

MANOJ KUMAR - 2048015

Data Read

In [1]:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
df=pd.read_csv("kidney_disease.csv")
```

In [2]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 400 entries, 0 to 399
Data columns (total 26 columns):
 #   Column                Non-Null Count  Dtype  
---  -
 0   id                    400 non-null   int64  
 1   age                  391 non-null   float64
 2   bp                   388 non-null   float64
 3   sg                   353 non-null   float64
 4   al                   354 non-null   float64
 5   su                   351 non-null   float64
 6   rbc                  248 non-null   object  
 7   pc                   335 non-null   object  
 8   pcc                  396 non-null   object  
 9   ba                   396 non-null   object  
10  bgr                  356 non-null   float64
11  bu                   381 non-null   float64
12  sc                   383 non-null   float64
13  sod                  313 non-null   float64
14  pot                  312 non-null   float64
15  hemo                 348 non-null   float64
16  pcv                  330 non-null   object  
17  wc                   295 non-null   object  
18  rc                   270 non-null   object  
19  htn                  398 non-null   object  
20  dm                   398 non-null   object  
21  cad                  398 non-null   object  
22  appet               399 non-null   object  
23  pe                   399 non-null   object  
24  ane                  399 non-null   object  
25  classification       400 non-null   object  
dtypes: float64(11), int64(1), object(14)
memory usage: 81.4+ KB
```

Data Pre-Processing

Data pre-processing is a way to convert the noisy and huge data into relevant and clean data, as the data available is Real world data, so it contains inaccurate data, missing values and other Noisy data, for removing this incon

sistent data from the Dataset, the proposed system have to clean the raw data.

This is an important part to complete the prediction model. It reduces the dimensionality and helps the machine to achieve better results. This is one of the most time consuming stage in building a classification model.

Following data pre-processing steps are followed:

Looking Up For Proper Format

Finding Missing Values

Data Transformation

In [3]:

```
# '?' character remove process in the dataset

for i in ['rc', 'wc', 'pcv']:
    df[i] = df[i].str.extract('(\d+)').astype(float)
```

In [4]:

```
# Filling missing numeric data in the dataset with mean

for i in ['age', 'bp', 'sg', 'al', 'su', 'bgr', 'bu', 'sc', 'sod', 'pot', 'hemo', 'rc', 'wc', 'pcv']:
    df[i].fillna(df[i].mean(), inplace=True)
```

In [5]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 400 entries, 0 to 399
Data columns (total 26 columns):
#   Column                Non-Null Count  Dtype
---  -
0   id                    400 non-null   int64
1   age                  400 non-null   float64
2   bp                   400 non-null   float64
3   sg                   400 non-null   float64
4   al                   400 non-null   float64
5   su                   400 non-null   float64
6   rbc                  248 non-null   object
7   pc                   335 non-null   object
8   pcc                  396 non-null   object
9   ba                   396 non-null   object
10  bgr                  400 non-null   float64
11  bu                   400 non-null   float64
12  sc                   400 non-null   float64
13  sod                  400 non-null   float64
14  pot                  400 non-null   float64
15  hemo                 400 non-null   float64
16  pcv                  400 non-null   float64
17  wc                   400 non-null   float64
18  rc                   400 non-null   float64
19  htn                  398 non-null   object
20  dm                   398 non-null   object
21  cad                  398 non-null   object
22  appet                399 non-null   object
23  pe                   399 non-null   object
24  ane                  399 non-null   object
25  classification       400 non-null   object
dtypes: float64(14), int64(1), object(11)
memory usage: 81.4+ KB
```

In [6]:

```
# Removing tab spaces in the data
```

```
df['dm'] = df['dm'].replace(to_replace={'\tno':'no','\tyes':'yes',' yes':'yes'})
df['cad'] = df['cad'].replace(to_replace='\tno',value='no')
df['classification'] = df['classification'].replace(to_replace='ckd\t',value='ckd')
```

Data Transformation:

In this step we transform the given real data into required format. The data downloaded consists of Nominal, Real and Decimal values. In this step we convert the Nominal data into numerical data of the form 0 and 1. The positive value is assigned the value of 1 and the negative value is assigned the value of 0. Now the resultant csv file comprises of all the integer and decimal values for different CKD related attributes.

In [7]:

```
# Mapping the text to 1/0 and cleaning the dataset
```

```
df[['htn','dm','cad','pe','ane']] = df[['htn','dm','cad','pe','ane']].replace(to_replace={'normal':0})
df[['rbc','pc']] = df[['rbc','pc']].replace(to_replace={'abnormal':1,'normal':0})
df[['pcc','ba']] = df[['pcc','ba']].replace(to_replace={'present':1,'notpresent':0})
df[['appet']] = df[['appet']].replace(to_replace={'good':1,'poor':0})
df['classification'] = df['classification'].replace(to_replace={'ckd':1,'notckd':0})

df.rename(columns={'classification':'class'},inplace=True)
```

In [8]:

```
df.drop('id',axis=1,inplace=True)
```

In [9]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
```

```
RangeIndex: 400 entries, 0 to 399
```

```
Data columns (total 25 columns):
```

| # | Column | Non-Null Count | Dtype |
|----|--------|----------------|---------|
| 0 | age | 400 non-null | float64 |
| 1 | bp | 400 non-null | float64 |
| 2 | sg | 400 non-null | float64 |
| 3 | al | 400 non-null | float64 |
| 4 | su | 400 non-null | float64 |
| 5 | rbc | 248 non-null | float64 |
| 6 | pc | 335 non-null | float64 |
| 7 | pcc | 396 non-null | float64 |
| 8 | ba | 396 non-null | float64 |
| 9 | bgr | 400 non-null | float64 |
| 10 | bu | 400 non-null | float64 |
| 11 | sc | 400 non-null | float64 |
| 12 | sod | 400 non-null | float64 |
| 13 | pot | 400 non-null | float64 |
| 14 | hemo | 400 non-null | float64 |
| 15 | pcv | 400 non-null | float64 |
| 16 | wc | 400 non-null | float64 |
| 17 | rc | 400 non-null | float64 |
| 18 | htn | 398 non-null | float64 |
| 19 | dm | 398 non-null | float64 |
| 20 | cad | 398 non-null | float64 |
| 21 | appet | 399 non-null | float64 |
| 22 | pe | 399 non-null | float64 |
| 23 | ane | 399 non-null | float64 |
| 24 | class | 400 non-null | int64 |

```
dtypes: float64(24), int64(1)
```

```
memory usage: 78.2 KB
```

Finding Missing Values:

When the data collected is real world data, and then it will contain missing values. This brings more change in the prediction accuracy. Sometimes these missing values can be simply deleted or ignored if they are not large in number. It is the simplest way to handle the missing data but it is not considered healthy for the model as the missing value can be an important attribute contributing to the disease. The missing values can also be replaced by zero this will not bring any change as whole, but this method cannot be much yielding. So an efficient way to handle missing values is to use mean, average of the observed attribute or value. This way we lead to more genuine data and better prediction results

In [10]:

```
# Filling the missing string data as the most repetitive (mod)

df=df.apply(lambda x:x.fillna(x.value_counts().index[0]))
```

In [11]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 400 entries, 0 to 399
Data columns (total 25 columns):
 #   Column      Non-Null Count  Dtype  
---  -
 0   age         400 non-null    float64
 1   bp          400 non-null    float64
 2   sg          400 non-null    float64
 3   al          400 non-null    float64
 4   su          400 non-null    float64
 5   rbc         400 non-null    float64
 6   pc          400 non-null    float64
 7   pcc         400 non-null    float64
 8   ba          400 non-null    float64
 9   bgr         400 non-null    float64
10  bu          400 non-null    float64
11  sc          400 non-null    float64
12  sod         400 non-null    float64
13  pot         400 non-null    float64
14  hemo        400 non-null    float64
15  pcv         400 non-null    float64
16  wc          400 non-null    float64
17  rc          400 non-null    float64
18  htn         400 non-null    float64
19  dm          400 non-null    float64
20  cad         400 non-null    float64
21  appet       400 non-null    float64
22  pe          400 non-null    float64
23  ane         400 non-null    float64
24  class       400 non-null    int64  
dtypes: float64(24), int64(1)
memory usage: 78.2 KB
```

Preparation of Model Data and Scaling of Data

In this step we select subset of relevant attributes from the total give attributes. This stage helps in reducing

in this step we select features or relevant attributes from the total given attributes and stage helps in reducing the dimensionality and making the model simpler and easy to use, thus leading to short training time and high accuracy.

To obtain highly dependent features for CKD prediction we have used Correlation and dependence method. The term correlation can be defined as mutual relationship between two. In this those attributes are chosen which highly influence the occurrence of Chronic Kidney Disease.

By using the correlation it is found that some of the attributed were highly correlated to the occurrence of CKD from the total of 25 attributes.

In [12]:

```
df.head(7).T
```

Out[12]:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|--------------|-------------|-------------|-------------|----------|-------------|----------|-------------|
| age | 48.000000 | 7.000000 | 62.000000 | 48.000 | 51.000000 | 60.000 | 68.000000 |
| bp | 80.000000 | 50.000000 | 80.000000 | 70.000 | 80.000000 | 90.000 | 70.000000 |
| sg | 1.020000 | 1.020000 | 1.010000 | 1.005 | 1.010000 | 1.015 | 1.010000 |
| al | 1.000000 | 4.000000 | 2.000000 | 4.000 | 2.000000 | 3.000 | 0.000000 |
| su | 0.000000 | 0.000000 | 3.000000 | 0.000 | 0.000000 | 0.000 | 0.000000 |
| rbc | 0.000000 | 0.000000 | 0.000000 | 0.000 | 0.000000 | 0.000 | 0.000000 |
| pc | 0.000000 | 0.000000 | 0.000000 | 1.000 | 0.000000 | 0.000 | 0.000000 |
| pcc | 0.000000 | 0.000000 | 0.000000 | 1.000 | 0.000000 | 0.000 | 0.000000 |
| ba | 0.000000 | 0.000000 | 0.000000 | 0.000 | 0.000000 | 0.000 | 0.000000 |
| bgr | 121.000000 | 148.036517 | 423.000000 | 117.000 | 106.000000 | 74.000 | 100.000000 |
| bu | 36.000000 | 18.000000 | 53.000000 | 56.000 | 26.000000 | 25.000 | 54.000000 |
| sc | 1.200000 | 0.800000 | 1.800000 | 3.800 | 1.400000 | 1.100 | 24.000000 |
| sod | 137.528754 | 137.528754 | 137.528754 | 111.000 | 137.528754 | 142.000 | 104.000000 |
| pot | 4.627244 | 4.627244 | 4.627244 | 2.500 | 4.627244 | 3.200 | 4.000000 |
| hemo | 15.400000 | 11.300000 | 9.600000 | 11.200 | 11.600000 | 12.200 | 12.400000 |
| pcv | 44.000000 | 38.000000 | 31.000000 | 32.000 | 35.000000 | 39.000 | 36.000000 |
| wc | 7800.000000 | 6000.000000 | 7500.000000 | 6700.000 | 7300.000000 | 7800.000 | 8406.122449 |
| rc | 5.000000 | 4.241636 | 4.241636 | 3.000 | 4.000000 | 4.000 | 4.241636 |
| htn | 1.000000 | 0.000000 | 0.000000 | 1.000 | 0.000000 | 1.000 | 0.000000 |
| dm | 1.000000 | 0.000000 | 1.000000 | 0.000 | 0.000000 | 1.000 | 0.000000 |
| cad | 0.000000 | 0.000000 | 0.000000 | 0.000 | 0.000000 | 0.000 | 0.000000 |
| appet | 1.000000 | 1.000000 | 0.000000 | 0.000 | 1.000000 | 1.000 | 1.000000 |
| pe | 0.000000 | 0.000000 | 0.000000 | 1.000 | 0.000000 | 1.000 | 0.000000 |
| ane | 0.000000 | 0.000000 | 1.000000 | 1.000 | 0.000000 | 0.000 | 0.000000 |
| class | 1.000000 | 1.000000 | 1.000000 | 1.000 | 1.000000 | 1.000 | 1.000000 |

In [13]:

```
numerical_features = []
categorical_features = []

for i in df.columns:
    if df[i].nunique()>7:
        numerical_features.append(i)
    else:
        categorical_features.append(i)
```

In [14]:

```
numerical_features
```

Out[14]:

```
['age', 'bp', 'bgr', 'bu', 'sc', 'sod', 'pot', 'hemo', 'pcv', 'wc']
```

In [15]:

```
categorical_features
```

Out[15]:

```
['sg',
 'al',
 'su',
 'rbc',
 'pc',
 'pcc',
 'ba',
 'rc',
 'htn',
 'dm',
 'cad',
 'appet',
 'pe',
 'ane',
 'class']
```

Importing the data set

In [16]:

```
# importing or loading the dataset
#dataset = df[['sg', 'al', 'su', 'rbc', 'pc', 'pcc', 'ba', 'htn', 'dm', 'cad', 'appet', 'pe', 'ane']]
dataset = df[['sg', 'al', 'su', 'rbc', 'htn', 'dm', 'cad', 'appet', 'pe', 'ane']]

# distributing the dataset into two components X and Y
X = dataset.iloc[:, 0:5].values
y = dataset.iloc[:, 5].values
```

Splitting the dataset into the Training set and Test set

In [17]:

```
# Splitting the X and Y into the  
# Training set and Testing set  
from sklearn.model_selection import train_test_split  
  
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_st
```

Feature Scaling

Doing the pre-processing part on training and testing set such as fitting the Standard scale.

In [18]:

```
# performing preprocessing part  
from sklearn.preprocessing import StandardScaler  
sc = StandardScaler()  
  
X_train = sc.fit_transform(X_train)  
X_test = sc.transform(X_test)
```

Applying PCA function

Applying the PCA function into training and testing set for analysis.

In [19]:

```
# Applying PCA function on training  
# and testing set of X component  
from sklearn.decomposition import PCA  
  
pca = PCA(n_components = 2)  
  
X_train = pca.fit_transform(X_train)  
X_test = pca.transform(X_test)  
  
explained_variance = pca.explained_variance_ratio_
```

Fitting Logistic Regression To the training set

In [20]:

```
# Fitting Logistic Regression To the training set  
from sklearn.linear_model import LogisticRegression  
  
classifier = LogisticRegression(random_state = 0)  
classifier.fit(X_train, y_train)
```

Out[20]:

```
LogisticRegression(random_state=0)
```

Predicting the test set result

In [21]:

```
# Predicting the test set result using  
# predict function under LogisticRegression  
y_pred = classifier.predict(X_test)
```

Making the confusion matrix

In [22]:

```
# making confusion matrix between  
# test set of Y and predicted value.  
from sklearn.metrics import confusion_matrix  
  
cm = confusion_matrix(y_test, y_pred)
```

Predicting the training set result

In [23]:

```
# Predicting the training set
# result through scatter plot
from matplotlib.colors import ListedColormap

X_set, y_set = X_train, y_train
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1,
                                stop = X_set[:, 0].max() + 1, step = 0.01),
                      np.arange(start = X_set[:, 1].min() - 1,
                                stop = X_set[:, 1].max() + 1, step = 0.01))

plt.contourf(X1, X2, classifier.predict(np.array([X1.ravel(),
                                                  X2.ravel()]).T).reshape(X1.shape), alpha = 0.75,
              cmap = ListedColormap(('yellow', 'white', 'aquamarine'))

plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())

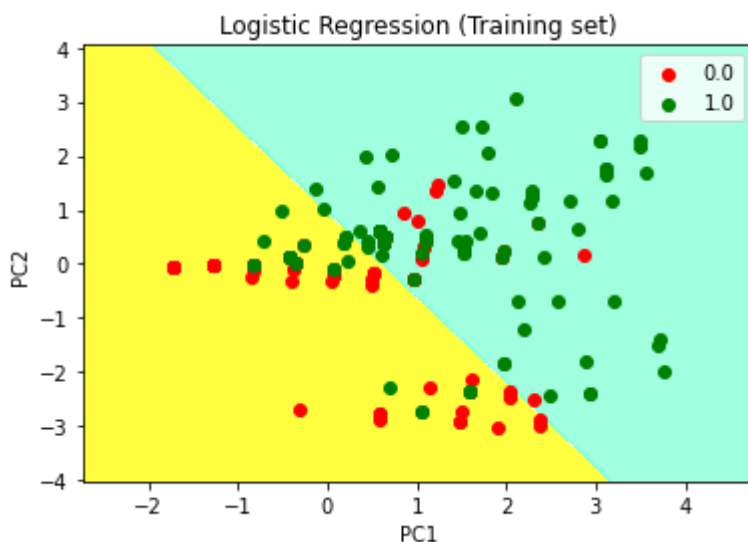
for i, j in enumerate(np.unique(y_set)):
    plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
                c = ListedColormap(('red', 'green', 'blue'))(i), label = j)

plt.title('Logistic Regression (Training set)')
plt.xlabel('PC1') # for Xlabel
plt.ylabel('PC2') # for Ylabel
plt.legend() # to show legend

