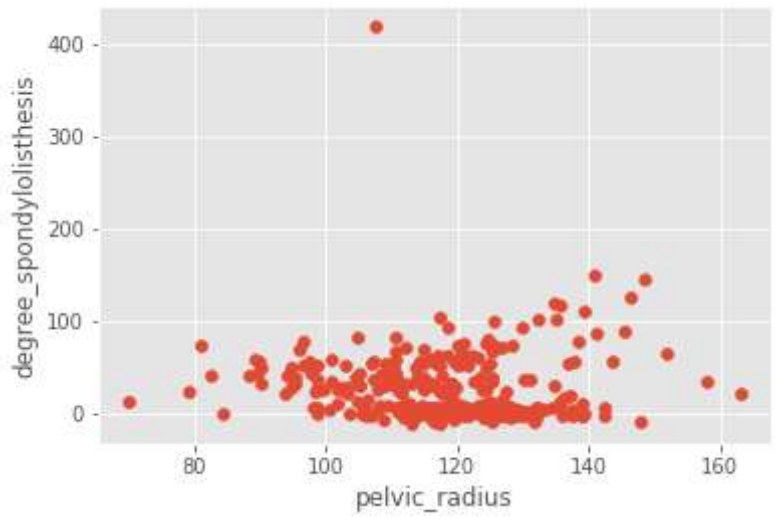
KMEANS

Let's explore our first unsupervised method: KMeans Clustering.

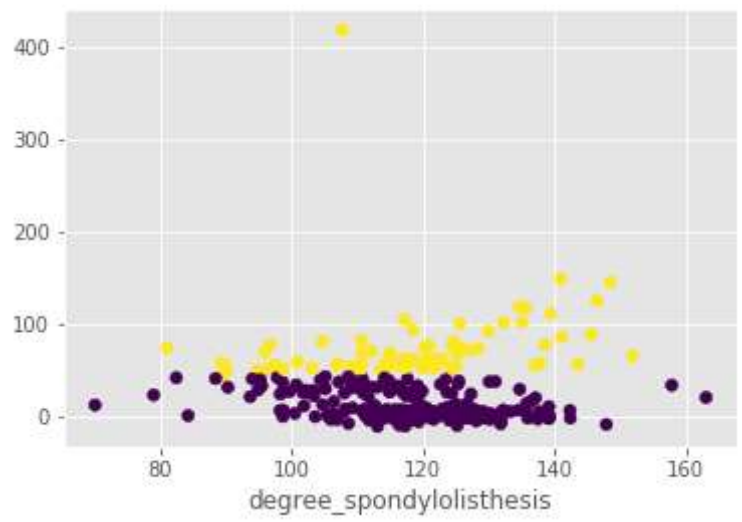
KMeans Clustering: This algorithm operates iteratively, assigning each data point to one of K groups determined by the provided features. The clustering process relies on the similarity of features among data points.

When implementing KMeans(n_clusters = 2), specifying n_clusters = 2 indicates the creation of 2 clusters.

In [24]:
```
# As you can see there is no labels in data
data = pd.read_csv('../input/column_2C_weka.csv')
plt.scatter(data['pelvic_radius'],data['degree_spondylolisthesis'])
plt.xlabel('pelvic_radius')
plt.ylabel('degree_spondylolisthesis')
plt.show()
```



In [25]:
```
# KMeans Clustering
data2 = data.loc[:,['degree_spondylolisthesis','pelvic_radius']]
from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters = 2)
kmeans.fit(data2)
labels = kmeans.predict(data2)
plt.scatter(data['pelvic_radius'],data['degree_spondylolisthesis'],c = labels)
plt.xlabel('pelvic_radius')
plt.xlabel('degree_spondylolisthesis')
plt.show()
```



Evaluation of Clustering

We cluster data in two groups. In order to evaluate clustering we will use cross tabulation table.

- There are two clusters that are *0* and *1*
- First class *0* includes 138 abnormal and 100 normal patients
- Second class *1* includes 72 abnormal and 0 normal patiens
- The majority of two clusters are abnormal patients.

