LAB-1

Q1) Write a python program to import and export data using Pandas library functions

```
[36] import pandas as pd
```

```
[37] airbnb_data = pd.read_csv("/content/austinHousingData.csv")
```

```
airbnb_data.to_csv("/content/austinHousingData.csv")
```

```
airbnb_data.head()
```

	Unnamed: 0.1	Unnamed: 0	zpid	city	streetAddress	zipcode	description	latitude	longitude	propertyTaxRate	...	numOfMiddleSchools	numOfHighSchools	avgSchoolDistance	avgSchoolRating	avgSchoolSize	MedianStudentsPerTeacher	numOfBathrooms
0	0	0	111373431	pflugerville	14424 Lake Victor Dr	78660	14424 Lake Victor Dr, Pflugerville, TX 78660 I...	30.430632	-97.663078	1.98	...	1	1	1.266667	2.666667	1063	14	3.0
1	1	1	120900430	pflugerville	1104 Strickling Dr	78660	Absolutely GORGEOUS 4 Bedroom home with 2 full...	30.432673	-97.661697	1.98	...	1	1	1.400000	2.666667	1063	14	2.0
2	2	2	2084491383	pflugerville	1408 Fort Dessau Rd	78660	Under construction - estimated completion in A...	30.409748	-97.639771	1.98	...	1	1	1.200000	3.000000	1108	14	2.0
3	3	3	120901374	pflugerville	1025 Strickling Dr	78660	Absolutely darling one story home in charming ...	30.432112	-97.661659	1.98	...	1	1	1.400000	2.666667	1063	14	2.0
4	4	4	60134862	pflugerville	15005 Donna Jane Loop	78660	Brimming with appeal & warm livability! Sleek ...	30.437368	-97.656860	1.98	...	1	1	1.133333	4.000000	1223	14	3.0

```
[40] import pandas as pd
```

```
[41] iris_data = pd.read_csv("/content/iris.data")
```

```
[42] iris_data.head()
```

	5.1	3.5	1.4	0.2	Iris-setosa
0	4.9	3.0	1.4	0.2	Iris-setosa
1	4.7	3.2	1.3	0.2	Iris-setosa
2	4.6	3.1	1.5	0.2	Iris-setosa
3	5.0	3.6	1.4	0.2	Iris-setosa
4	5.4	3.9	1.7	0.4	Iris-setosa

Next steps: View recommended plots

```
[43] # Webpage URL
     url = "https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data"

     # Define the column names
     col_names = ["sepal_length_in_cm",
                  "sepal_width_in_cm",
                  "petal_length_in_cm",
                  "petal_width_in_cm",
                  "class"]

     # Read data from URL
     iris_data = pd.read_csv(url, names=col_names)

     iris_data.head()
```

	sepal_length_in_cm	sepal_width_in_cm	petal_length_in_cm	petal_width_in_cm	class
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

Austin & iris

import

```
import pandas as pd
airbnb_data = pd.read_csv("/content/austinHousingData.csv")
airbnb_data.head()
```

output

Export :-

```
airbnb_data.to_csv("/content/austinHousingData.csv")
```

austinHousingData.csv

Reading Data from url :-

~~url & https~~

```
import pandas as pd
iris_data = pd.read_csv("/content/iris.data")
iris_data.head()

url = "https://archive.ics.uci.edu/ml/
       machine-learning-databases/iris/iris.data"

colnames = ["Sepal_Length_in_cm",
            "Sepal_width_in_cm",
            "petal_Length_in_cm",
            "petal_width_in_cm",
            "class"]

iris_data = pd.read_csv(url, names=colnames)
iris_data.head()
```

LAB-2

Use appropriate dataset to building the decision tree (ID3) and apply this knowledge to classify a new sample.

1.) importing data set

```
import numpy as np
import pandas as pd
eps = np.finfo(float).eps
from numpy import log2 as log
import matplotlib.pyplot as plt
```

```
[2] outlook = 'overcast,overcast,overcast,overcast,rainy,rainy,rainy,rainy,rainy,sunny,sunny,sunny,sunny,sunny'.split(',')
    temp = 'hot,cool,mild,hot,mild,cool,cool,mild,mild,hot,hot,mild,cool,mild'.split(',')
    humidity = 'high,normal,high,normal,high,normal,normal,normal,high,high,high,high,normal,normal'.split(',')
    windy = 'FALSE,TRUE,TRUE,FALSE,FALSE,FALSE,TRUE,FALSE,TRUE,FALSE,TRUE,FALSE,FALSE,TRUE'.split(',')
    play = 'yes,yes,yes,yes,yes,yes,no,yes,no,no,no,no,yes,yes'.split(',')
```

```
dataset ={'outlook':outlook,'temp':temp,'humidity':humidity,'windy':windy,'play':play}
df = pd.DataFrame(dataset,columns=['outlook','temp','humidity','windy','play'])
df
```

	outlook	temp	humidity	windy	play
0	overcast	hot	high	FALSE	yes
1	overcast	cool	normal	TRUE	yes
2	overcast	mild	high	TRUE	yes
3	overcast	hot	normal	FALSE	yes
4	rainy	mild	high	FALSE	yes
5	rainy	cool	normal	FALSE	yes
6	rainy	cool	normal	TRUE	no
7	rainy	mild	normal	FALSE	yes
8	rainy	mild	high	TRUE	no
9	sunny	hot	high	FALSE	no
10	sunny	hot	high	TRUE	no
11	sunny	mild	high	FALSE	no
12	sunny	cool	normal	FALSE	yes
13	sunny	mild	normal	TRUE	yes

2)find the entropy

```
[4] ##1. claculate entropy o the whole dataset

    entropy_node = 0  #Initialize Entropy
    values = df.play.unique()  #Unique objects - 'Yes', 'No'
    for value in values:
        fraction = df.play.value_counts()[value]/len(df.play)
        entropy_node += -fraction*np.log2(fraction)

    print(f'Values: {values}')
    print(f'entropy_node: {entropy_node}')
```

```
Values: ['yes' 'no']
entropy_node: 0.9402859586706311
```

```
[5] def ent(df,attribute):
        target_variables = df.play.unique()  #This gives all 'Yes' and 'No'
        variables = df[attribute].unique()    #This gives different features in that attribute (like 'Sweet')


        entropy_attribute = 0
        for variable in variables:
            entropy_each_feature = 0
            for target_variable in target_variables:
                num = len(df[attribute][df[attribute]==variable][df.play ==target_variable]) #numerator
                den = len(df[attribute][df[attribute]==variable])  #denominator
                fraction = num/(den+eps)  #pi
                entropy_each_feature += -fraction*log(fraction+eps) #This calculates entropy for one feature like 'Sweet'
            fraction2 = den/len(df)
            entropy_attribute += -fraction2*entropy_each_feature   #Sums up all the entropy ETaste

        return(abs(entropy_attribute))
    a_entropy = {k:ent(df,k) for k in df.keys()[:-1]}
    a_entropy
```

```
{'outlook': 0.6935361388961914,
 'temp': 0.9110633930116756,
 'humidity': 0.7884504573082889,
 'windy': 0.892158928262361}
```

3)find the information gain

```
[6] def ig(e_dataset,e_attr):
        return(e_dataset-e_attr)
    #entropy_node = entropy of dataset
    #a_entropy[k] = entropy of k(th) attr
    IG = {k:ig(entropy_node,a_entropy[k]) for k in a_entropy}
    IG
```

```
{'outlook': 0.24674981977443977,
 'temp': 0.029222565658955535,
 'humidity': 0.15183550136234225,
 'windy': 0.048127030408270155}
```