# show scatter plot
plt.show()
```

c argument looks like a single numeric RGB or RGBA sequence, which should be avoided as value-mapping will have precedence in case its length matches with *x* & *y*. Please use the *color* keyword-argument or provide a 2-D array with a single row if you intend to specify the same RGB or RGBA value for all points.

c argument looks like a single numeric RGB or RGBA sequence, which should be avoided as value-mapping will have precedence in case its length matches with *x* & *y*. Please use the *color* keyword-argument or provide a 2-D array with a single row if you intend to specify the same RGB or RGBA value for all points.



Visualising the Test set results

In [24]:

```
# Visualising the Test set results through scatter plot
from matplotlib.colors import ListedColormap

X_set, y_set = X_test, y_test

X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1,
                               stop = X_set[:, 0].max() + 1, step = 0.01),
                     np.arange(start = X_set[:, 1].min() - 1,
                               stop = X_set[:, 1].max() + 1, step = 0.01))

plt.contourf(X1, X2, classifier.predict(np.array([X1.ravel(),
                                                  X2.ravel()]).T).reshape(X1.shape), alpha = 0.75,
             cmap = ListedColormap(('yellow', 'white', 'aquamarine'))

plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())

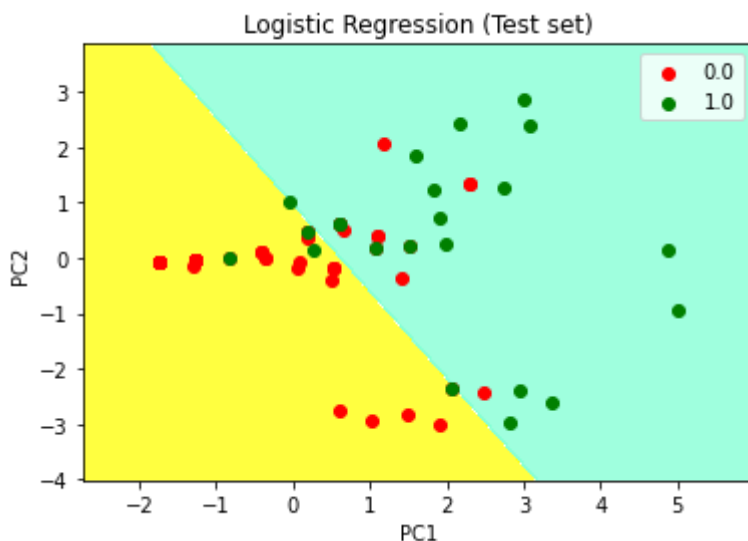
for i, j in enumerate(np.unique(y_set)):
    plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
               c = ListedColormap(('red', 'green', 'blue'))(i), label = j)

# title for scatter plot
plt.title('Logistic Regression (Test set)')
plt.xlabel('PC1') # for Xlabel
plt.ylabel('PC2') # for Ylabel
plt.legend()

# show scatter plot
plt.show()
```

c argument looks like a single numeric RGB or RGBA sequence, which should be avoided as value-mapping will have precedence in case its length matches with *x* & *y*. Please use the *color* keyword-argument or provide a 2-D array with a single row if you intend to specify the same RGB or RGBA value for all points.

c argument looks like a single numeric RGB or RGBA sequence, which should be avoided as value-mapping will have precedence in case its length matches with *x* & *y*. Please use the *color* keyword-argument or provide a 2-D array with a single row if you intend to specify the same RGB or RGBA value for all points.



Principal Component Analysis (PCA) is an unsupervised, non-parametric statistical technique primarily used for

dimensionality reduction in machine learning.

PCA is based on the Pearson correlation coefficient framework and inherits similar assumptions. PCA is a handy addition to the data scientist toolkit and will improve model performance in most scenarios.

Advantages of Dimensionality Reduction

1. It helps in data compression, and hence reduced storage space.
2. It reduces computation time.
3. It also helps remove redundant features, if any.
4. Disadvantages of Dimensionality Reduction

It may lead to some amount of data loss.

1. PCA tends to find linear correlations between variables, which is sometimes undesirable.
2. PCA fails in cases where mean and covariance are not enough to define datasets.
3. We may not know how many principal components to keep- in practice, some thumb rules are applied.

LDA for Dimensionality

Importing or loading the dataset

In [25]:

```
# importing or loading the dataset
#dataset = df[['sg', 'al', 'su', 'rbc', 'pc', 'pcc', 'ba', 'htn', 'dm', 'cad', 'appet', 'pe', 'ane']]
dataset = df[['sg', 'al', 'su', 'rbc', 'htn', 'dm', 'cad', 'appet', 'pe', 'ane']]

# distributing the dataset into two components X and Y
X = dataset.iloc[:, 0:5].values
y = dataset.iloc[:, 5].values
```

In [26]:

```
#3. Split the dataset into Training set and Test set
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 42)
```

In [27]:

```
#4. Feature Scaling
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
```

In [28]:

```
#5. Apply LDA
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
lda = LDA(n_components = 1)
X_train = lda.fit_transform(X_train, y_train)
X_test = lda.transform(X_test)
```

In [29]:

```
#6. Fit Logistic Regression to the Training set
from sklearn.linear_model import LogisticRegression
classifier = LogisticRegression(random_state = 0)
classifier.fit(X_train, y_train)
```

Out[29]:

```
LogisticRegression(random_state=0)
```

In [30]:

```
#7. Predict the Test set results
y_pred = classifier.predict(X_test)
```

In [31]:

```
#8. Make the Confusion Matrix
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)
```

In [32]:

```
#9. Visualize the Training set results
from matplotlib.colors import ListedColormap
X_set, y_set = X_train, y_train

X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1,
                               stop = X_set[:, 0].max() + 1, step = 0.01),
                    np.arange(start = X_set[:, 1].min() - 1,
                               stop = X_set[:, 1].max() + 1, step = 0.01))
```

In [33]:

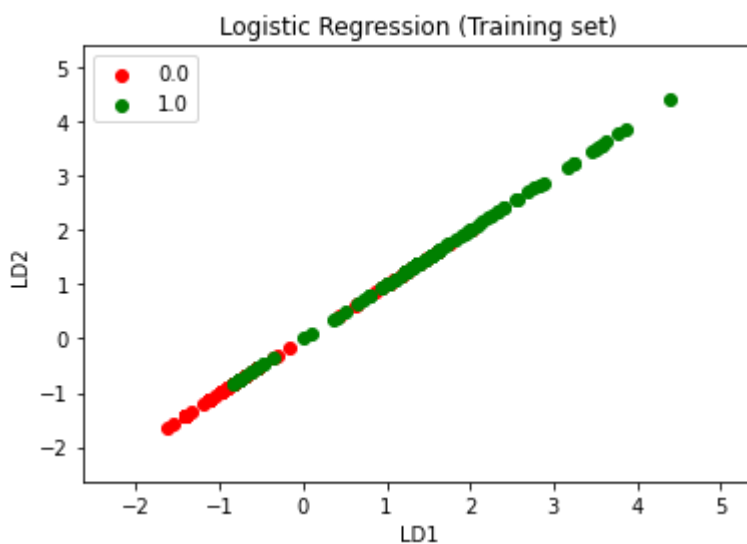
```
# plt.contourf(X1, X2, classifier.predict(np.array([X1.ravel(),
#           X2.ravel()])).T).reshape(X1.shape), alpha = 0.75,
#           cmap = ListedColormap(('yellow', 'white', 'aquamarine'))

plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())

for i, j in enumerate(np.unique(y_set)):
    plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
                c = ListedColormap(('red', 'green', 'blue'))(i), label = j)
plt.title('Logistic Regression (Training set)')
plt.xlabel('LD1')
plt.ylabel('LD2')
plt.legend()
plt.show()
```

c argument looks like a single numeric RGB or RGBA sequence, which should be avoided as value-mapping will have precedence in case its length matches with *x* & *y*. Please use the *color* keyword-argument or provide a 2-D array with a single row if you intend to specify the same RGB or RGBA value for all points.

c argument looks like a single numeric RGB or RGBA sequence, which should be avoided as value-mapping will have precedence in case its length matches with *x* & *y*. Please use the *color* keyword-argument or provide a 2-D array with a single row if you intend to specify the same RGB or RGBA value for all points.



In [34]:

```
import matplotlib.pyplot as plt

from sklearn import datasets
from sklearn.decomposition import PCA
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

iris = datasets.load_iris()

X = iris.data
y = iris.target
target_names = iris.target_names

pca = PCA(n_components=2)
X_r = pca.fit(X).transform(X)

lda = LinearDiscriminantAnalysis(n_components=2)
X_r2 = lda.fit(X, y).transform(X)

# Percentage of variance explained for each components
print('explained variance ratio (first two components): %s'
      % str(pca.explained_variance_ratio_))

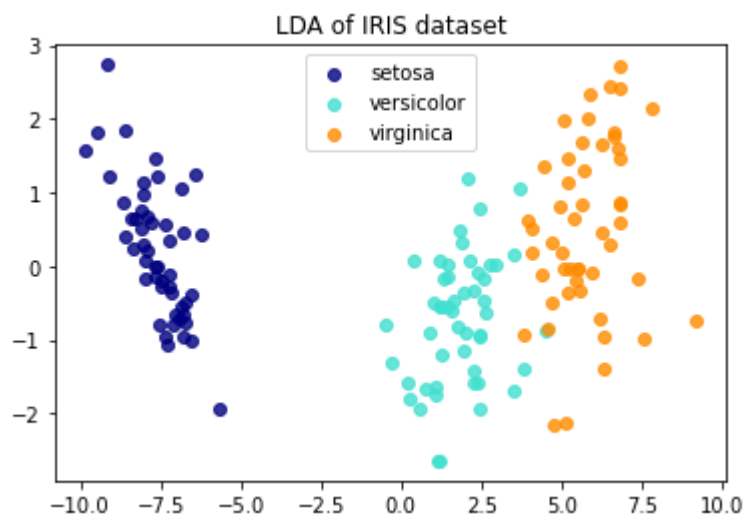
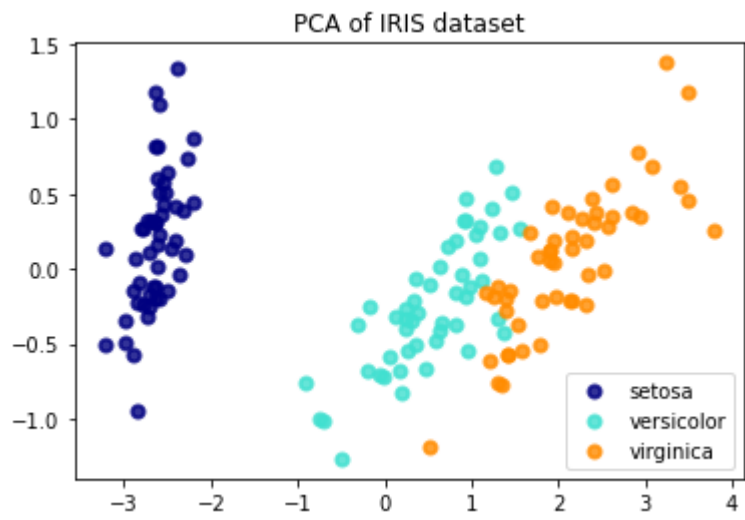
plt.figure()
colors = ['navy', 'turquoise', 'darkorange']
lw = 2

for color, i, target_name in zip(colors, [0, 1, 2], target_names):
    plt.scatter(X_r[y == i, 0], X_r[y == i, 1], color=color, alpha=.8, lw=lw,
                label=target_name)
plt.legend(loc='best', shadow=False, scatterpoints=1)
plt.title('PCA of IRIS dataset')

plt.figure()
for color, i, target_name in zip(colors, [0, 1, 2], target_names):
    plt.scatter(X_r2[y == i, 0], X_r2[y == i, 1], alpha=.8, color=color,
                label=target_name)
plt.legend(loc='best', shadow=False, scatterpoints=1)
plt.title('LDA of IRIS dataset')

plt.show()
```

```
explained variance ratio (first two components): [0.92461872 0.0530664
8]
```

In []: