

**BOWEN UNIVERSITY, IWO**

**COLLEGE OF COMPUTING AND COMMUNICATION STUDIES**

**DEPARTMENT OF COMPUTER SCIENCE**

**CSC 404- MACHINE LEARNING WITH PYTHON**

**LECTURE NOTE 11**

**Introduction to Logistic Regression**

****

Logistic regression is a supervised learning classification algorithm used to predict the probability of a target variable. The nature of target or dependent variable is dichotomous, which means there would be only two possible classes.

In simple words, the dependent variable is binary in nature having data coded as either 1 (stands for success/yes) or 0 (stands for failure/no).

Mathematically, a logistic regression model predicts P(Y=1) as a function of X. It is one of the simplest ML algorithms that can be used for various classification problems such as spam detection, Diabetes prediction, cancer detection etc.

**Types of Logistic Regression**

****

Generally, logistic regression means binary logistic regression having binary target variables, but there can be two more categories of target variables that can be predicted by it. Based on those number of categories, Logistic regression can be divided into following types:

**Binary or Binomial**

In such a kind of classification, a dependent variable will have only two possible types either 1 and 0. For example, these variables may represent success or failure, yes or no, win or loss etc.

**Multinomial**

In such a kind of classification, dependent variable can have 3 or more possible ***unordered*** types or the types having no quantitative significance. For example, these variables may represent “Type A” or “Type B” or “Type C”.

**Ordinal**

In such a kind of classification, dependent variable can have 3 or more possible ***ordered*** types or the types having a quantitative significance. For example, these variables may represent “poor” or “good”, “very good”, “Excellent” and each category can have the scores like 0,1,2,3.

**Logistic Regression Assumptions**

****

Before diving into the implementation of logistic regression, we must be aware of the following assumptions about the same:

* In case of binary logistic regression, the target variables must be binary always and the desired outcome is represented by the factor level 1.
* There should not be any multi-collinearity in the model, which means the independent variables must be independent of each other.
* We must include meaningful variables in our model.
* We should choose a large sample size for logistic regression.

**Binary Logistic Regression model**

****

The simplest form of logistic regression is binary or binomial logistic regression in which the target or dependent variable can have only 2 possible types either 1 or 0. It allows us to model a relationship between multiple predictor variables and a binary/binomial target variable. In case of logistic regression, the linear function is basically used as an input to another function such as in the following relation:

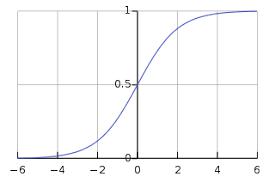
ℎ ( ) = ( ) ℎ 0 ≤ ℎ ≤ 1

Here, is the logistic or sigmoid function which can be given as follows:

1

( ) = 1 + −  ℎ =

To sigmoid curve can be represented with the help of following graph. We can see the values of y-axis lie between 0 and 1 and crosses the axis at 0.5.



The classes can be divided into positive or negative. The output comes under the probability of positive class if it lies between 0 and 1. For our implementation, we are interpreting the output of hypothesis function as positive if it is ≥ 0.5, otherwise negative.

We also need to define a loss function to measure how well the algorithm performs using the weights on functions, represented by theta as follows:

ℎ = ( )

1

( ) = . (− log(ℎ) − (1 − ) log(1 − ℎ))

Now, after defining the loss function our prime goal is to minimize the loss function. It can be done with the help of fitting the weights which means by increasing or decreasing the weights. With the help of derivatives of the loss function w.r.t each weight, we would be able to know what parameters should have high weight and what should have smaller weight.

The following gradient descent equation tells us how loss would change if we modified the parameters:

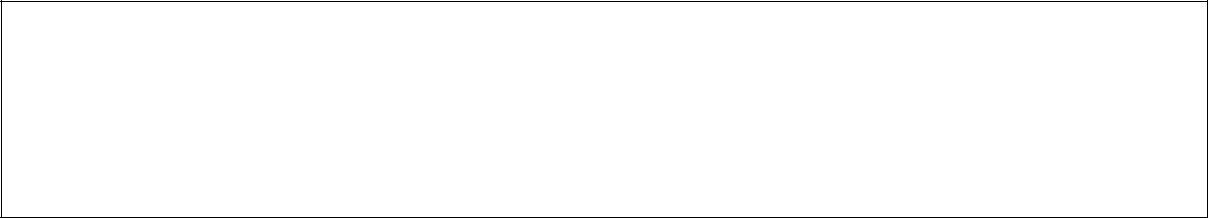
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ( ) | | = | 1 | ( ( ) − ) |
|  |  |  |
|  | |  |  |  |

**Implementation in Python**

****

Now we will implement the above concept of binomial logistic regression in Python. For this purpose, we are using a multivariate flower dataset named ‘iris’ which have 3 classes of 50 instances each, but we will be using the first two feature columns. Every class represents a type of iris flower.

First, we need to import the necessary libraries as follows:



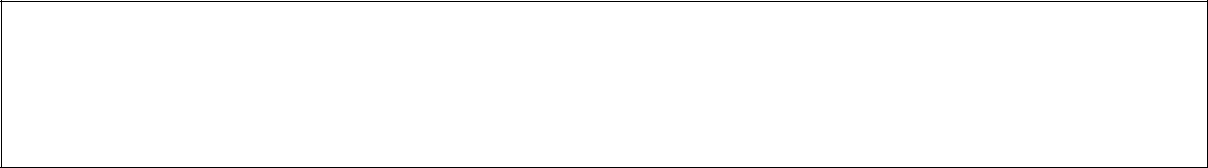
import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn import datasets

Next, load the iris dataset as follows:

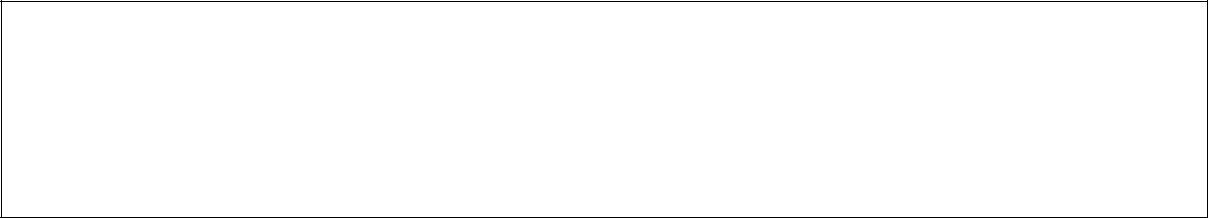


iris = datasets.load\_iris()

X = iris.data[:, :2]

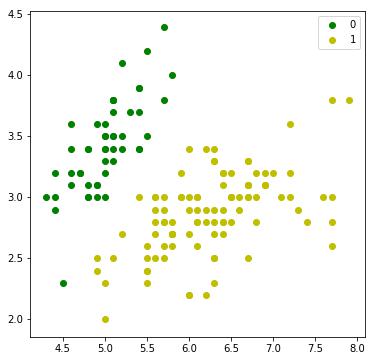
y = (iris.target != 0) \* 1

We can plot our training data s follows:



plt.figure(figsize=(6, 6))

plt.scatter(X[y == 0][:, 0], X[y == 0][:, 1], color='g', label='0') plt.scatter(X[y == 1][:, 0], X[y == 1][:, 1], color='y', label='1') plt.legend();



Next, we will define sigmoid function, loss function and gradient descend as follows:



class LogisticRegression:

def \_\_init\_\_(self, lr=0.01, num\_iter=100000, fit\_intercept=True, verbose=False):

self.lr = lr

self.num\_iter = num\_iter

self.fit\_intercept = fit\_intercept

self.verbose = verbose

def \_\_add\_intercept(self, X):

intercept = np.ones((X.shape[0], 1))

return np.concatenate((intercept, X), axis=1)

def \_\_sigmoid(self, z):

return 1 / (1 + np.exp(-z))

def \_\_loss(self, h, y):

return (-y \* np.log(h) - (1 - y) \* np.log(1 - h)).mean()

def fit(self, X, y):

if self.fit\_intercept:

X = self.\_\_add\_intercept(X)

Now, initialize the weights as follows:



self.theta = np.zeros(X.shape[1])

for i in range(self.num\_iter):

z = np.dot(X, self.theta)

h = self.\_\_sigmoid(z)

gradient = np.dot(X.T, (h - y)) / y.size

self.theta -= self.lr \* gradient

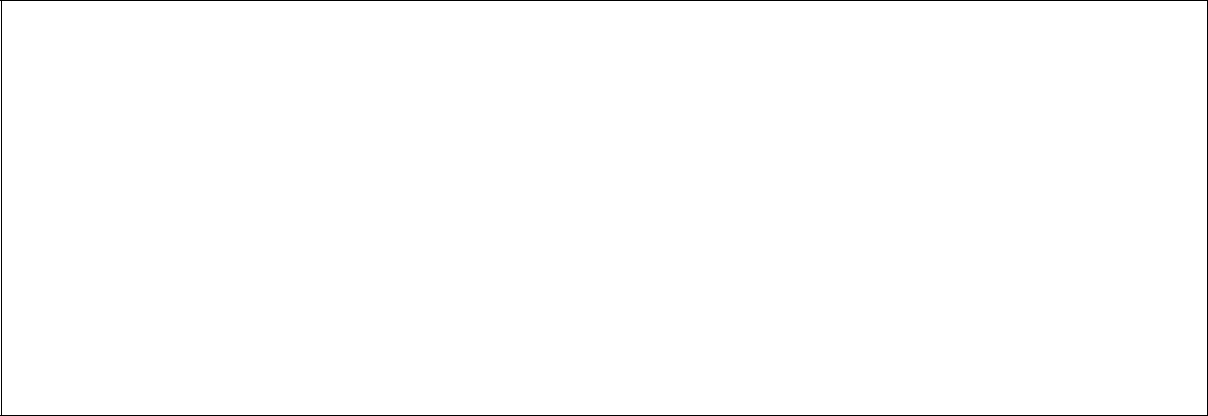
z = np.dot(X, self.theta)

1. = self.\_\_sigmoid(z) loss = self.\_\_loss(h, y)

if(self.verbose ==True and i % 10000 == 0):

print(f'loss: {loss} \t')

With the help of the following script, we can predict the output probabilities:



def predict\_prob(self, X):

if self.fit\_intercept:

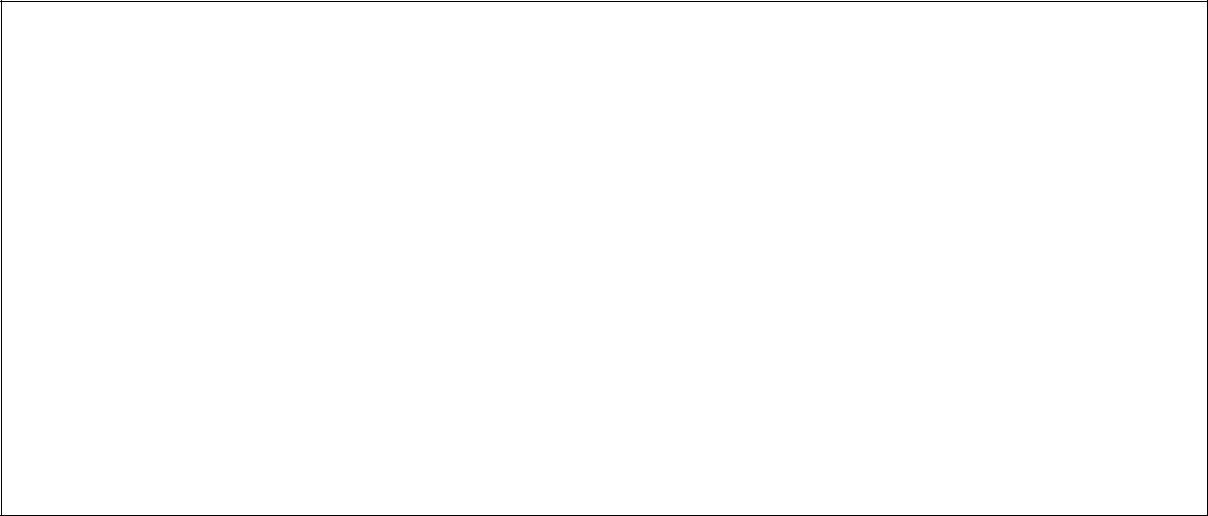
X = self.\_\_add\_intercept(X)

return self.\_\_sigmoid(np.dot(X, self.theta))

def predict(self, X):

return self.predict\_prob(X).round()