4)find the attribute with the max information gain

```
def find_entropy(df):
    Class = df.keys()[-1]   #To make the code generic, changing target variable class name
    entropy = 0
    values = df[Class].unique()
    for value in values:
        fraction = df[Class].value_counts()[value]/len(df[Class])
        entropy += -fraction*np.log2(fraction)
    return entropy


def find_entropy_attribute(df,attribute):
  Class = df.keys()[-1]   #To make the code generic, changing target variable class name
  target_variables = df[Class].unique()  #This gives all 'Yes' and 'No'
  variables = df[attribute].unique()    #This gives different features in that attribute (like 'Hot','Cold' in Temperature)
  entropy2 = 0
  for variable in variables:
      entropy = 0
      for target_variable in target_variables:
          num = len(df[attribute][df[attribute]==variable][df[Class] ==target_variable])
          den = len(df[attribute][df[attribute]==variable])
          fraction = num/(den+eps)
          entropy += -fraction*log(fraction+eps)
      fraction2 = den/len(df)
      entropy2 += -fraction2*entropy
  return abs(entropy2)


def find_winner(df):
    Entropy_att = []
    IG = []
    for key in df.keys()[:-1]:
#         Entropy_att.append(find_entropy_attribute(df,key))
        IG.append(find_entropy(df)-find_entropy_attribute(df,key))
    return df.keys()[:-1][np.argmax(IG)]


def get_subtable(df, node,value):
  return df[df[node] == value].reset_index(drop=True)
```

5)build the tree

```
    def buildTree(df,tree=None):
        Class = df.keys()[-1]   #To make the code generic, changing target variable class name

        #Here we build our decision tree

        #Get attribute with maximum information gain
        node = find_winner(df)

        #Get distinct value of that attribute e.g Salary is node and Low,Med and High are values
        attValue = np.unique(df[node])

        #Create an empty dictionary to create tree
        if tree is None:
            tree={}
            tree[node] = {}

       #We make loop to construct a tree by calling this function recursively.
        #In this we check if the subset is pure and stops if it is pure.

        for value in attValue:

            subtable = get_subtable(df,node,value)
            clValue,counts = np.unique(subtable[Class],return_counts=True)

            if len(counts)==1:#Checking purity of subset
                tree[node][value] = clValue[0]
            else:
                tree[node][value] = buildTree(subtable) #Calling the function recursively

        return tree

    t  = buildTree(df)
    import pprint
    pprint.pprint(t)
```

Output:-

```
{'outlook': {'overcast': 'yes',
             'rainy': {'windy': {'FALSE': 'yes', 'TRUE': 'no'}},
             'sunny': {'humidity': {'high': 'no', 'normal': 'yes'}}}}
```

algo output

LAB-2

Use an appropriate dataset for building the decision tree (ID3). (entropy)

Algorithm:- D is data set.

- Create a root node for the decision tree.
- if all instances in 'D' belong to the same class return a leaf node labelled with that class.
- If the attribute set 'A' is empty return a leaf node labelled with the majority class in 'D'
- Calculate the entropy H(D) of the dataset 'D'
- Calculate information gain of dataset and attribute set.
- Select attribute with highest information gain
- Create a decision node for attribute 'a*)
  - create branch from decision tree node labelled with value 'V'
  - Return decision tree T.

Output:-

Calculate the entropy of whole set.

values : ['yes', 'No']

entropy_node : 0.940285 ...

a-entropy

{ 'outlook' : 0.693536 ...

'temp' : 0.911063 ...

'humidity' : 0.788450 ...

'windy' : 0.892158 ...

}

calculate information gain :-

{'outlook' : 0.2467498 . . .

'temp' : 0.0292225 . . . .

'humidity' : 0.1518355 . . .

'windy' : 0.0482170 . . . .

}

get max information gain attribute

build tree.

{'outlook' : {'overcast' : 'yes',

'raining' : {'windy' : {'false' : 'yes',

'True' : 'no

'Sunny' : {'humidity' : {'high' : 'no',

'normal' : 'ye

LAB -3

KNN:-

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

In [ ]:
```
Iris = pd.read_csv('/content/Iris.csv')
```

In [ ]:
```
Iris.drop('Id',inplace=True,axis=1) #Drop Id column
Iris.head().style.background_gradient(sns.color_palette("YlOrBr", as_cmap=True))
```

Out[ ]:

	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	5.100000	3.500000	1.400000	0.200000	Iris-setosa
1	4.900000	3.000000	1.400000	0.200000	Iris-setosa
2	4.700000	3.200000	1.300000	0.200000	Iris-setosa
3	4.600000	3.100000	1.500000	0.200000	Iris-setosa
4	5.000000	3.600000	1.400000	0.200000	Iris-setosa

In [ ]:
```
X = Iris.iloc[:,:-1] #Set our training data

y = Iris.iloc[:,-1] #Set training labels
```

In [ ]:
```
class KNN:
    """
    K-Nearest Neighbors (KNN) classification algorithm

    Parameters:
    -----------
    n_neighbors : int, optional (default=5)
        Number of neighbors to use in the majority vote.

    Methods:
    --------
    fit(X_train, y_train):
        Stores the values of X_train and y_train.

    predict(X):
        Predicts the class labels for each example in X.

    """
    def __init__(self, n_neighbors=5):
        self.n_neighbors = n_neighbors

    def euclidean_distance(self, x1, x2):
        """
        Calculate the Euclidean distance between two data points.

        Parameters:
        -----------
        x1 : numpy.ndarray, shape (n_features,)
            A data point in the dataset.

        x2 : numpy.ndarray, shape (n_features,)
            A data point in the dataset.

        Returns:
        --------
        distance : float
            The Euclidean distance between x1 and x2.
        """
        return np.linalg.norm(x1 - x2)

    def fit(self, X_train, y_train):
        """
        Stores the values of X_train and y_train.

        Parameters:
        -----------
        X_train : numpy.ndarray, shape (n_samples, n_features)
            The training dataset.

        y_train : numpy.ndarray, shape (n_samples,)
            The target labels.
        """
        self.X_train = X_train
        self.y_train = y_train
```

```python
def fit(self, X_train, y_train):
    """
    Stores the values of X_train and y_train.

    Parameters:
    -----------
    X_train : numpy.ndarray, shape (n_samples, n_features)
        The training dataset.

    y_train : numpy.ndarray, shape (n_samples,)
        The target labels.
    """
    self.X_train = X_train
    self.y_train = y_train

def predict(self, X):
    """
    Predicts the class labels for each example in X.

    Parameters:
    -----------
    X : numpy.ndarray, shape (n_samples, n_features)
        The test dataset.

    Returns:
    --------
    predictions : numpy.ndarray, shape (n_samples,)
        The predicted class labels for each example in X.
    """
    # Create empty array to store the predictions
    predictions = []
    # Loop over X examples
    for x in X:
        # Get prediction using the prediction helper function
        prediction = self._predict(x)
        # Append the prediction to the predictions list
        predictions.append(prediction)
    return np.array(predictions)

def _predict(self, x):
    """
    Predicts the class label for a single example.

    Parameters:
    -----------
    x : numpy.ndarray, shape (n_features,)
        A data point in the test dataset.

    Returns:
    --------
    most_occuring_value : int
        The predicted class label for x.
    """
    # Create empty array to store distances
    distances = []
    # Loop over all training examples and compute the distance between x and all the training examples
    for x_train in self.X_train:
        distance = self.euclidean_distance(x, x_train)
        distances.append(distance)
    distances = np.array(distances)

    # Sort by ascendingly distance and return indices of the given n neighbours
    n_neighbors_idxs = np.argsort(distances)[: self.n_neighbors]
```

In [ ]:

```
def train_test_split(X, y, random_state=42, test_size=0.2):
    """
    Splits the data into training and testing sets.

    Parameters:
        X (numpy.ndarray): Features array of shape (n_samples, n_features).
        y (numpy.ndarray): Target array of shape (n_samples,).
        random_state (int): Seed for the random number generator. Default is 42.
        test_size (float): Proportion of samples to include in the test set. Default is 0.2.

    Returns:
        Tuple[numpy.ndarray]: A tuple containing X_train, X_test, y_train, y_test.
    """
    # Get number of samples
    n_samples = X.shape[0]

    # Set the seed for the random number generator
    np.random.seed(random_state)

    # Shuffle the indices
    shuffled_indices = np.random.permutation(np.arange(n_samples))

    # Determine the size of the test set
    test_size = int(n_samples * test_size)

    # Split the indices into test and train
    test_indices = shuffled_indices[:test_size]
    train_indices = shuffled_indices[test_size:]

    # Split the features and target arrays into test and train
    X_train, X_test = X[train_indices], X[test_indices]
    y_train, y_test = y[train_indices], y[test_indices]

    return X_train, X_test, y_train, y_test
X_train, X_test, y_train, y_test = train_test_split(X.values, y.values, test_size = 0.2, random_state=42) #split the  data
model = KNN(7)
model.fit(X_train, y_train)
def compute_accuracy(y_true, y_pred):
    """
    Computes the accuracy of a classification model.

    Parameters:
    y_true (numpy array): A numpy array of true labels for each data point.
    y_pred (numpy array): A numpy array of predicted labels for each data point.

    Returns:
    float: The accuracy of the model, expressed as a percentage.
    """
    y_true = y_true.flatten()
    total_samples = len(y_true)
    correct_predictions = np.sum(y_true == y_pred)
    return (correct_predictions / total_samples)
predictions = model.predict(X_test)
accuracy = compute_accuracy(y_test, predictions)
print(f" our model got accuracy score of : {accuracy}")
```

```
our model got accuracy score of : 0.9666666666666667
```

LAB-3

① Use appropriate dataset and demonstrate KNN.

Algorithm :-

1.) Input :- Training dataset with labelled data, new unlabelled data point to classify value of K (no. of neighbours to consider.)

2.) Compute distances :- Calculate the distance b/w the new data point and all points in the training dataset. Common distance metrics include Euclidean distance etc

3.) Find the Nearest Neighbors :- Select the K data points from the training dataset that are the closet to the new data point based on the computed distances.

4.) Majority Vote :- Determine the class labels of the K nearest neighbors, and assign the most common class label as the predicted class for the new data point.

Output :-

using iris dataset.

Our model gets accuracy score of : 0.966666---

LINEAR REGRESSION: -

In [1]:
```
# Import Library
import pandas  as pd #Data manipulation
import numpy as np #Data manipulation
import matplotlib.pyplot as plt # Visualization
import seaborn as sns #Visualization
plt.rcParams['figure.figsize'] = [8,5]
plt.rcParams['font.size'] =14
plt.rcParams['font.weight']= 'bold'
plt.style.use('seaborn-whitegrid')
```

```
<ipython-input-1-edfe5ae80b05>:9: MatplotlibDeprecationWarning: The seaborn styles shipped by Matplotlib are deprecated since 3.6, as they no longer correspond to the styles shipped by seaborn. However, they will remain available as 'seaborn-v0_8-<style>'. Alternatively, directly use the seaborn API instead.
  plt.style.use('seaborn-whitegrid')
```

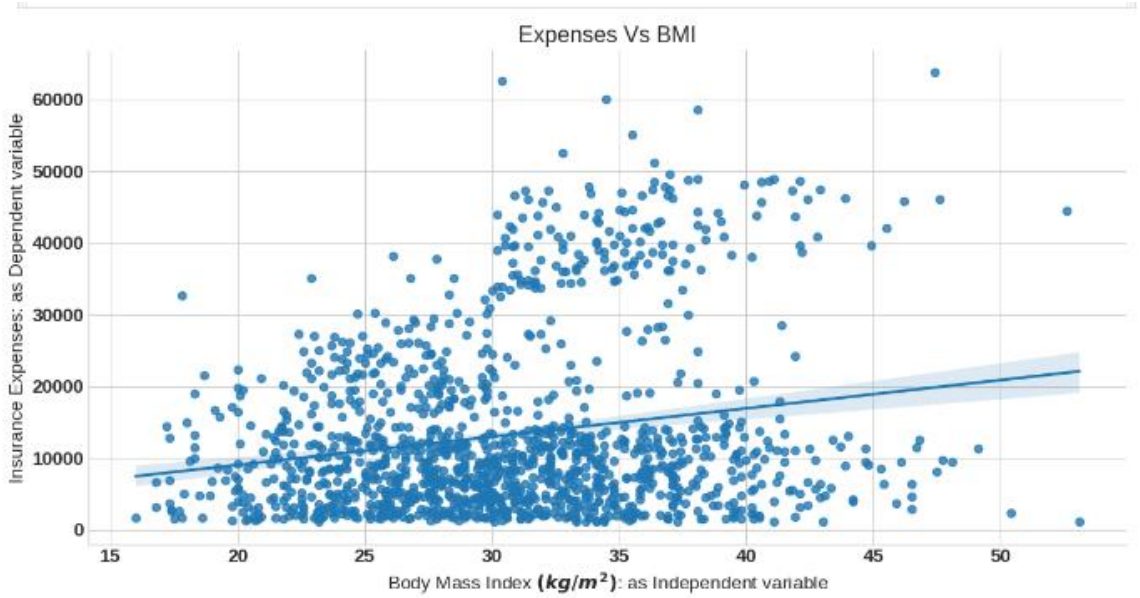
In [3]:
```
# Import necessary library
import pandas as pd

# Define the path to the dataset
path = '/content/insurance.csv'  # Update the path to point to the correct location of your CSV file

# Read the dataset
df = pd.read_csv(path)

# Print the shape of the dataset
print('Number of rows and columns in the data set:', df.shape)
print()

# Display the first few rows of the dataset
df.head()
```

```
Number of rows and columns in the data set: (1338, 7)
```

Out[3]:

	age	sex	bmi	children	smoker	region	expenses
0	19	female	27.9	0	yes	southwest	16884.92
1	18	male	33.8	1	no	southeast	1725.55
2	28	male	33.0	3	no	southeast	4449.46
3	33	male	22.7	0	no	northwest	21984.47
4	32	male	28.9	0	no	northwest	3866.86

In [4]:
```
# Import necessary libraries
import seaborn as sns
import matplotlib.pyplot as plt

# Plotting the lmplot with 'expenses' as the dependent variable
sns.lmplot(x='bmi', y='expenses', data=df, aspect=2, height=6)
plt.xlabel('Body Mass Index ($kg/m^2$): as Independent variable')
plt.ylabel('Insurance Expenses: as Dependent variable')
plt.title('Expenses Vs BMI')
plt.show()
```


Expenses Vs BMI
Insurance Expenses: as Dependent variable
60000
50000
40000
30000
20000
10000
0
15
20
25
30
35
40
45
50
Body Mass Index (kg/m^2): as Independent variable

② Linear Regression :-

1.) Initialize parameters :- Set initial values for coefficients and intercept.