In [26]:
```
# cross tabulation table
df = pd.DataFrame({'labels':labels,"class":data['class']})
ct = pd.crosstab(df['labels'],df['class'])
print(ct)
```

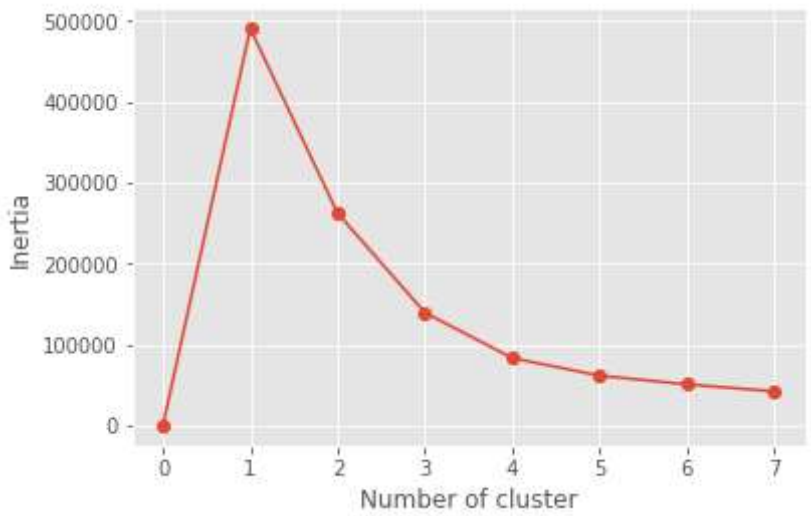
```
class   Abnormal  Normal
labels
0            138     100
1             72       0
```

The new question arises when the number of classes in the data is unknown. This scenario resembles the hyperparameter selection in KNN or regressions.

Inertia measures how spread out the clusters are in terms of distance from each sample. Lower inertia suggests more compact clusters.

Determining the best number of clusters involves a trade-off between achieving low inertia and avoiding an excessive number of clusters. This trade-off can often be identified using the 'elbow method.

In [27]:
```
# inertia
inertia_list = np.empty(8)
for i in range(1,8):
    kmeans = KMeans(n_clusters=i)
    kmeans.fit(data2)
    inertia_list[i] = kmeans.inertia_
plt.plot(range(0,8),inertia_list,'-o')
plt.xlabel('Number of cluster')
plt.ylabel('Inertia')
plt.show()
```



Standardization

- Standardizaton is important for both supervised and unsupervised learning
- Do not forget standardization as pre-processing
- As we already have visualized data so you got the idea. Now we can use all features for clustering.
- We can use pipeline like supervised learning.

In [28]:
```
data = pd.read_csv('../input/column_2C_weka.csv')
data3 = data.drop('class',axis = 1)
```

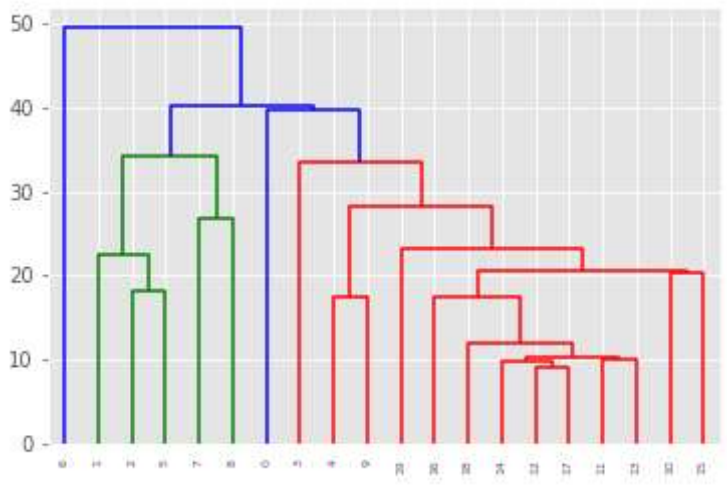
In [29]:
```
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
scalar = StandardScaler()
kmeans = KMeans(n_clusters = 2)
pipe = make_pipeline(scalar,kmeans)
pipe.fit(data3)
labels = pipe.predict(data3)
df = pd.DataFrame({'labels':labels,"class":data['class']})
ct = pd.crosstab(df['labels'],df['class'])
print(ct)
```

```
class   Abnormal  Normal
labels
0            116      10
1             94      90
```

Hierarchy

- vertical lines are clusters
- height on dendrogram: distance between merging cluster
- method= 'single' : closest points of clusters