Next, we can evaluate the model and plot it as follows:



model = LogisticRegression(lr=0.1, num\_iter=300000)

preds = model.predict(X)

(preds == y).mean()

plt.figure(figsize=(10, 6))

plt.scatter(X[y == 0][:, 0], X[y == 0][:, 1], color='g', label='0')

plt.scatter(X[y == 1][:, 0], X[y == 1][:, 1], color='y', label='1')

plt.legend()

x1\_min, x1\_max = X[:,0].min(), X[:,0].max(),

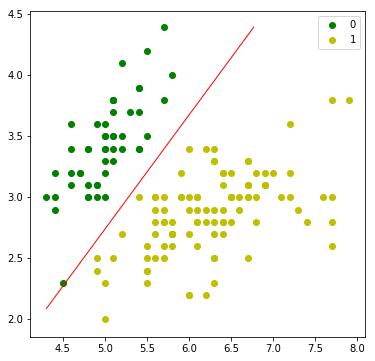
x2\_min, x2\_max = X[:,1].min(), X[:,1].max(),

xx1, xx2 = np.meshgrid(np.linspace(x1\_min, x1\_max), np.linspace(x2\_min, x2\_max))

grid = np.c\_[xx1.ravel(), xx2.ravel()]

probs = model.predict\_prob(grid).reshape(xx1.shape)

plt.contour(xx1, xx2, probs, [0.5], linewidths=1, colors='red');



**Multinomial Logistic Regression Model**

****

Another useful form of logistic regression is multinomial logistic regression in which the target or dependent variable can have 3 or more possible ***unordered*** types i.e. the types having no quantitative significance.

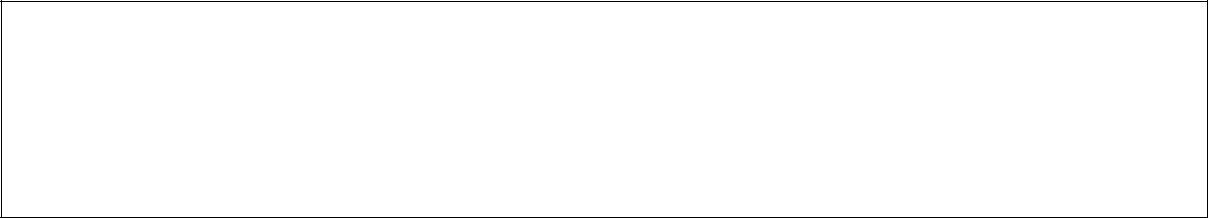
**Implementation in Python**

****

Now we will implement the above concept of multinomial logistic regression in Python.

For this purpose, we are using a dataset from **sklearn** named ***digit***.

First, we need to import the necessary libraries as follows:



Import sklearn

from sklearn import datasets

from sklearn import linear\_model

from sklearn import metrics

from sklearn.model\_selection import train\_test\_split

Next, we need to load digit dataset**:**

****

digits = datasets.load\_digits()

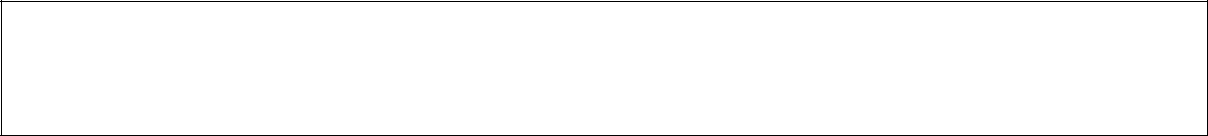
Now, define the feature matrix(X) and response vector(y)as follows:



X = digits.data

y = digits.target

With the help of next line of code, we can split X and y into training and testing sets:



X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y,

test\_size=0.4, random\_state= 1)

Now create an object of logistic regression as follows:



digreg = linear\_model.LogisticRegression()

Now, we need to train the model by using the training sets as follows:



digreg.fit(X\_train, y\_train)

Next, make the predictions on testing set as follows:



y\_pred = digreg.predict(X\_test)

Next print the accuracy of the model as follows:



print("Accuracy of Logistic Regression model is:", metrics.accuracy\_score(y\_test, y\_pred)\*100)

**Output**

****

Accuracy of Logistic Regression model is: 95.6884561891516

From the above output we can see the accuracy of our model is around 96 percent.

**Introduction to SVM**

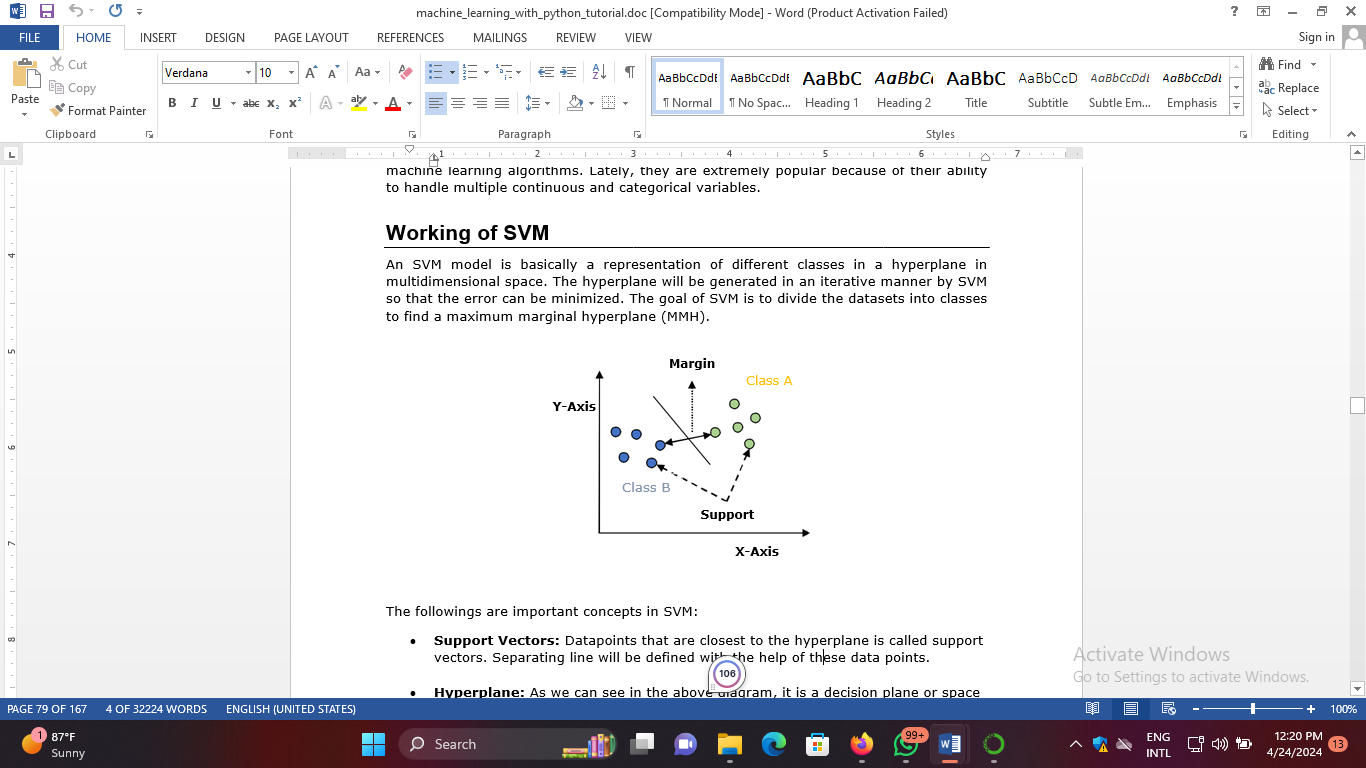
****

Support vector machines (SVMs) are powerful yet flexible supervised machine learning algorithms which are used both for classification and regression. But generally, they are used in classification problems. In 1960s, SVMs were first introduced but later they got refined in 1990. SVMs have their unique way of implementation as compared to other machine learning algorithms. Lately, they are extremely popular because of their ability to handle multiple continuous and categorical variables.

**Working of SVM**

****

An SVM model is basically a representation of different classes in a hyperplane in multidimensional space. The hyperplane will be generated in an iterative manner by SVM so that the error can be minimized. The goal of SVM is to divide the datasets into classes to find a maximum marginal hyperplane (MMH).



The followings are important concepts in SVM:

* **Support Vectors:** Datapoints that are closest to the hyperplane is called support vectors. Separating line will be defined with the help of these data points.
* **Hyperplane:** As we can see in the above diagram, it is a decision plane or space which is divided between a set of objects having different classes.
* **Margin:** It may be defined as the gap between two lines on the closet data points of different classes. It can be calculated as the perpendicular distance from the line to the support vectors. Large margin is considered as a good margin and small margin is considered as a bad margin.

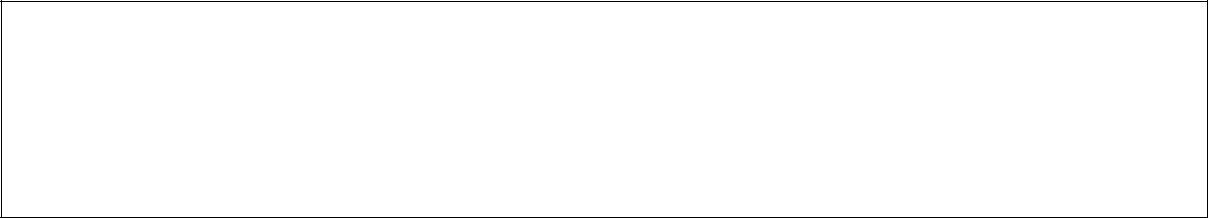
The main goal of SVM is to divide the datasets into classes to find a maximum marginal hyperplane (MMH) and it can be done in the following two steps:

* First, SVM will generate hyperplanes iteratively that segregates the classes in best way.
* Then, it will choose the hyperplane that separates the classes correctly.