2.) Compute Predictions :-
Use the current parameter values to make predictions for each data point in the training set.
computed as :- $y_i = w_0 + w_1 \times x_{i1} + w_2 \times x_{i2} + \dots + w_n \times x_{in}$

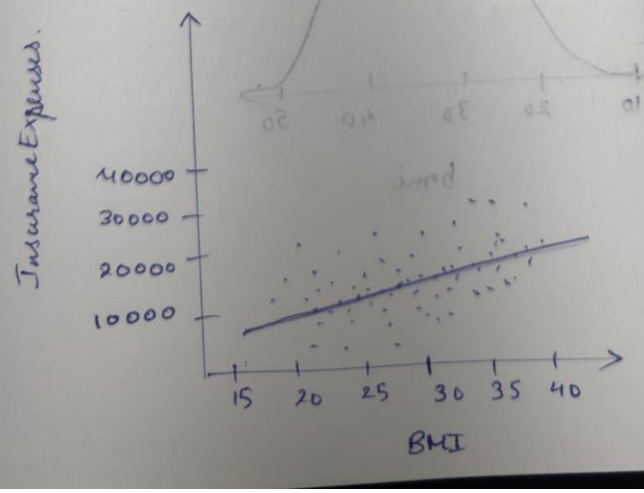
$y_i \rightarrow$ predicted value for data point x_i.
w_0 is the intercept term
$w_1, w_2, \dots, w_n$

3.) Compute loss :- Calculate the difference b/w predicted and actual target values using a loss function.

4.) update parameters :- adjust parameters to minimize the loss using an optimization algorithm. like gradient decent.

Output :-

age	sex	bmi	children	smoker	region	expenses
19	f	27.9	0	Y	SW	16884.94
18	M	33.8	1	N	SE	1725.55
28	M	33.0	3	N	SE	4449.46
33	M	22.7	0	N	NW	21984.47
32	M	28.9	0	N	NW	3866.86



MULTIPLE REGRESSION:-

In [6]:
```
# Import Libraries
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
from sklearn.linear_model import LinearRegression
```

In [2]:
```
# Get dataset
df_start = pd.read_csv('/content/insurance.csv')
df_start.head()
```

Out[2]:

	age	sex	bmi	children	smoker	region	expenses
0	19	female	27.9	0	yes	southwest	16884.92
1	18	male	33.8	1	no	southeast	1725.55
2	28	male	33.0	3	no	southeast	4449.46
3	33	male	22.7	0	no	northwest	21984.47
4	32	male	28.9	0	no	northwest	3866.86

In [3]:
```
# Describe data
df_start.describe()
```

Out[3]:

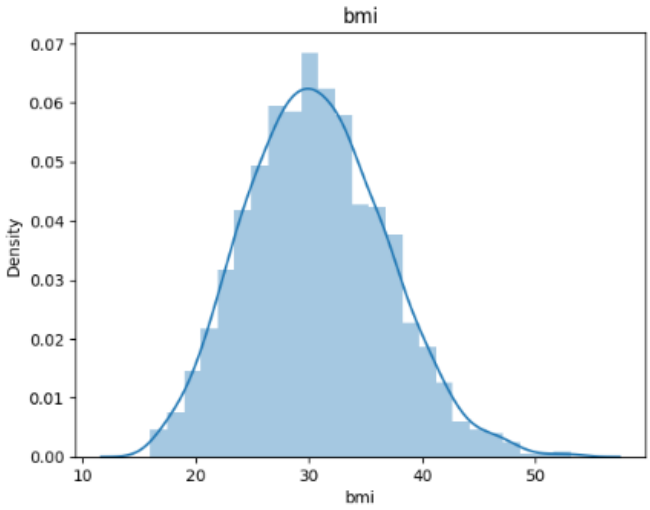
	age	bmi	children	expenses
count	1338.000000	1338.000000	1338.000000	1338.000000
mean	39.207025	30.665471	1.094918	13270.422414
std	14.049960	6.098382	1.205493	12110.011240
min	18.000000	16.000000	0.000000	1121.870000
25%	27.000000	26.300000	0.000000	4740.287500
50%	39.000000	30.400000	1.000000	9382.030000
75%	51.000000	34.700000	2.000000	16639.915000
max	64.000000	53.100000	5.000000	63770.430000

In [10]:
```
# Data distribution
plt.title('bmi')
sns.distplot(df_start['bmi'])
plt.show()
```

In [10]:

```
# Data distribution
plt.title('bmi')
sns.distplot(df_start['bmi'])
plt.show()
```

```
<ipython-input-10-ba0b86e43c8d>:3: UserWarning:

`distplot` is a deprecated function and will be removed in seaborn v0.14.0.

Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `histplot` (an axes-level function for histograms).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  sns.distplot(df_start['bmi'])
```

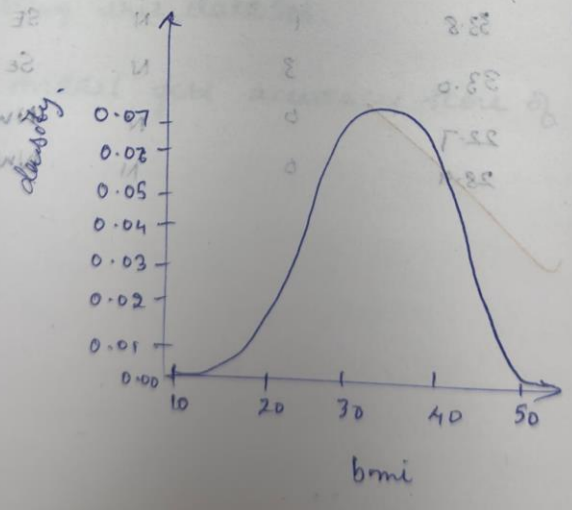


③ Multiple Regression :-
linear

1.) Import the required python packages
2.) Load the dataset
3.) Data analysis
4.) Split the dataset into dependent / independent variables
5.) One -> hot - Encoding of categorial
It is a method of represent a categorial variable in a numerical way, by creating new binary columns (known as "dummy variables") for each category in the original variable.
6.) Split data into train & test set.
7.) Train the regression model

Output:-

using the same dataset
calculate mean, count, std, min, 25%, 50%, 75%, max



LOGISTIC REGRESSION: -

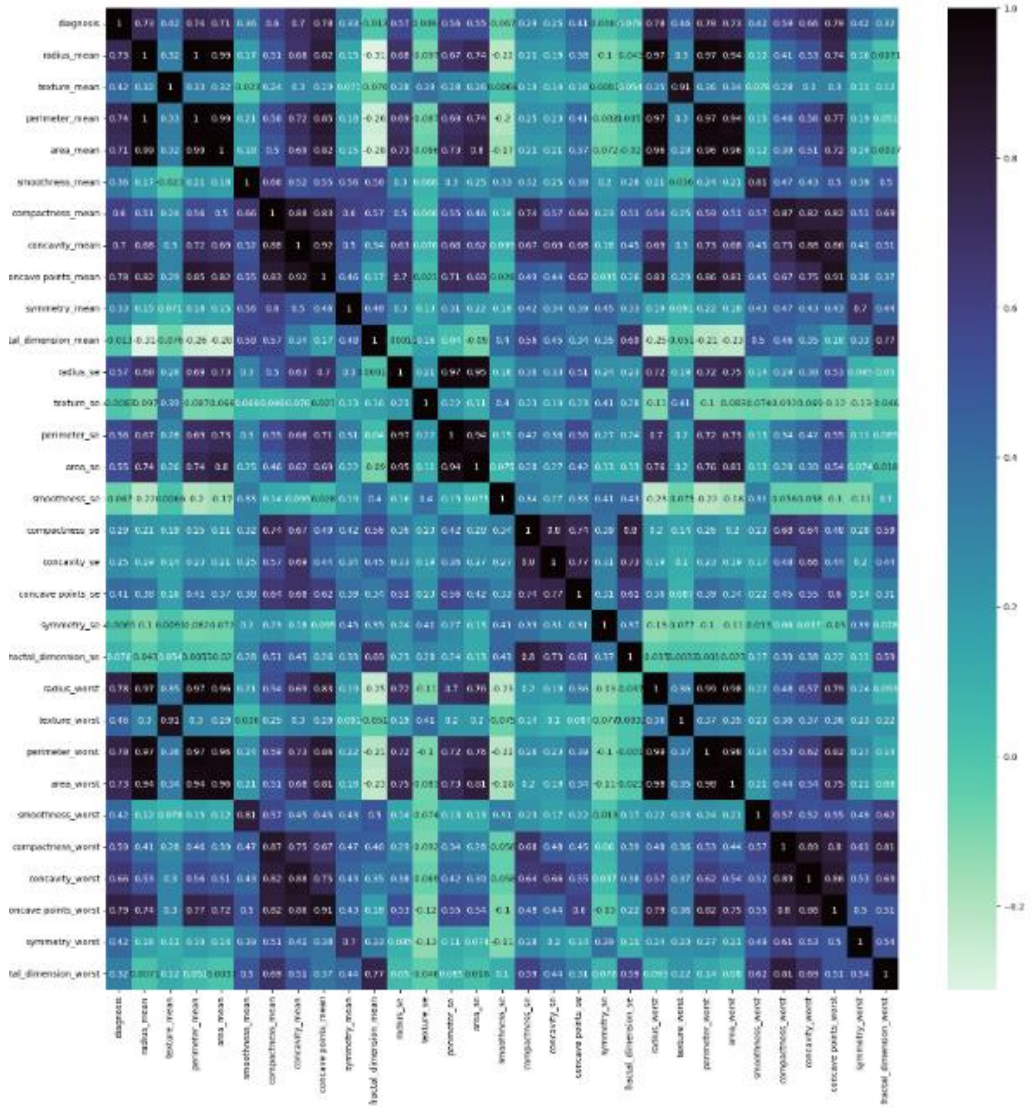
In [1]:
```
import math
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px
import pprint
import pickle
```

In [3]:
```
df = pd.read_csv('/content/breast-cancer.csv')
```

In [4]:
```
df.drop('id', axis=1, inplace=True) #drop redundant columns
df['diagnosis'] = (df['diagnosis'] == 'M').astype(int) #encode the label into 1/0
```

In [5]:
```
corr = df.corr()
```

In [6]:
```
plt.figure(figsize=(20,20))
sns.heatmap(corr, cmap='mako_r',annot=True)
plt.show()
```



In [6]:

In [7]:
```
# Get the absolute value of the correlation
cor_target = abs(corr["diagnosis"])

# Select highly correlated features (thresold = 0.2)
relevant_features = cor_target[cor_target>0.2]

# Collect the names of the features
names = [index for index, value in relevant_features.items()]

# Drop the target variable from the results
names.remove('diagnosis')

# Display the results
pprint.pprint(names)
```

```
['radius_mean',
 'texture_mean',
 'perimeter_mean',
 'area_mean',
 'smoothness_mean',
 'compactness_mean',
 'concavity_mean',
 'concave points_mean',
 'symmetry_mean',
 'radius_se',
 'perimeter_se',
 'area_se',
 'compactness_se',
 'concavity_se',
 'concave points_se',
 'radius_worst',
 'texture_worst',
 'perimeter_worst',
 'area_worst',
 'smoothness_worst',
 'compactness_worst',
 'concavity_worst',
 'concave points_worst',
 'symmetry_worst',
 'fractal_dimension_worst']
```

[8]:
```
X = df[names].values
y = df['diagnosis'].values
```

[9]:
```
def train_test_split(X, y, random_state=42, test_size=0.2):
    """
    Splits the data into training and testing sets.

    Parameters:
        X (numpy.ndarray): Features array of shape (n_samples, n_features).
        y (numpy.ndarray): Target array of shape (n_samples,).
        random_state (int): Seed for the random number generator. Default is 42.
        test_size (float): Proportion of samples to include in the test set. Default is 0.2.

    Returns:
        Tuple[numpy.ndarray]: A tuple containing X_train, X_test, y_train, y_test.
    """
    # Get number of samples
    n_samples = X.shape[0]

    # Set the seed for the random number generator
    np.random.seed(random_state)

    # Shuffle the indices
    shuffled_indices = np.random.permutation(np.arange(n_samples))

    # Determine the size of the test set
    test_size = int(n_samples * test_size)

    # Split the indices into test and train
    test_indices = shuffled_indices[:test_size]
    train_indices = shuffled_indices[test_size:]

    # Split the features and target arrays into test and train
    X_train, X_test = X[train_indices], X[test_indices]
    y_train, y_test = y[train_indices], y[test_indices]

    return X_train, X_test, y_train, y_test
```

10]:
```
X_train, X_test, y_train, y_test = train_test_split(X,y)
```

11]:
```
def standardize_data(X_train, X_test):
    """
    Standardizes the input data using mean and standard deviation.

    Parameters:
        X_train (numpy.ndarray): Training data.
        X_test (numpy.ndarray): Testing data.

    Returns:
        Tuple of standardized training and testing data.
    """
    # Calculate the mean and standard deviation using the training data
    mean = np.mean(X_train, axis=0)
    std = np.std(X_train, axis=0)

    # Standardize the data
    X_train = (X_train - mean) / std
    X_test = (X_test - mean) / std

    return X_train, X_test

X_train, X_test = standardize_data(X_train, X_test)
```

In [12]:

```
def sigmoid(z):
    """
    Compute the sigmoid function for a given input.

    The sigmoid function is a mathematical function used in logistic regression and neural networks
    to map any real-valued number to a value between 0 and 1.

    Parameters:
        z (float or numpy.ndarray): The input value(s) for which to compute the sigmoid.

    Returns:
        float or numpy.ndarray: The sigmoid of the input value(s).

    Example:
        >>> sigmoid(0)
        0.5
    """
    # Compute the sigmoid function using the formula: 1 / (1 + e^(-z)).
    sigmoid_result = 1 / (1 + np.exp(-z))

    # Return the computed sigmoid value.
    return sigmoid_result
```

In [13]:

```
z = np.linspace(-12, 12, 200)

fig = px.line(x=z, y=sigmoid(z),title='Logistic Function',template="plotly_dark")
fig.update_layout(
    title_font_color="#41BEE9",
    xaxis=dict(color="#41BEE9"),
    yaxis=dict(color="#41BEE9")
)
fig.show()
```