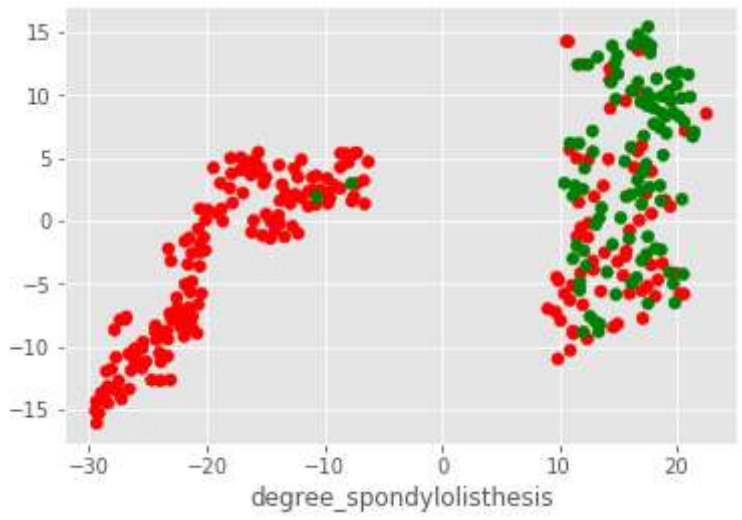
In [30]:
```
from scipy.cluster.hierarchy import linkage,dendrogram

merg = linkage(data3.iloc[200:220,:],method = 'single')
dendrogram(merg, leaf_rotation = 90, leaf_font_size = 6)
plt.show()
```



T - Distributed Stochastic Neighbor Embedding (T - SNE)

- learning rate: 50-200 in normal
- fit_transform: it is both fit and transform. t-sne has only have fit_transform
- Varieties have same position relative to one another

In [31]:
```
from sklearn.manifold import TSNE
model = TSNE(learning_rate=100)
transformed = model.fit_transform(data2)
x = transformed[:,0]
y = transformed[:,1]
plt.scatter(x,y,c = color_list )
plt.xlabel('pelvic_radius')
plt.xlabel('degree_spondylolisthesis')
plt.show()
```



Principal Component Analysis (PCA)

- Fundemental dimension reduction technique
- first step is decorrelation:
  - rotates data samples to be aligned with axes
  - shifts data asmples so they have mean zero
  - no information lost
  - fit() : learn how to shift samples
  - transform(): apply the learned transformation. It can also be applies test data
- Resulting PCA features are not linearly correlated
- Principle components: directions of variance

In [32]:
```
# PCA
from sklearn.decomposition import PCA
model = PCA()
model.fit(data3)
transformed = model.transform(data3)
print('Principle components: ',model.components_)
```

```
Principle components:  [[ 3.23645647e-01  1.13192291e-01  3.03674740e-01  2.10453357e-01
  -2.99598300e-02  8.63153779e-01]
 [-4.76634849e-01 -9.85632787e-02 -5.32783979e-01 -3.78071570e-01
   3.21809199e-01  4.82438036e-01]
 [-1.54481282e-03 -2.64657410e-01 -4.96541893e-01  2.63112598e-01
  -7.74612852e-01  1.18940778e-01]
 [ 3.73677251e-01  7.54113757e-01 -3.39411757e-01 -3.80436506e-01
  -1.75106042e-01 -3.29143086e-02]
 [-4.41703869e-01  7.35414748e-02  5.12024113e-01 -5.15245344e-01
  -5.14639730e-01  8.35992525e-02]
 [ 5.77350269e-01 -5.77350269e-01 -1.08931753e-11 -5.77350269e-01
  -3.59057228e-12  3.06721315e-12]]
```

In [33]:
```
# PCA variance
scaler = StandardScaler()
pca = PCA()
pipeline = make_pipeline(scaler,pca)
pipeline.fit(data3)

plt.bar(range(pca.n_components_), pca.explained_variance_)
plt.xlabel('PCA feature')
plt.ylabel('variance')
plt.show()
```

- Second step: intrinsic dimension: number of feature needed to approximate the data essential idea behind dimension reduction
- PCA identifies intrinsic dimension when samples have any number of features
- intrinsic dimension = number of PCA feature with significant variance
- In order to choose intrinsic dimension try all of them and find best accuracy

In [34]:
```
# apply PCA
pca = PCA(n_components = 2)
pca.fit(data3)
transformed = pca.transform(data3)
x = transformed[:,0]
y = transformed[:,1]
plt.scatter(x,y,c = color_list)
plt.show()
```