**Implementing SVM in Python**

****

For implementing SVM in Python we will start with the standard libraries import as follows:



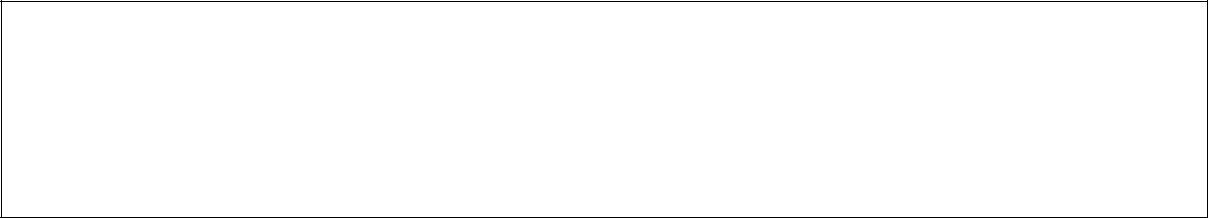
import numpy as np

import matplotlib.pyplot as plt

from scipy import stats

import seaborn as sns; sns.set()

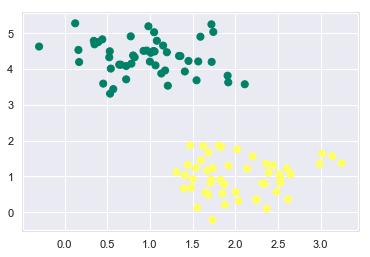
Next, we are creating a sample dataset, having linearly separable data, from sklearn.dataset.sample\_generator for classification using SVM:



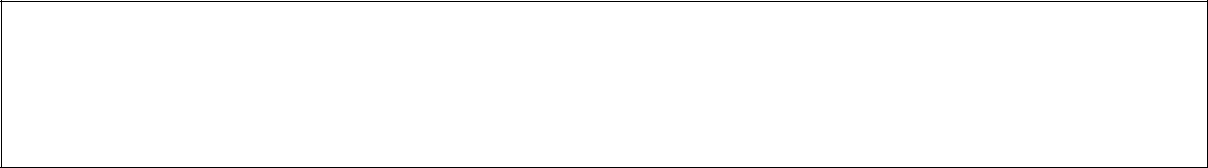
from sklearn.datasets.samples\_generator import make\_blobs X, y = make\_blobs(n\_samples=100, centers=2,

random\_state=0, cluster\_std=0.50) plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='summer');

The following would be the output after generating sample dataset having 100 samples and 2 clusters:



We know that SVM supports discriminative classification. it divides the classes from each other by simply finding a line in case of two dimensions or manifold in case of multiple dimensions. It is implemented on the above dataset as follows:



xfit = np.linspace(-1, 3.5)

plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='summer')

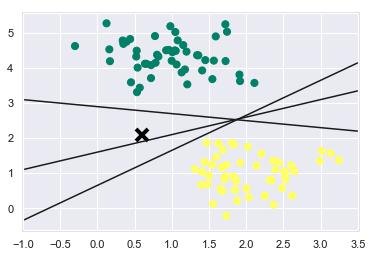
plt.plot([0.6], [2.1], 'x', color='black', markeredgewidth=4, markersize=12)

for m, b in [(1, 0.65), (0.5, 1.6), (-0.2, 2.9)]:

plt.plot(xfit, m \* xfit + b, '-k')

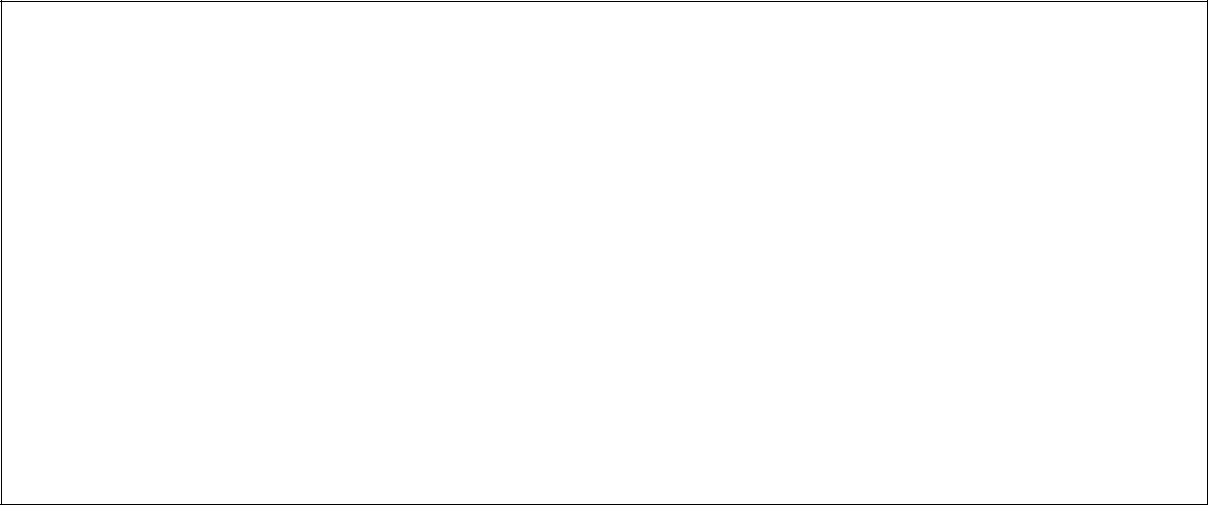
plt.xlim(-1, 3.5);

The output is as follows:



We can see from the above output that there are three different separators that perfectly discriminate the above samples.

As discussed, the main goal of SVM is to divide the datasets into classes to find a maximum marginal hyperplane (MMH) hence rather than drawing a zero line between classes we can draw around each line a margin of some width up to the nearest point. It can be done as follows:



**xfit = np.linspace(-1, 3.5)**

**plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='summer')**

**for m, b, d in [(1, 0.65, 0.33), (0.5, 1.6, 0.55), (-0.2, 2.9, 0.2)]:**

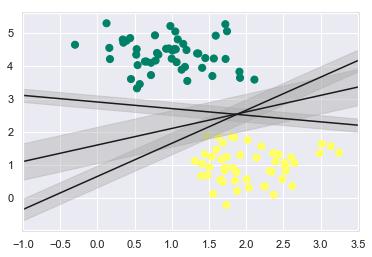
**yfit = m \* xfit + b**

**plt.plot(xfit, yfit, '-k')**

**plt.fill\_between(xfit, yfit - d, yfit + d, edgecolor='none',**

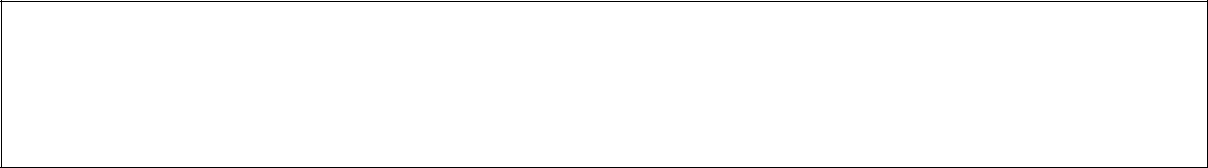
**color='#AAAAAA', alpha=0.4)**

**plt.xlim(-1, 3.5);**



From the above image in output, we can easily observe the “margins” within the discriminative classifiers. SVM will choose the line that maximizes the margin.

Next, we will use Scikit-Learn’s support vector classifier to train an SVM model on this data. Here, we are using linear kernel to fit SVM as follows:

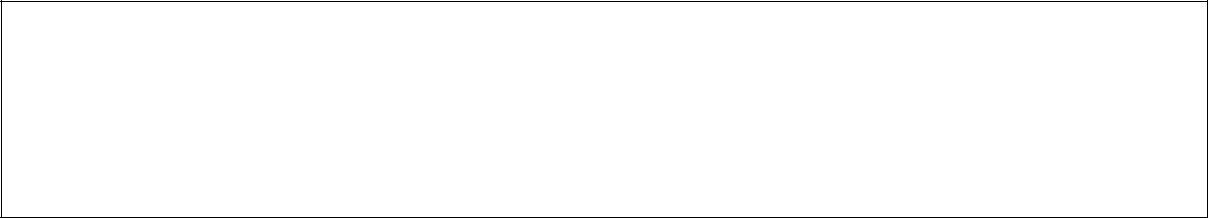


from sklearn.svm import SVC # "Support vector classifier"

model = SVC(kernel='linear', C=1E10)

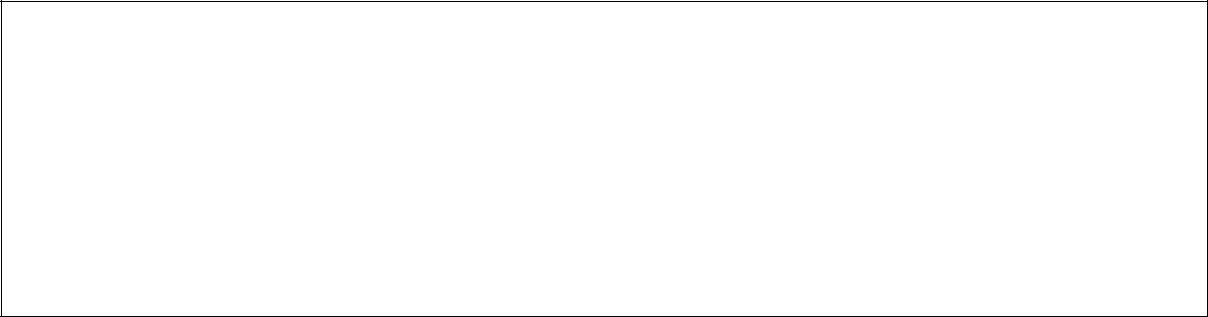
model.fit(X, y)

The output is as follows:



SVC(C=10000000000.0, cache\_size=200, class\_weight=None, coef0=0.0, decision\_function\_shape='ovr', degree=3, gamma='auto\_deprecated', kernel='linear', max\_iter=-1, probability=False, random\_state=None, shrinking=True, tol=0.001, verbose=False)

Now, for a better understanding, the following will plot the decision functions for 2D SVC:



def decision\_function(model, ax=None, plot\_support=True):

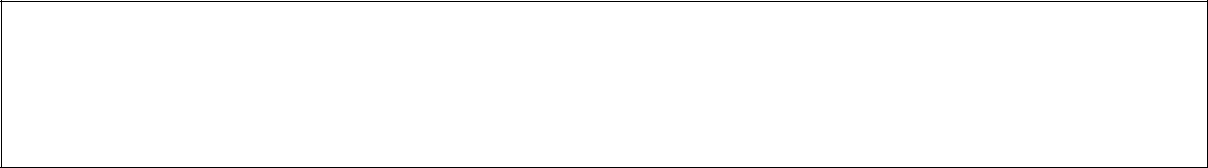
if ax is None:

ax = plt.gca()

xlim = ax.get\_xlim()

ylim = ax.get\_ylim()

For evaluating model, we need to create grid as follows:



x = np.linspace(xlim[0], xlim[1], 30)

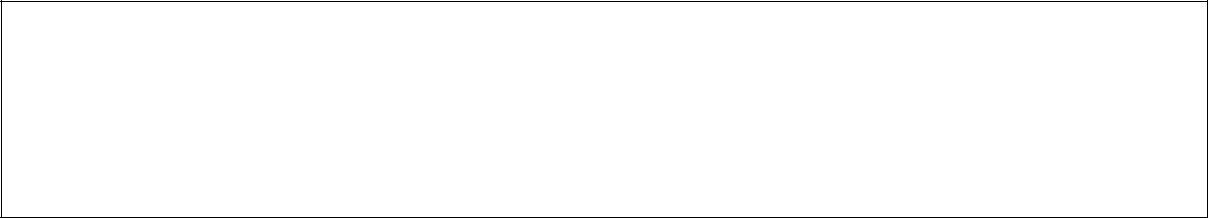
y = np.linspace(ylim[0], ylim[1], 30)

Y, X = np.meshgrid(y, x)

xy = np.vstack([X.ravel(), Y.ravel()]).T

P = model.decision\_function(xy).reshape(X.shape)

Next, we need to plot decision boundaries and margins as follows:

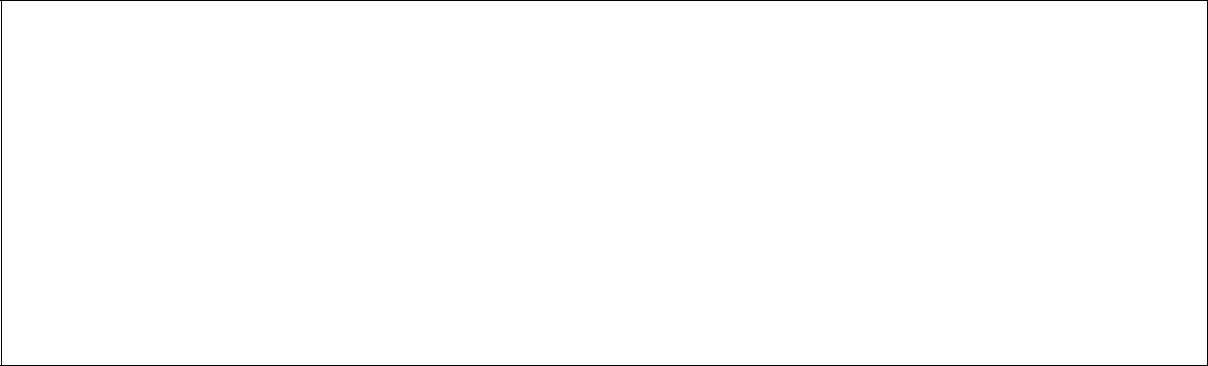


ax.contour(X, Y, P, colors='k',

levels=[-1, 0, 1], alpha=0.5,

linestyles=['--', '-', '--'])

Now, similarly plot the support vectors as follows:



if plot\_support:

ax.scatter(model.support\_vectors\_[:, 0],

model.support\_vectors\_[:, 1],

s=300, linewidth=1, facecolors='none'); ax.set\_xlim(xlim)

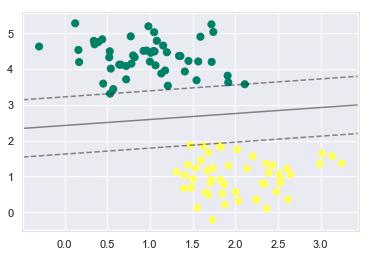
ax.set\_ylim(ylim)

Now, use this function to fit our models as follows:



plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='summer')

decision\_function(model);

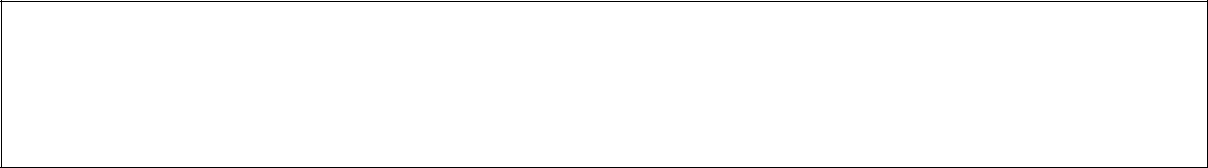


We can observe from the above output that an SVM classifier fit to the data with margins i.e. dashed lines and support vectors, the pivotal elements of this fit, touching the dashed line. These support vector points are stored in the **support\_vectors\_** attribute of the classifier as follows:



model.support\_vectors\_

The output is as follows:



array([[0.5323772 , 3.31338909],

[2.11114739, 3.57660449],

[1.46870582, 1.86947425]])

**SVM Kernels**

****

In practice, SVM algorithm is implemented with kernel that transforms an input data space into the required form. SVM uses a technique called the kernel trick in which kernel takes a low dimensional input space and transforms it into a higher dimensional space. In simple words, kernel converts non-separable problems into separable problems by adding more dimensions to it. It makes SVM more powerful, flexible and accurate. The following are some of the types of kernels used by SVM:

**Linear Kernel**

It can be used as a dot product between any two observations. The formula of linear kernel is as below:

( , ) = ( ∗ )

From the above formula, we can see that the product between two vectors say & is the sum of the multiplication of each pair of input values.

**Polynomial Kernel**

It is more generalized form of linear kernel and distinguish curved or nonlinear input space.

Following is the formula for polynomial kernel:

*K(x, xi) = 1 + sum(x \* xi)^d*

Here d is the degree of polynomial, which we need to specify manually in the learning algorithm.

**Radial Basis Function (RBF) Kernel**

RBF kernel, mostly used in SVM classification, maps input space in indefinite dimensional space. Following formula explains it mathematically:

*K(x,xi) = exp(-gamma \* sum((x – xi^2))*

Here, *gamma* ranges from 0 to 1. We need to manually specify it in the learning algorithm.

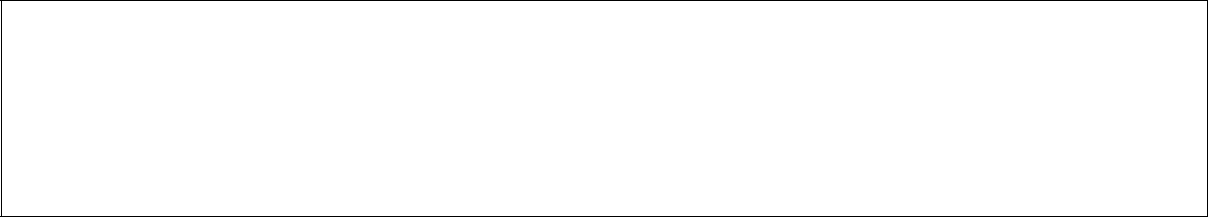
A good default value of *gamma* is 0.1.

As we implemented SVM for linearly separable data, we can implement it in Python for the data that is not linearly separable. It can be done by using kernels.

**Example**

The following is an example for creating an SVM classifier by using kernels. We will be using **iris** dataset from **scikit-learn**:

We will start by importing following packages:



import pandas as pd

import numpy as np

from sklearn import svm, datasets

import matplotlib.pyplot as plt

Now, we need to load the input data:



iris = datasets.load\_iris()

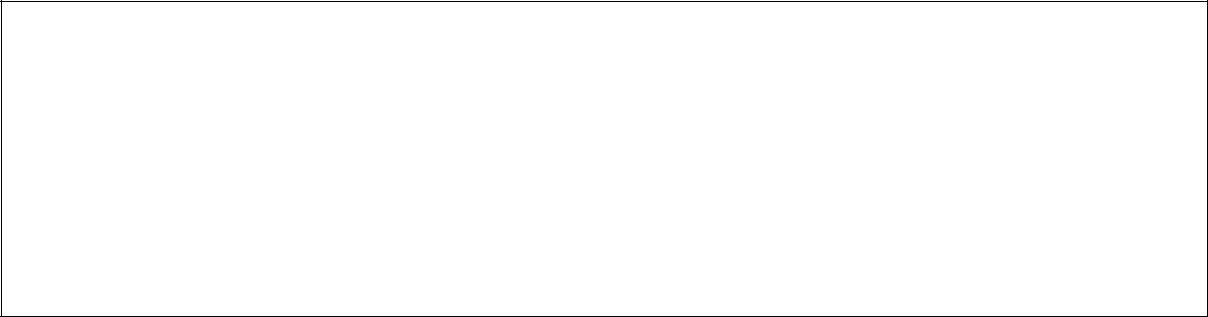
From this dataset, we are taking first two features as follows:



X = iris.data[:, :2]

y = iris.target

Next, we will plot the SVM boundaries with original data as follows:



x\_min, x\_max = X[:, 0].min() - 1, X[:, 0].max() + 1 y\_min, y\_max = X[:, 1].min() - 1, X[:, 1].max() + 1 h = (x\_max / x\_min)/100

xx, yy = np.meshgrid(np.arange(x\_min, x\_max, h), np.arange(y\_min, y\_max, h))

X\_plot = np.c\_[xx.ravel(), yy.ravel()]

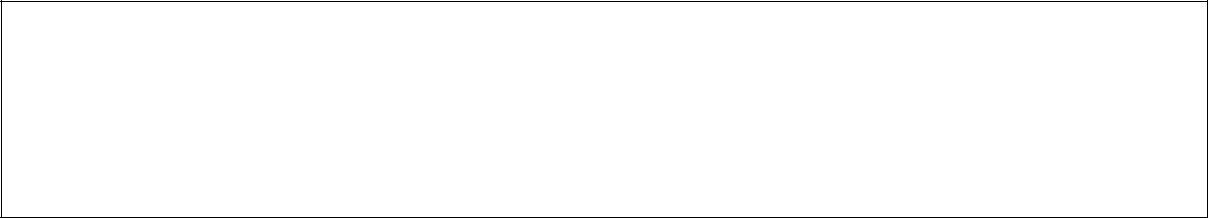
Now, we need to provide the value of regularization parameter as follows:



C=1.0

Next, SVM classifier object can be created as follows:

Svc\_classifier = svm.SVC(kernel='linear', C=C).fit(X, y)



1. = svc\_classifier.predict(X\_plot) Z = Z.reshape(xx.shape) plt.figure(figsize=(15, 5)) plt.subplot(121)

plt.contourf(xx, yy, Z, cmap=plt.cm.tab10, alpha=0.3)

plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Set1)

plt.xlabel('Sepal length')

plt.ylabel('Sepal width')

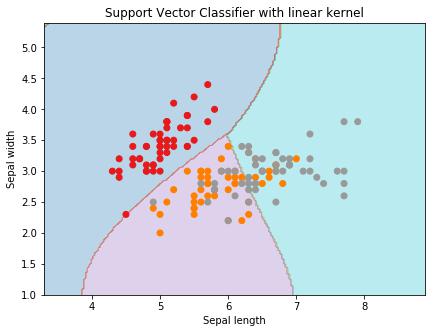
plt.xlim(xx.min(), xx.max())

plt.title('Support Vector Classifier with linear kernel')

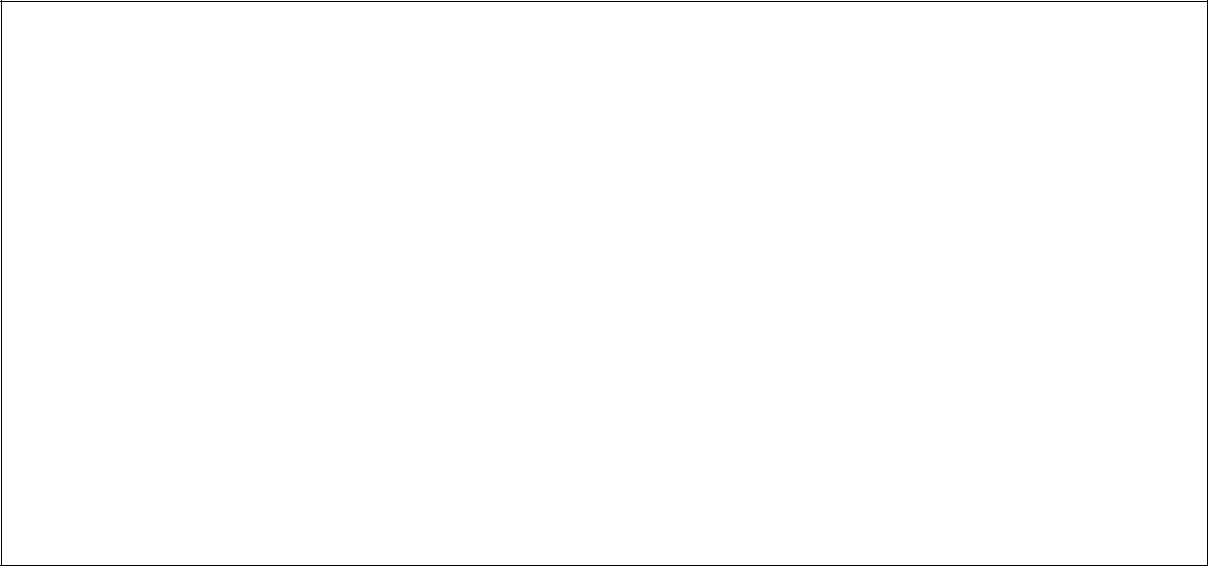
**Output**

****

Text(0.5, 1.0, 'Support Vector Classifier with linear kernel')



For creating SVM classifier with **rbf** kernel, we can change the kernel to **rbf** as follows:



Svc\_classifier = svm.SVC(kernel='rbf', gamma =‘auto’,C=C).fit(X, y)

1. = svc\_classifier.predict(X\_plot) Z = Z.reshape(xx.shape) plt.figure(figsize=(15, 5)) plt.subplot(121)

plt.contourf(xx, yy, Z, cmap=plt.cm.tab10, alpha=0.3) plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Set1) plt.xlabel('Sepal length')

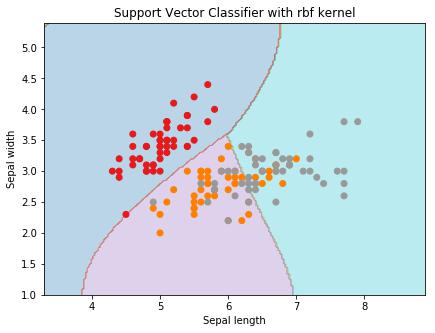
plt.ylabel('Sepal width') plt.xlim(xx.min(), xx.max())

plt.title('Support Vector Classifier with rbf kernel')

**Output**

****

Text(0.5, 1.0, 'Support Vector Classifier with rbf kernel')



We put the value of gamma to ‘auto’ but you can provide its value between 0 to 1 also.