In [14]:

```
class LogisticRegression:
    """
    Logistic Regression model.

    Parameters:
        learning_rate (float): Learning rate for the model.

    Methods:
        initialize_parameter(): Initializes the parameters of the model.
        sigmoid(z): Computes the sigmoid activation function for given input z.
        forward(X): Computes forward propagation for given input X.
        compute_cost(predictions): Computes the cost function for given predictions.
        compute_gradient(predictions): Computes the gradients for the model using given predictions.
        fit(X, y, iterations, plot_cost): Trains the model on given input X and labels y for specified iterations.
        predict(X): Predicts the labels for given input X.
    """

    def __init__(self, learning_rate=0.0001):
        np.random.seed(1)
        self.learning_rate = learning_rate

    def initialize_parameter(self):
        """
        Initializes the parameters of the model.
        """
        self.W = np.zeros(self.X.shape[1])
        self.b = 0.0


    def forward(self, X):
        """
        Computes forward propagation for given input X.

        Parameters:
            X (numpy.ndarray): Input array.

        Returns:
            numpy.ndarray: Output array.
        """
#         print(X.shape, self.W.shape)
        Z = np.matmul(X, self.W) + self.b
        A = sigmoid(Z)
        return A

    def compute_cost(self, predictions):
        """
        Computes the cost function for given predictions.

        Parameters:
            predictions (numpy.ndarray): Predictions of the model.

        Returns:
            float: Cost of the model.
        """
        m = self.X.shape[0]  # number of training examples
        # compute the cost
        cost = np.sum((-np.log(predictions + 1e-8) * self.y) + (-np.log(1 - predictions + 1e-8)) * (
                1 - self.y))  # we are adding small value epsilon to avoid log of 0
        cost = cost / m
        return cost
    def compute_gradient(self, predictions):
        """
        Computes the gradients for the model using given predictions.

        Parameters:
            predictions (numpy.ndarray): Predictions of the model.
        """
        # get training shape
        m = self.X.shape[0]

        # compute gradients
        self.dW = np.matmul(self.X.T, (predictions - self.y))
        self.dW = np.array([np.mean(grad) for grad in self.dW])

        self.db = np.sum(np.subtract(predictions, self.y))
```

```
    def save_model(self, filename=None):
        """
        Save the trained model to a file using pickle.

        Parameters:
            filename (str): The name of the file to save the model to.
        """
        model_data = {
            'learning_rate': self.learning_rate,
            'W': self.W,
            'b': self.b
        }

        with open(filename, 'wb') as file:
            pickle.dump(model_data, file)

    @classmethod
    def load_model(cls, filename):
        """
        Load a trained model from a file using pickle.

        Parameters:
            filename (str): The name of the file to load the model from.

        Returns:
            LogisticRegression: An instance of the LogisticRegression class with loaded parameters.
        """
        with open(filename, 'rb') as file:
            model_data = pickle.load(file)

        # Create a new instance of the class and initialize it with the loaded parameters
        loaded_model = cls(model_data['learning_rate'])
        loaded_model.W = model_data['W']
        loaded_model.b = model_data['b']

        return loaded_model
```

In [15]:
```
lg = LogisticRegression()
lg.fit(X_train, y_train,100000)
```

```
Cost after iteration 0: 0.6931471605599454
Cost after iteration 10000: 0.2570778370558246
Cost after iteration 20000: 0.19529178673689726
Cost after iteration 30000: 0.16685820756163852
Cost after iteration 40000: 0.14978939548676498
Cost after iteration 50000: 0.1381876134031554
Cost after iteration 60000: 0.1296814121248933
Cost after iteration 70000: 0.1231144039988139
Cost after iteration 80000: 0.11785163708790082
Cost after iteration 90000: 0.113513771386002
```

In [16]:
```
lg.save_model("model.pkl")
```

In [17]:

```python
class ClassificationMetrics:
    @staticmethod
    def accuracy(y_true, y_pred):
        """
        Computes the accuracy of a classification model.

        Parameters:
        y_true (numpy array): A numpy array of true labels for each data point.
        y_pred (numpy array): A numpy array of predicted labels for each data point.

        Returns:
        float: The accuracy of the model, expressed as a percentage.
        """
        y_true = y_true.flatten()
        total_samples = len(y_true)
        correct_predictions = np.sum(y_true == y_pred)
        return (correct_predictions / total_samples)

    @staticmethod
    def precision(y_true, y_pred):
        """
        Computes the precision of a classification model.

        Parameters:
        y_true (numpy array): A numpy array of true labels for each data point.
        y_pred (numpy array): A numpy array of predicted labels for each data point.

        Returns:
        float: The precision of the model, which measures the proportion of true positive predictions
        out of all positive predictions made by the model.
        """
        true_positives = np.sum((y_true == 1) & (y_pred == 1))
        false_positives = np.sum((y_true == 0) & (y_pred == 1))
        return true_positives / (true_positives + false_positives)

    @staticmethod
    def recall(y_true, y_pred):
        """
        Computes the recall (sensitivity) of a classification model.

        Parameters:
        y_true (numpy array): A numpy array of true labels for each data point.
        y_pred (numpy array): A numpy array of predicted labels for each data point.

        Returns:
        float: The recall of the model, which measures the proportion of true positive predictions
        out of all actual positive instances in the dataset.
        """
        true_positives = np.sum((y_true == 1) & (y_pred == 1))
        false_negatives = np.sum((y_true == 1) & (y_pred == 0))
        return true_positives / (true_positives + false_negatives)

    @staticmethod
    def f1_score(y_true, y_pred):
        """
        Computes the F1-score of a classification model.

        Parameters:
        y_true (numpy array): A numpy array of true labels for each data point.
        y_pred (numpy array): A numpy array of predicted labels for each data point.

        Returns:
        float: The F1-score of the model, which is the harmonic mean of precision and recall.
        """
        precision_value = ClassificationMetrics.precision(y_true, y_pred)
        recall_value = ClassificationMetrics.recall(y_true, y_pred)
        return 2 * (precision_value * recall_value) / (precision_value + recall_value)
```

In [18]:

```python
model = LogisticRegression.load_model("model.pkl")


y_pred = model.predict(X_test)
accuracy = ClassificationMetrics.accuracy(y_test, y_pred)
precision = ClassificationMetrics.precision(y_test, y_pred)
recall = ClassificationMetrics.recall(y_test, y_pred)
f1_score = ClassificationMetrics.f1_score(y_test, y_pred)

print(f"Accuracy: {accuracy:.2%}")
print(f"Precision: {precision:.2%}")
print(f"Recall: {recall:.2%}")
print(f"F1-Score: {f1_score:.2%}")
```

```
Accuracy: 98.23%
Precision: 100.00%
Recall: 95.24%
F1-Score: 97.56%
```

Output :-

Accuracy-98.2%

Precision-100.0%

Recall-95.24%

F1-Score-97.5%

KMEANS:-

In [ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

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

In [ ]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import plotly.express as px
import seaborn as sns
import plotly.graph_objects as go
```

In [ ]:
```
iris = pd.read_csv("/content/iris.csv") #Load Data
```

In [ ]:
```
X = iris.iloc[:,:-1] #Set our training data

y = iris.iloc[:,-1] #We'll use this just for visualization as clustering doesn't require labels
```

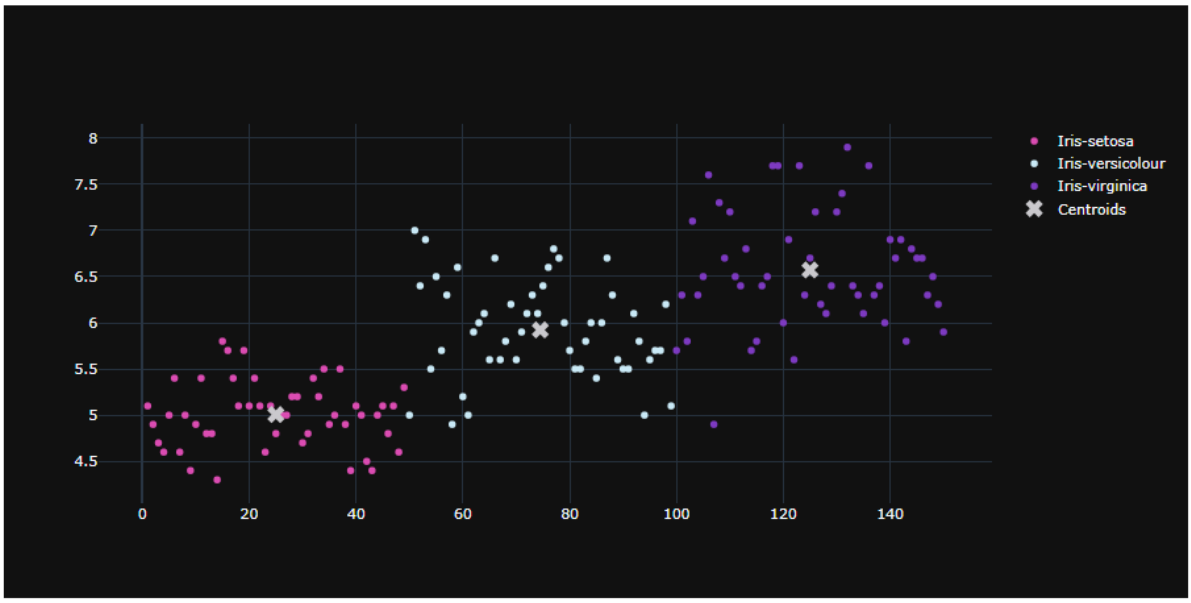
In [ ]:
```
class Kmeans:
    """
    K-Means clustering algorithm implementation.

    Parameters:
        K (int): Number of clusters

    Attributes:
        K (int): Number of clusters
        centroids (numpy.ndarray): Array containing the centroids of each cluster

    Methods:
        __init__(self, K): Initializes the Kmeans instance with the specified number of clusters.
        initialize_centroids(self, X): Initializes the centroids for each cluster by selecting K random points from the dat
        assign_points_centroids(self, X): Assigns each point in the dataset to the nearest centroid.
        compute_mean(self, X, points): Computes the mean of the points assigned to each centroid.
        fit(self, X, iterations=10): Clusters the dataset using the K-Means algorithm.
    """

    def __init__(self, K):
        assert K > 0, "K should be a positive integer."
        self.K = K

    def initialize_centroids(self, X):
        assert X.shape[0] >= self.K, "Number of data points should be greater than or equal to K."

        randomized_X = np.random.permutation(X.shape[0])
        centroid_idx = randomized_X[:self.K] # get the indices for the centroids
        self.centroids = X[centroid_idx] # assign the centroids to the selected points

    def assign_points_centroids(self, X):
        """
        Assign each point in the dataset to the nearest centroid.

        Parameters:
        X (numpy.ndarray): dataset to cluster

        Returns:
        numpy.ndarray: array containing the index of the centroid for each point
        """
        X = np.expand_dims(X, axis=1) # expand dimensions to match shape of centroids
        distance = np.linalg.norm((X - self.centroids), axis=-1) # calculate Euclidean distance between each point and each
        points = np.argmin(distance, axis=1) # assign each point to the closest centroid
        assert len(points) == X.shape[0], "Number of assigned points should equal the number of data points."
        return points

    def compute_mean(self, X, points):
        """
        Compute the mean of the points assigned to each centroid.

        Parameters:
        X (numpy.ndarray): dataset to cluster
        points (numpy.ndarray): array containing the index of the centroid for each point
```

```
        Returns:
        numpy.ndarray: array containing the final centroids for each cluster
        numpy.ndarray: array containing the index of the centroid for each point
        """
        self.initialize_centroids(X) # initialize the centroids
        for i in range(iterations):
            points = self.assign_points_centroids(X) # assign each point to the nearest centroid
            self.centroids = self.compute_mean(X, points) # compute the new centroids based on the mean of their points

            # Assertions for debugging and validation
            assert len(self.centroids) == self.K, "Number of centroids should equal K."
            assert X.shape[1] == self.centroids.shape[1], "Dimensionality of centroids should match input data."
            assert max(points) < self.K, "Cluster index should be less than K."
            assert min(points) >= 0, "Cluster index should be non-negative."

        return self.centroids, points
```

]:
```
X = X.values
```

]:
```
kmeans = Kmeans(3)

centroids, points = kmeans.fit(X, 1000)
```

]:
```
fig = go.Figure()
fig.add_trace(go.Scatter(
    x=X[points == 0, 0], y=X[points == 0, 1],
    mode='markers',marker_color='#DB4CB2',name='Iris-setosa'
))

fig.add_trace(go.Scatter(
    x=X[points == 1, 0], y=X[points == 1, 1],
    mode='markers',marker_color='#c9e9f6',name='Iris-versicolour'
))

fig.add_trace(go.Scatter(
    x=X[points == 2, 0], y=X[points == 2, 1],
    mode='markers',marker_color='#7D3AC1',name='Iris-virginica'
))

fig.add_trace(go.Scatter(
    x=centroids[:, 0], y=centroids[:,1],
    mode='markers',marker_color='#CAC9CD',marker_symbol=4,marker_size=13,name='Centroids'
))
fig.update_layout(template='plotly_dark',width=1000, height=500,)
```



Kmeans 24/05/24

→ Kmeans clustering algorithm

① Select the number K to decide the number of clusters

② Select random K points or centroids

③ Assign each point to the closest cubroid, which will from the prefined K clusters

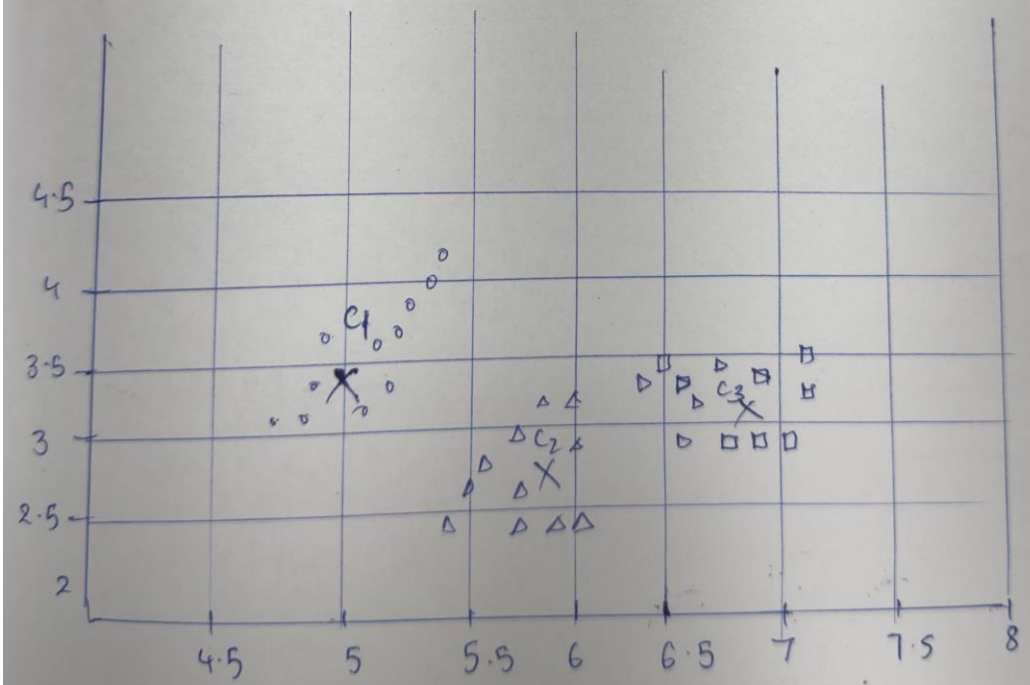
④ Calculate the variance and place a new centroid of each cluster.

⑤ Repeat ③ step, reassign the centroid

⑥ if any reassignment occurs go to step ④ else go to finish

⑦ The model is ready

output



PCA:-

In [2]:
```
import seaborn as sns
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import plotly.express as px
import plotly.graph_objects as go
from plotly.subplots import make_subplots
```

In [3]:
```
df = pd.read_csv('/content/breast-cancer.csv')
df.head()
```

Out[3]:

	id	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_mean
0	842302	M	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001
1	842517	M	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869
2	84300903	M	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974
3	84348301	M	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414
4	84358402	M	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980

5 rows × 32 columns

In [4]:
```
df.drop('id', axis=1, inplace=True) #drop redundant columns
```

In [5]:
```
df['diagnosis'] = (df['diagnosis'] == 'M').astype(int) #encode the label into 1/0
```

In [6]:
```
corr = df.corr()
```

In [7]:
```
# Get the absolute value of the correlation
cor_target = abs(corr["diagnosis"])

# Select highly correlated features (thresold = 0.2)
relevant_features = cor_target[cor_target>0.2]

# Collect the names of the features
names = [index for index, value in relevant_features.items()]

# Drop the target variable from the results
names.remove('diagnosis')

# Display the results
print(names)
```

```
['radius_mean', 'texture_mean', 'perimeter_mean', 'area_mean', 'smoothness_mean', 'compactness_mean', 'concavity_mean', 'conc
ave points_mean', 'symmetry_mean', 'radius_se', 'perimeter_se', 'area_se', 'compactness_se', 'concavity_se', 'concave points_
se', 'radius_worst', 'texture_worst', 'perimeter_worst', 'area_worst', 'smoothness_worst', 'compactness_worst', 'concavity_wo
rst', 'concave points_worst', 'symmetry_worst', 'fractal_dimension_worst']
```

In [8]:
```
X = df[names].values
```

In [9]:

```
class PCA:
    """
    Principal Component Analysis (PCA) class for dimensionality reduction.
    """

    def __init__(self, n_components):
        """
        Constructor method that initializes the PCA object with the number of components to retain.

        Args:
        - n_components (int): Number of principal components to retain.
        """
        self.n_components = n_components
    def fit(self, X):
        """
        Fits the PCA model to the input data and computes the principal components.

        Args:
        - X (numpy.ndarray): Input data matrix with shape (n_samples, n_features).
        """
        # Compute the mean of the input data along each feature dimension.
        mean = np.mean(X, axis=0)

        # Subtract the mean from the input data to center it around zero.
        X = X - mean

        # Compute the covariance matrix of the centered input data.
        cov = np.cov(X.T)

        # Compute the eigenvectors and eigenvalues of the covariance matrix.
        eigenvalues, eigenvectors = np.linalg.eigh(cov)
        # Reverse the order of the eigenvalues and eigenvectors.
        eigenvalues = eigenvalues[::-1]
        eigenvectors = eigenvectors[:,::-1]

        # Keep only the first n_components eigenvectors as the principal components.
        self.components = eigenvectors[:,:self.n_components]

        # Compute the explained variance ratio for each principal component.
        # Compute the total variance of the input data
        total_variance = np.sum(np.var(X, axis=0))

        # Compute the variance explained by each principal component
        self.explained_variances = eigenvalues[:self.n_components]

        # Compute the explained variance ratio for each principal component
        self.explained_variance_ratio_ = self.explained_variances / total_variance
    def transform(self, X):
        """
        Transforms the input data by projecting it onto the principal components.

        Args:
```

```
        Args:
        - X (numpy.ndarray): Input data matrix with shape (n_samples, n_features).

        Returns:
        - transformed_data (numpy.ndarray): Transformed data matrix with shape (n_samples, n_components).
        """
        # Center the input data around zero using the mean computed during the fit step.
        X = X - np.mean(X, axis=0)

        # Project the centered input data onto the principal components.
        transformed_data = np.dot(X, self.components)

        return transformed_data

    def fit_transform(self, X):
        """
        Fits the PCA model to the input data and computes the principal components then
        transforms the input data by projecting it onto the principal components.

        Args:
        - X (numpy.ndarray): Input data matrix with shape (n_samples, n_features).
        """
        self.fit(X)
        transformed_data = self.transform(X)
        return transformed_data
```

In [10]:
```
pca = PCA(2)
```

In [11]:
```
pca.fit(X)
```

In [12]:
```
pca.explained_variance_ratio_
```

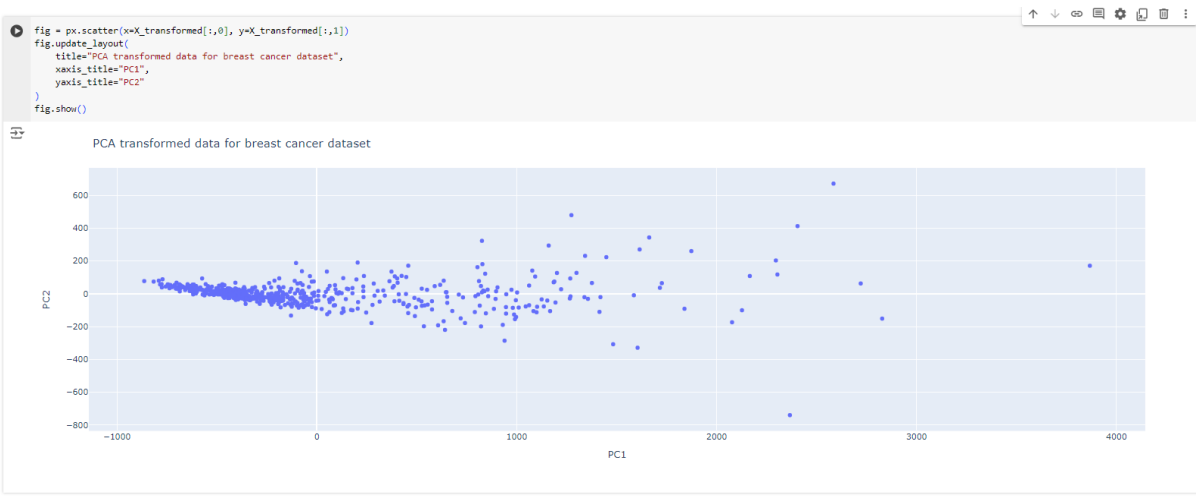
Out[12]:
```
array([0.98377428, 0.01620498])
```

In [13]:
```
X_transformed = pca.transform(X)
```

In [14]:
```
X_transformed[:,1].shape
```

Out[14]:
```
(569,)
```

In [15]:
```
fig = px.scatter(x=X_transformed[:,0], y=X_transformed[:,1])
fig.update_layout(
    title="PCA transformed data for breast cancer dataset",
    xaxis_title="PC1",
    yaxis_title="PC2"
)
fig.show()
```



PCA 24/05/24

→ PCA

① Calculate mean

② calculation of covariance matrix

③ Eigenvalues of the covariance matrix

④ Computation of the eigenvector - and eigenvalues

⑤ Computation of first principal components

⑥ Geometrical meaning of first principal components.

Output:

pca.explained_variance_ratio_

array [[0.98377428, 0.01620428]]