**Pros and Cons of SVM Classifiers**

****

**Pros of SVM classifiers**

SVM classifiers offers great accuracy and work well with high dimensional space. SVM classifiers basically use a subset of training points hence in result uses very less memory.

**Cons of SVM classifiers**

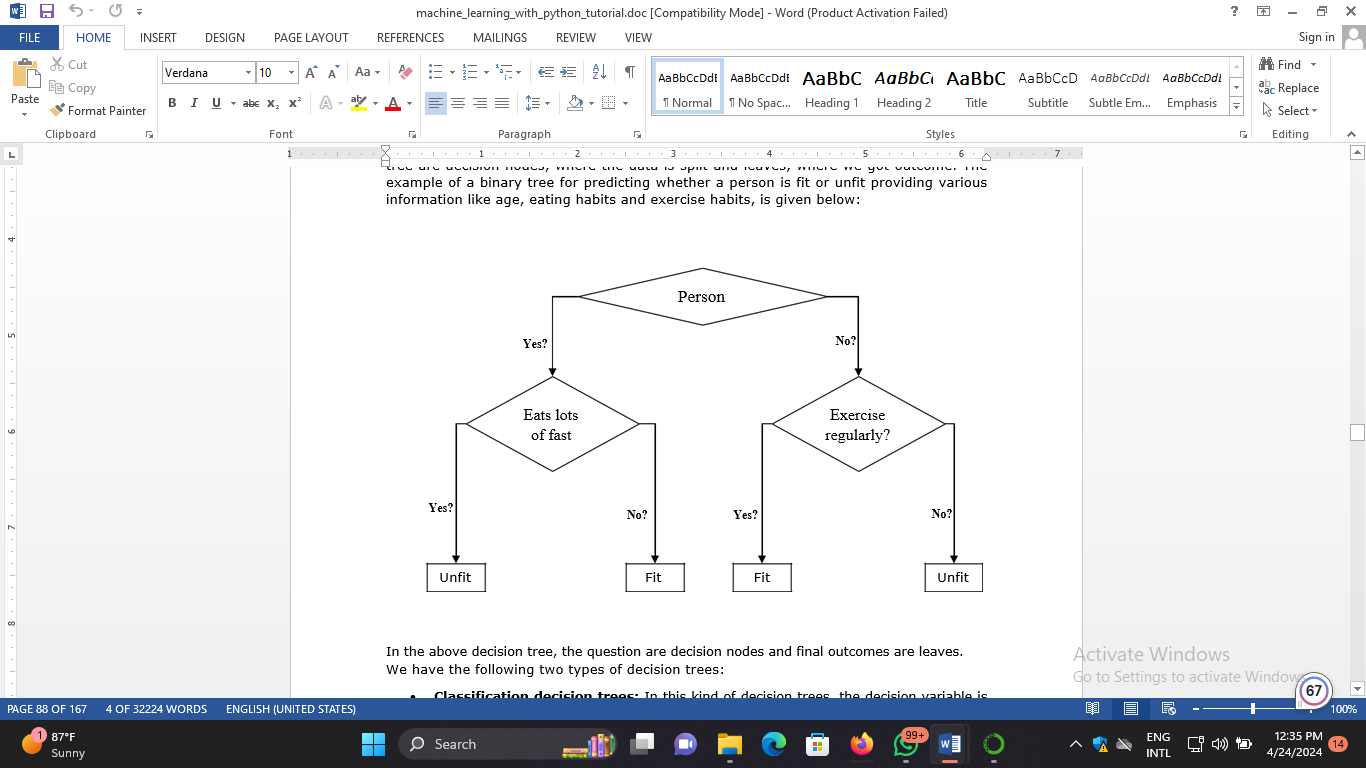
They have high training time hence in practice not suitable for large datasets. Another disadvantage is that SVM classifiers do not work well with overlapping classes.

**Introduction to Decision Tree**

****

In general, Decision tree analysis is a predictive modelling tool that can be applied across many areas. Decision trees can be constructed by an algorithmic approach that can split the dataset in different ways based on different conditions. Decisions tress are the most powerful algorithms that falls under the category of supervised algorithms.

They can be used for both classification and regression tasks. The two main entities of a tree are decision nodes, where the data is split and leaves, where we got outcome. The example of a binary tree for predicting whether a person is fit or unfit providing various information like age, eating habits and exercise habits, is given below:



In the above decision tree, the question are decision nodes and final outcomes are leaves.

We have the following two types of decision trees:

* **Classification decision trees:** In this kind of decision trees, the decision variable is categorical. The above decision tree is an example of classification decision tree.
* **Regression decision trees:** In this kind of decision trees, the decision variable is continuous.

**Implementing Decision Tree Algorithm**

****

**Gini Index**

It is the name of the cost function that is used to evaluate the binary splits in the dataset and works with the categorial target variable “Success” or “Failure”.

Higher the value of Gini index, higher the homogeneity. A perfect Gini index value is 0 and worst is 0.5 (for 2 class problem). Gini index for a split can be calculated with the help of following steps:

* First, calculate Gini index for sub-nodes by using the formula *p^2+q^2* , which is the sum of the square of probability for success and failure.
* Next, calculate Gini index for split using weighted Gini score of each node of that split.

Classification and Regression Tree (CART) algorithm uses Gini method to generate binary splits.

**Split Creation**

A split is basically including an attribute in the dataset and a value. We can create a split in dataset with the help of following three parts:

* **Part1: Calculating Gini Score:** We have just discussed this part in the previous section.
* **Part2: Splitting a dataset:** It may be defined as separating a dataset into two lists of rows having index of an attribute and a split value of that attribute. After getting the two groups - right and left, from the dataset, we can calculate the value of split by using Gini score calculated in first part. Split value will decide in which group the attribute will reside.
* **Part3: Evaluating all splits:** Next part after finding Gini score and splitting dataset is the evaluation of all splits. For this purpose, first, we must check every value associated with each attribute as a candidate split. Then we need to find the best possible split by evaluating the cost of the split. The best split will be used as a node in the decision tree.

**Building a Tree**

****

As we know that a tree has root node and terminal nodes. After creating the root node, we can build the tree by following two parts:

**Part1: Terminal node creation**

While creating terminal nodes of decision tree, one important point is to decide when to stop growing tree or creating further terminal nodes. It can be done by using two criteria namely maximum tree depth and minimum node records as follows:

* **Maximum Tree Depth:** As name suggests, this is the maximum number of the nodes in a tree after root node. We must stop adding terminal nodes once a tree

reached at maximum depth i.e. once a tree got maximum number of terminal nodes.

* **Minimum Node Records:** It may be defined as the minimum number of training patterns that a given node is responsible for. We must stop adding terminal nodes once tree reached at these minimum node records or below this minimum.

Terminal node is used to make a final prediction.

**Part2: Recursive Splitting**

As we understood about when to create terminal nodes, now we can start building our tree. Recursive splitting is a method to build the tree. In this method, once a node is created, we can create the child nodes (nodes added to an existing node) recursively on each group of data, generated by splitting the dataset, by calling the same function again and again.

**Prediction**

After building a decision tree, we need to make a prediction about it. Basically, prediction involves navigating the decision tree with the specifically provided row of data.

We can make a prediction with the help of recursive function, as did above. The same prediction routine is called again with the left or the child right nodes.

**Assumptions**

The following are some of the assumptions we make while creating decision tree:

* While preparing decision trees, the training set is as root node.
* Decision tree classifier prefers the features values to be categorical. In case if you want to use continuous values then they must be done discretized prior to model building.
* Based on the attribute’s values, the records are recursively distributed.
* Statistical approach will be used to place attributes at any node position i.e.as root node or internal node.

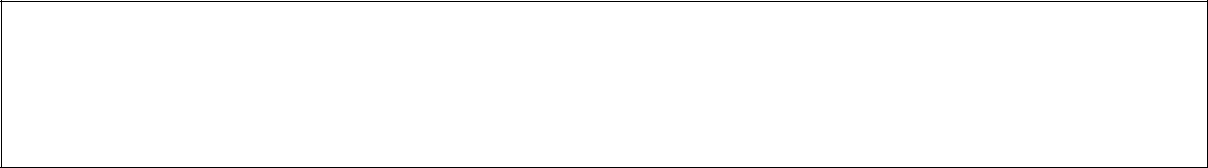
**Implementation in Python**

****

**Example**

In the following example, we are going to implement Decision Tree classifier on Pima Indian Diabetes:

First, start with importing necessary python packages:

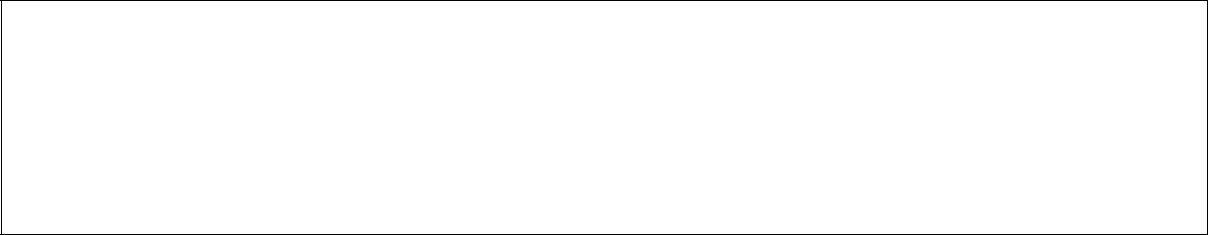


import pandas as pd

from sklearn.tree import DecisionTreeClassifier

from sklearn.model\_selection import train\_test\_split

Next, download the iris dataset from its weblink as follows:



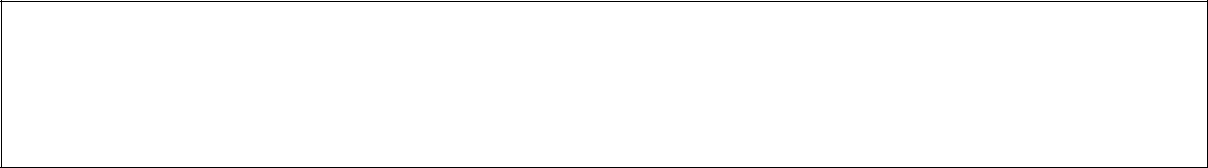
col\_names = ['pregnant', 'glucose', 'bp', 'skin', 'insulin', 'bmi', 'pedigree', 'age', 'label']

pima = pd.read\_csv(r"C:\pima-indians-diabetes.csv", header=None, names=col\_names)

pima.head()

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **pregnant** | **glucose** | **bp** | **skin** | **insulin** | **bmi** | **pedigree** | **age** | **label** |
| **0** | 6 | 148 | 72 | 35 | 0 | 33.6 | 0.627 | 50 | 1 |
| **1** | 1 | 85 | 66 | 29 | 0 | 26.6 | 0.351 | 31 | 0 |
| **2** | 8 | 183 | 64 | 0 | 0 | 23.3 | 0.672 | 32 | 1 |
| **3** | 1 | 89 | 66 | 23 | 94 | 28.1 | 0.167 | 21 | 0 |
| **4** | 0 | 137 | 40 | 35 | 168 | 43.1 | 2.288 | 33 | 1 |

Now, split the dataset into features and target variable as follows:



feature\_cols = ['pregnant', 'insulin', 'bmi', 'age','glucose','bp','pedigree']

1. = pima[feature\_cols] # Features y = pima.label # Target variable

Next, we will divide the data into train and test split. The following code will split the dataset into 70% training data and 30% of testing data:



X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=1)

Next, train the model with the help of **DecisionTreeClassifier** class of **sklearn** as follows:



clf = DecisionTreeClassifier()

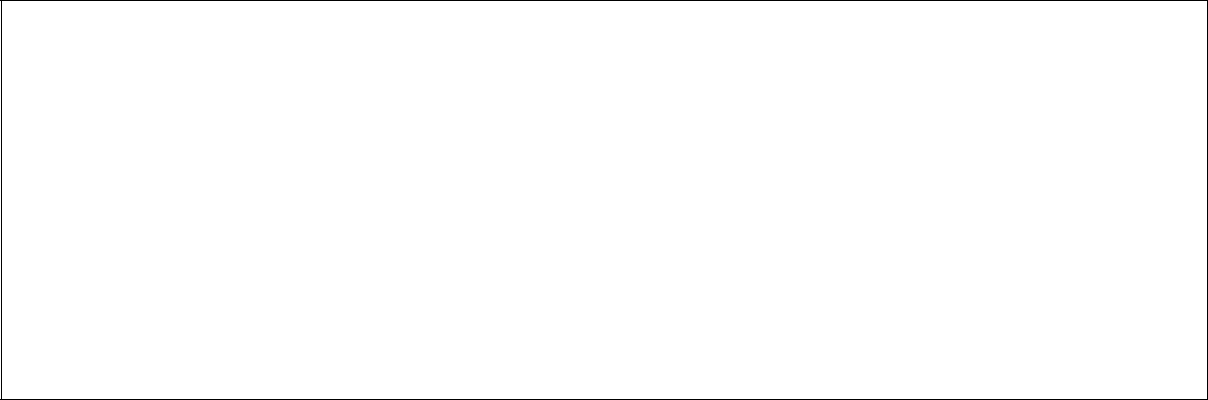
clf = clf.fit(X\_train,y\_train)

At last we need to make prediction. It can be done with the help of following script:



y\_pred = clf.predict(X\_test)

Next, we can get the accuracy score, confusion matrix and classification report as follows:



from sklearn.metrics import classification\_report, confusion\_matrix, accuracy\_score

result = confusion\_matrix(y\_test, y\_pred)

print("Confusion Matrix:")

print(result)

result1 = classification\_report(y\_test, y\_pred)

print("Classification Report:",)

print (result1)

result2 = accuracy\_score(y\_test,y\_pred)

print("Accuracy:",result2)

**Output**

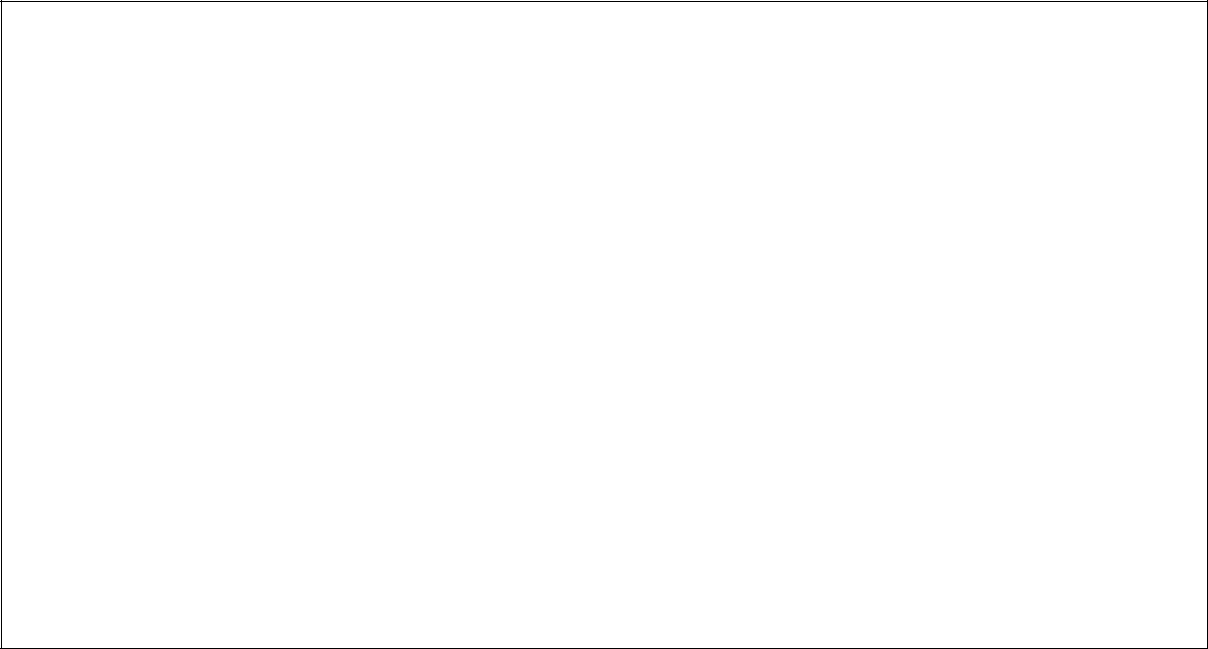
****

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Confusion Matrix: | | |  |  |  |
| [[116 | 30] |  |  |  |  |
| [ 46 | 39]] |  |  |  |  |
| Classification Report: | | |  |  |  |
|  |  | precision | recall | f1-score | support |
|  | 0 | 0.72 | 0.79 | 0.75 | 146 |
|  | 1 | 0.57 | 0.46 | 0.51 | 85 |
| micro avg | | 0.67 | 0.67 | 0.67 | 231 |
| macro avg | | 0.64 | 0.63 | 0.63 | 231 |
| weighted avg | | 0.66 | 0.67 | 0.66 | 231 |

Accuracy: 0.670995670995671

**Visualizing Decision Tree**

The above decision tree can be visualized with the help of following code:



from sklearn.tree import export\_graphviz from sklearn.externals.six import StringIO from IPython.display import Image import pydotplus

dot\_data = StringIO()

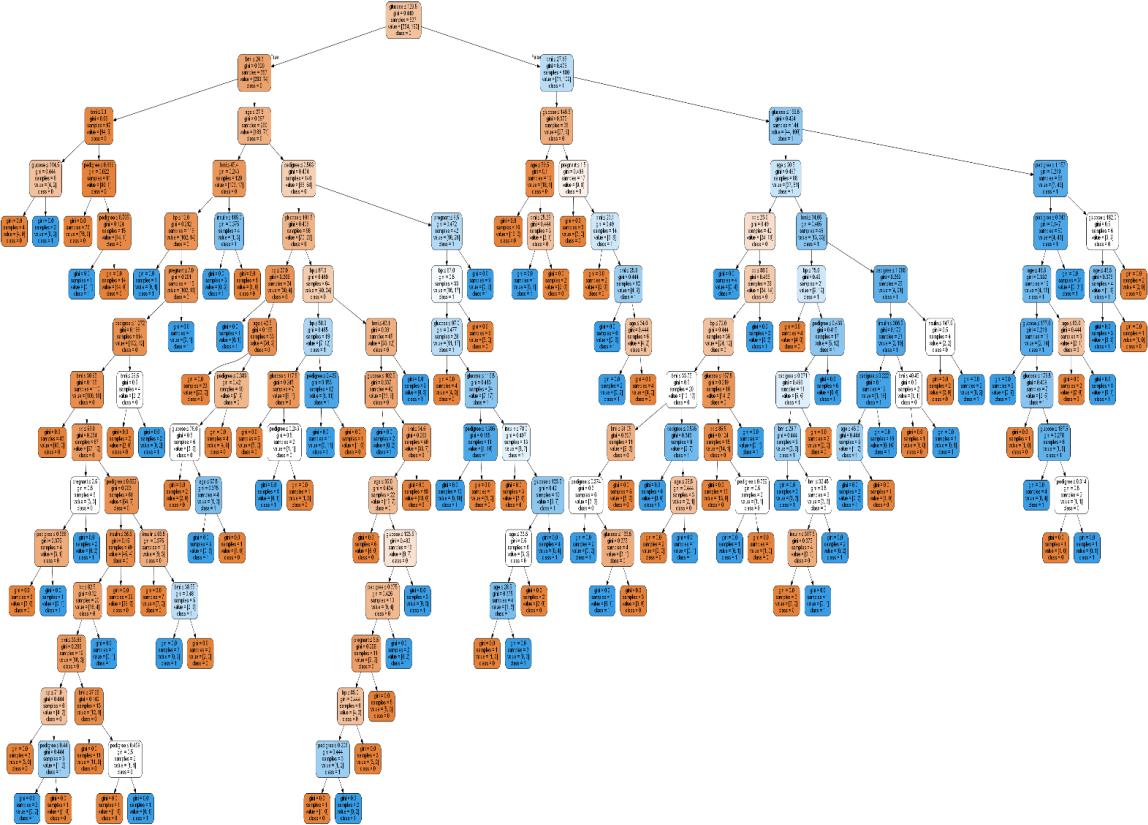
export\_graphviz(clf, out\_file=dot\_data, filled=True, rounded=True,

special\_characters=True,feature\_names = feature\_cols,class\_names=['0','1'])

graph = pydotplus.graph\_from\_dot\_data(dot\_data.getvalue())

graph.write\_png('Pima\_diabetes\_Tree.png')

Image(graph.create\_png())



**Introduction to Naïve Bayes Algorithm**

****

Naïve Bayes algorithms is a classification technique based on applying Bayes’ theorem with a strong assumption that all the predictors are independent to each other. In simple words, the assumption is that the presence of a feature in a class is independent to the presence of any other feature in the same class. For example, a phone may be considered as smart if it is having touch screen, internet facility, good camera etc. Though all these features are dependent on each other, they contribute independently to the probability of that the phone is a smart phone.

In Bayesian classification, the main interest is to find the posterior probabilities i.e. the probability of a label given some observed features, ( | ). With the help of Bayes theorem, we can express this in quantitative form as follows:

( ) ( | )

( | ) =

( )

Here, ( | ) is the posterior probability of class.

( ) is the prior probability of class.

( | ) is the likelihood which is the probability of predictor given class.

( ) is the prior probability of predictor.

**Building model using Naïve Bayes in Python**

****

Python library, Scikit learn is the most useful library that helps us to build a Naïve Bayes model in Python. We have the following three types of Naïve Bayes model under Scikit learn Python library:

**Gaussian Naïve Bayes**

It is the simplest Naïve Bayes classifier having the assumption that the data from each label is drawn from a simple Gaussian distribution.

**Multinomial Naïve Bayes**

Another useful Naïve Bayes classifier is Multinomial Naïve Bayes in which the features are assumed to be drawn from a simple Multinomial distribution. Such kind of Naïve Bayes are most appropriate for the features that represents discrete counts.

**Bernoulli Naïve Bayes**

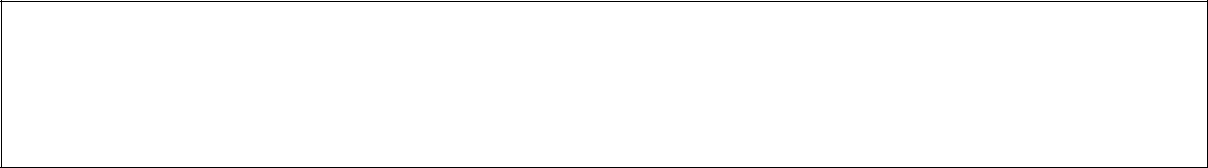
Another important model is Bernoulli Naïve Bayes in which features are assumed to be binary (0s and 1s). Text classification with ‘bag of words’ model can be an application of Bernoulli Naïve Bayes.

**Example**

Depending on our data set, we can choose any of the Naïve Bayes model explained above.

Here, we are implementing Gaussian Naïve Bayes model in Python:

We will start with required imports as follows:

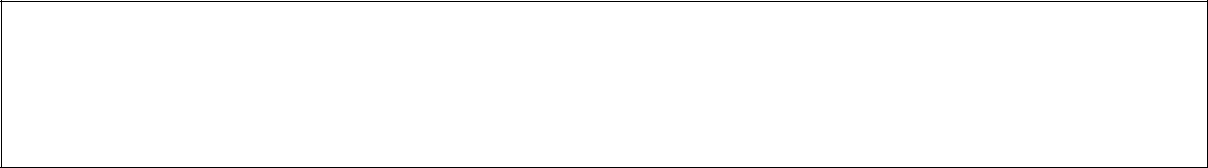


import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns; sns.set()

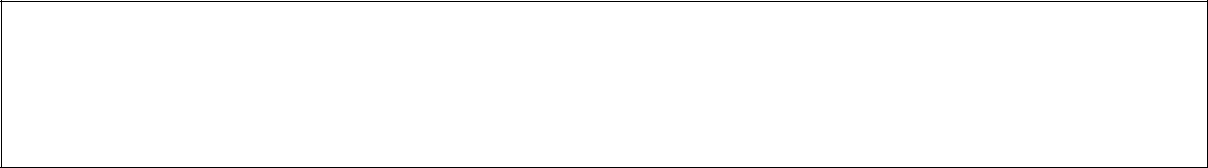
Now, by using **make\_blobs()** function of **Scikit learn,** we can generate blobs of points with Gaussian distribution as follows:



from sklearn.datasets import make\_blobs

X, y = make\_blobs(300, 2, centers=2, random\_state=2, cluster\_std=1.5) plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='summer');

Next, for using **GaussianNB** model, we need to import and make its object as follows:

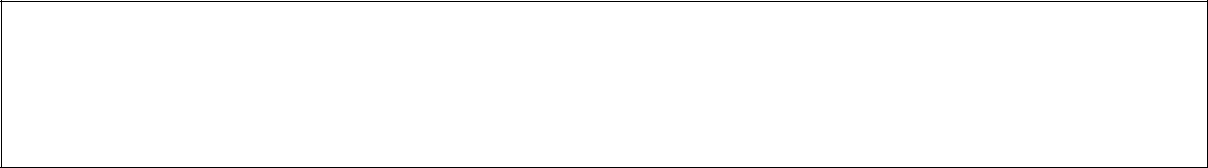


from sklearn.naive\_bayes import GaussianNB

model\_GBN = GaussianNB()

model\_GNB.fit(X, y);

Now, we have to do prediction. It can be done after generating some new data as follows:

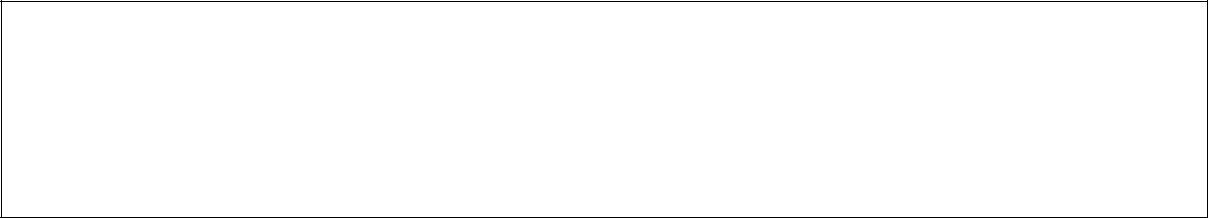


rng = np.random.RandomState(0)

Xnew = [-6, -14] + [14, 18] \* rng.rand(2000, 2)

ynew = model\_GNB.predict(Xnew)

Next, we are plotting new data to find its boundaries:



plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='summer')

lim = plt.axis()

plt.scatter(Xnew[:, 0], Xnew[:, 1], c=ynew, s=20, cmap='summer', alpha=0.1)

plt.axis(lim);

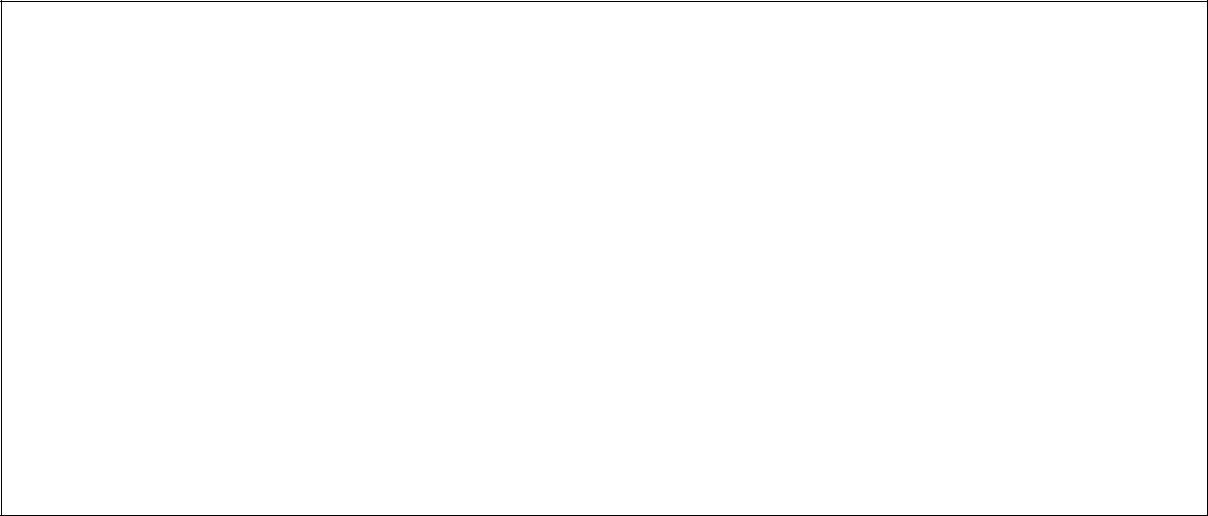
Now, with the help of following line of codes, we can find the posterior probabilities of first and second label:



yprob = model\_GNB.predict\_proba(Xnew)

yprob[-10:].round(3)

**Output**

****

array([[0.998, 0.002],

[1. ,0. ],

[0.987, 0.013],

[1. ,0. ],

[1. ,0. ],

[1. ,0. ],

[1. ,0. ],

[1. ,0. ],

[0. ,1. ],

[0.986, 0.014]])

**Pros & Cons**

****

**Pros**

The followings are some pros of using Naïve Bayes classifiers:

* Naïve Bayes classification is easy to implement and fast.
* It will converge faster than discriminative models like logistic regression.
* It requires less training data.
* It is highly scalable in nature, or they scale linearly with the number of predictors and data points.
* It can make probabilistic predictions and can handle continuous as well as discrete data.
* Naïve Bayes classification algorithm can be used for binary as well as multi-class classification problems both.

**Cons**

The followings are some cons of using Naïve Bayes classifiers:

* One of the most important cons of Naïve Bayes classification is its strong feature independence because in real life it is almost impossible to have a set of features which are completely independent of each other.
* Another issue with Naïve Bayes classification is its ‘zero frequency’ which means that if a categorial variable has a category but not being observed in training data set, then Naïve Bayes model will assign a zero probability to it and it will be unable to make a prediction.

**Applications of Naïve Bayes classification**

****

The following are some common applications of Naïve Bayes classification:

**Real-time prediction:** Due to its ease of implementation and fast computation, it can be used to do prediction in real-time.

**Multi-class prediction:** Naïve Bayes classification algorithm can be used to predict posterior probability of multiple classes of target variable.

**Text classification:** Due to the feature of multi-class prediction, Naïve Bayes classification algorithms are well suited for text classification. That is why it is also used to solve problems like spam-filtering and sentiment analysis.

**Recommendation system:** Along with the algorithms like collaborative filtering, Naïve Bayes makes a Recommendation system which can be used to filter unseen information and to predict weather a user would like the given resource or not.

**Introduction**

****

Random forest is a supervised learning algorithm which is used for both classification as well as regression. But however, it is mainly used for classification problems. As we know that a forest is made up of trees and more trees means more robust forest. Similarly, random forest algorithm creates decision trees on data samples and then gets the prediction from each of them and finally selects the best solution by means of voting. It is an ensemble method which is better than a single decision tree because it reduces the over-fitting by averaging the result.

**Working of Random Forest Algorithm**

****

We can understand the working of Random Forest algorithm with the help of following steps:

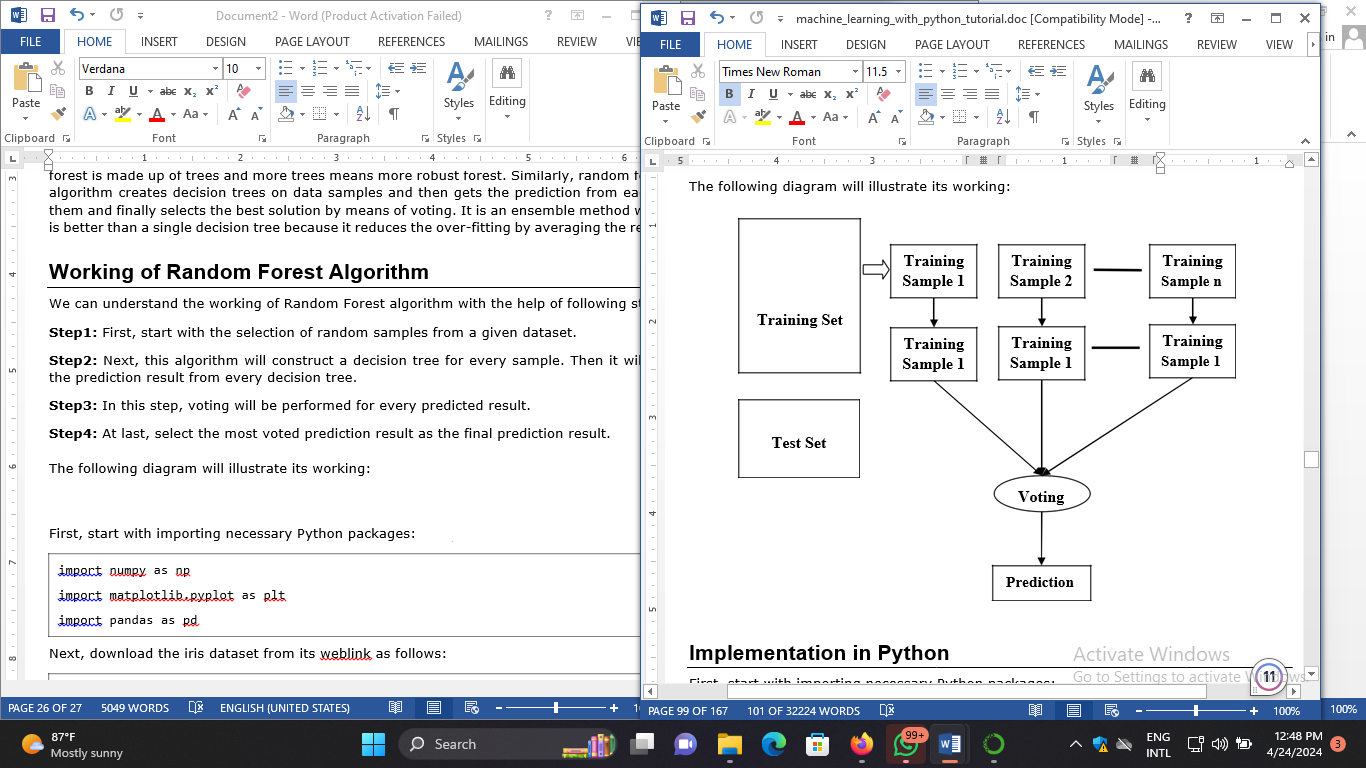
**Step1:** First, start with the selection of random samples from a given dataset.

**Step2:** Next, this algorithm will construct a decision tree for every sample. Then it will get the prediction result from every decision tree.

**Step3:** In this step, voting will be performed for every predicted result.

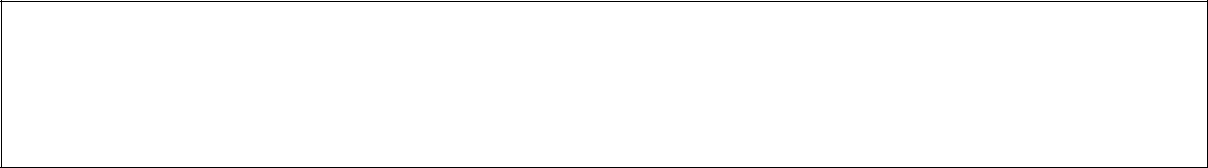
**Step4:** At last, select the most voted prediction result as the final prediction result.

The following diagram will illustrate its working:



**Implantation in Python**

First, start with importing necessary Python packages:



import numpy as np

import matplotlib.pyplot as plt

import pandas as pd

Next, download the iris dataset from its weblink as follows:



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

Next, we need to assign column names to the dataset as follows:



headernames = ['sepal-length', 'sepal-width', 'petal-length', 'petal-width', 'Class']

Now, we need to read dataset to **pandas dataframe** as follows:



dataset = pd.read\_csv(path, names=headernames)

dataset.head()

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |
|  | **sepal-** | **sepal-** | **petal-** | **petal-width** | **Class** |  |
|  | **length** | **width** | **length** |  |  |  |
|  |  |  |  |  |  |  |
| **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 |  |
|  |  |  |  |  |  |  |

Data Preprocessing will be done with the help of following script lines:



1. = dataset.iloc[:, :-1].values y = dataset.iloc[:, 4].values

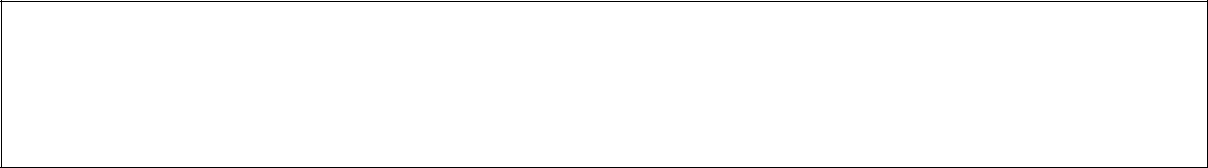
Next, we will divide the data into train and test split. The following code will split the dataset into 70% training data and 30% of testing data:



from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.30)

Next, train the model with the help of **RandomForestClassifier** class of **sklearn** as follows:



from sklearn.ensemble import RandomForestClassifier

classifier = RandomForestClassifier(n\_estimators=50)

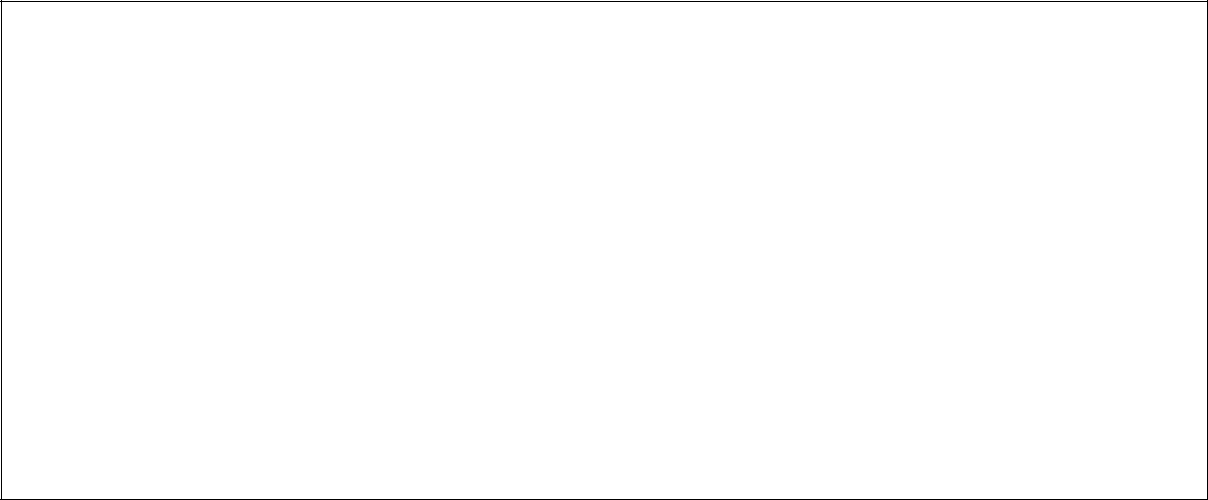
classifier.fit(X\_train, y\_train)

At last, we need to make prediction. It can be done with the help of following script:



y\_pred = classifier.predict(X\_test)

Next, print the results as follows:



from sklearn.metrics import classification\_report, confusion\_matrix, accuracy\_score

result = confusion\_matrix(y\_test, y\_pred)

print("Confusion Matrix:")

print(result)

result1 = classification\_report(y\_test, y\_pred)

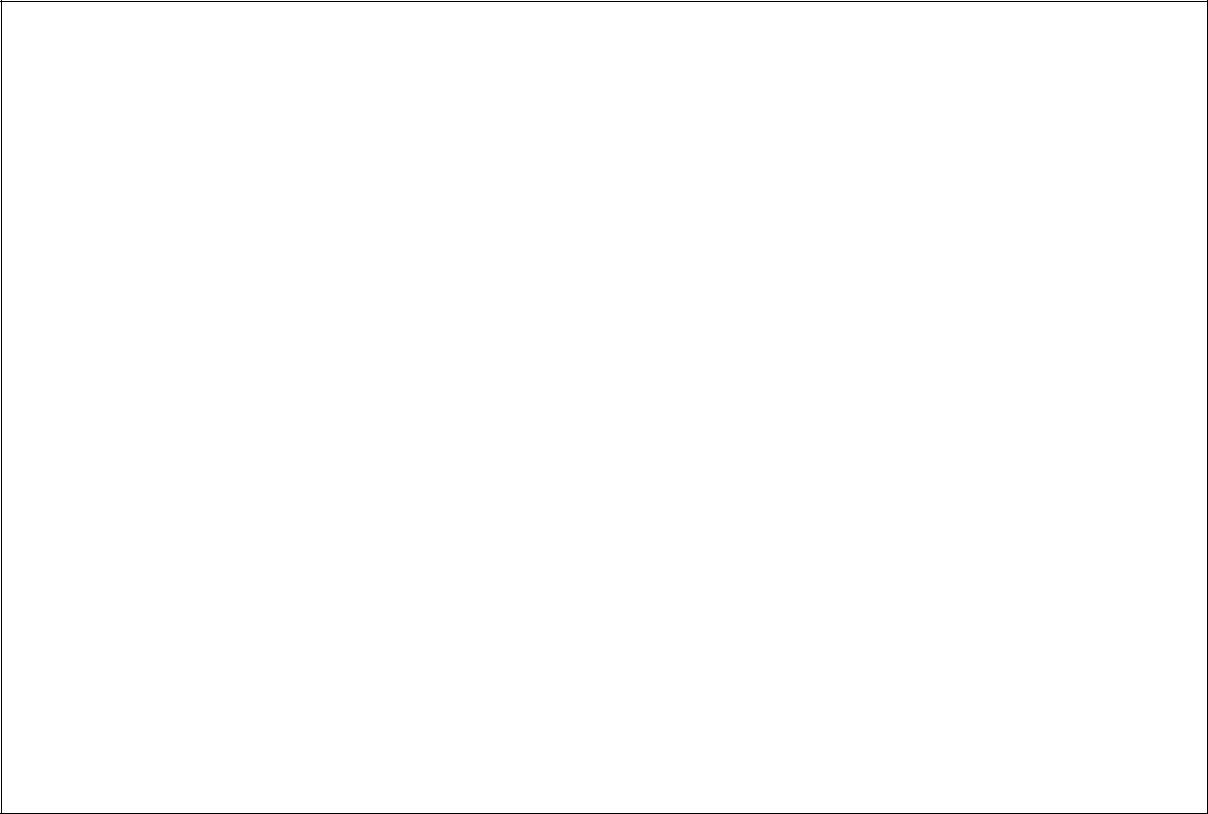
print("Classification Report:",)

print (result1)

result2 = accuracy\_score(y\_test,y\_pred)

print("Accuracy:",result2)

**Output**

****

|  |  |  |  |
| --- | --- | --- | --- |
| Confusion Matrix: | | | |
| [[14 | | 0 | 0] |
| [ | 0 | 18 | 1] |
| [ | 0 | 0 | 12]] |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Classification Report: | |  |  |  |
|  | precision | recall | f1-score | support |
| Iris-setosa | 1.00 | 1.00 | 1.00 | 14 |
| Iris-versicolor | 1.00 | 0.95 | 0.97 | 19 |
| Iris-virginica | 0.92 | 1.00 | 0.96 | 12 |
| micro avg | 0.98 | 0.98 | 0.98 | 45 |
| macro avg | 0.97 | 0.98 | 0.98 | 45 |
| weighted avg | 0.98 | 0.98 | 0.98 | 45 |

Accuracy: 0.9777777777777777

**Pros and Cons of Random Forest**

****

**Pros**

The following are the advantages of Random Forest algorithm:

* It overcomes the problem of overfitting by averaging or combining the results of different decision trees.
* Random forests work well for a large range of data items than a single decision tree does.
* Random forest has less variance then single decision tree.
* Random forests are very flexible and possess very high accuracy.
* Scaling of data does not require in random forest algorithm. It maintains good accuracy even after providing data without scaling.
* Random Forest algorithms maintains good accuracy even a large proportion of the data is missing.

**Cons**

The following are the disadvantages of Random Forest algorithm:

* Complexity is the main disadvantage of Random forest algorithms.
* Construction of Random forests are much harder and time-consuming than decision trees.
* More computational resources are required to implement Random Forest algorithm.
* It is less intuitive in case when we have a large collection of decision trees.
* The prediction process using random forests is very time-consuming in comparison with other algorithms.