ML - 2 (Classification)

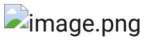
Logistic Regression

Logistic Regression is a statistical method used in machine learning and statistics to model and predict the probability of a certain outcome. Despite its name, logistic regression is actually a classification algorithm, not a regression algorithm. It is used to solve binary classification problems (where the target variable has two possible outcomes, e.g., "yes" or "no," "0" or "1") and can be extended to multiclass classification.

Applications of Logistic Regression

- Binary Classification:
  - Email Spam Detection (Spam or Not Spam).
  - Disease Diagnosis (e.g., Cancer Detection: Malignant or Benign).
  - Customer Churn Prediction (Stay or Leave).
- Multiclass Classification (via Extensions):
  - Using techniques like One-vs-Rest (OvR) or Softmax Regression.

Sigmoid Function



```
!pip install pyforest
```

```
Collecting pyforest
  Downloading pyforest-1.1.2.tar.gz (17 kB)
  Preparing metadata (setup.py) ... done
Building wheels for collected packages: pyforest
  Building wheel for pyforest (setup.py) ... done
  Created wheel for pyforest: filename=pyforest-1.1.2-py2.py3-none-any.whl size=15897 sha256=f3becd8b1d878fa6eae35062b3c4793ebbbac863cae321025bc91073734cb5ac
  Stored in directory: /root/.cache/pip/wheels/c5/ca/73/5cdc3d087111bfbdef90be5457aa03c00c0e32241b2752445c
Successfully built pyforest
Installing collected packages: pyforest
Successfully installed pyforest-1.1.2
```

When discussing **types of learning** in the context of **Logistic Regression**, it's important to clarify that **Logistic Regression** is a **supervised learning** algorithm, typically used for classification tasks. However, Logistic Regression does not typically fall into **distance-based learning** methods like some other algorithms (e.g., k-Nearest Neighbors or Support Vector Machines). That said, it can be understood in terms of different **types of learning** based on its approach to optimization, feature selection, and the way it handles data.

Here, I will break down **Logistic Regression** in terms of **learning paradigms** and how they relate to concepts like **distance-based learning**, **probabilistic learning**, and **optimization-based learning**.

1. Optimization-Based Learning (Gradient Descent / Iterative Learning)

In **Logistic Regression**, the model parameters (coefficients) are learned through an **optimization process** rather than by directly measuring distances between data points, as in **distance-based learning** methods like k-Nearest Neighbors (KNN).

- **How it works**: Logistic Regression uses a **cost function** (usually **log-likelihood** or **cross-entropy loss**) to measure the difference between the predicted probabilities and the true labels. The goal is to **minimize** this cost function by adjusting the model's weights.
- The most common approach to optimization is **Gradient Descent**, an iterative process that adjusts the coefficients of the model in the direction of the steepest descent of the cost function.

  Example:

  - If you're using **Batch Gradient Descent**, all training examples are used to compute the gradient at each step, and coefficients are updated accordingly.
  - **Stochastic Gradient Descent (SGD)** updates the coefficients after looking at each individual data point, making it faster but potentially noisier.
- **Learning Type**: **Optimization-based learning** (minimizing the cost function using gradient-based methods).

2. Probabilistic Learning (Log-Odds and Sigmoid Function)

Logistic Regression is often described as a **probabilistic model** because it predicts the **probability** that an instance belongs to a certain class, rather than directly predicting the class itself.

- **How it works**: Logistic Regression models the **log-odds** (the logarithm of the odds ratio) of the dependent variable using a **linear combination** of the input features, which is then passed through the **sigmoid function** to squash the output to a probability between 0 and 1.

  The general formula for logistic regression is:

```
\[
P(y = 1 | X) = \frac{1}{1 + e^{-z}}
\]
### P(y=1 | X) = ÷

Where \( z = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \dots + \beta_n x_n \) is the linear combination of the input features.
```

- **Learning Type**: **Probabilistic learning** (estimating probabilities of class membership using the logistic function).

3. **Linear Decision Boundary / Model-Based Learning**

Logistic Regression is a **linear classifier**, meaning it learns a **linear decision boundary** that separates the different classes. This decision boundary is determined by the coefficients (weights) learned during the training process.

- **How it works**: Logistic Regression fits a **linear model** to the data, with the aim of finding the hyperplane (in higher dimensions) that best separates the classes in the feature space. This hyperplane corresponds to a **threshold** on the logistic function, which gives the predicted probability of the instance belonging to the positive class.
  - If the probability (P(y=1 | X)) is above a certain threshold (usually 0.5), the classifier predicts class 1; otherwise, it predicts class 0.
- **Learning Type**: **Model-based learning** (using a linear model to learn a decision boundary).

4. **Distance-Based Learning (Not Directly Applicable to Logistic Regression)**

Although **Logistic Regression** is not inherently a **distance-based learning** method like **k-Nearest Neighbors (KNN)** or **Support Vector Machines (SVM)**, there are **indirect** ways in which distance-based concepts can play a role in logistic regression, particularly in the **feature space** or when using regularization techniques.

- **How it works**: While Logistic Regression itself doesn't measure the **distance** between points directly, the **decision boundary** learned by Logistic Regression can be influenced by the relative positions of data points in the feature space. The decision boundary separates instances that are closer to each other in terms of feature values.
  - For example, if two classes are **separable by a straight line** (in 2D) or a hyperplane (in higher dimensions), the decision boundary learned by logistic regression could be thought of as dividing regions in the feature space.
  - **Regularization** (e.g., L2 regularization) helps prevent overfitting by constraining the model's coefficients, which indirectly affects how "far apart" or "close together" data points are in terms of model predictions.
- **Learning Type**: **Not directly distance-based** but can be related to decision boundaries in the feature space.

5. **Regularized Logistic Regression (L1 and L2 Regularization)**

Regularization techniques such as **L1** (Lasso) and **L2** (Ridge) regularization add a penalty term to the logistic regression cost function to **control model complexity** and **prevent overfitting**.

- **L1 Regularization (Lasso)**: Encourages sparsity in the model by driving some coefficients to zero, effectively performing **feature selection**.
  - **Learning Type**: **Regularization-based learning** (penalizing large coefficients).
- **L2 Regularization (Ridge)**: Prevents the coefficients from becoming too large by adding a penalty proportional to the square of the coefficients.
  - **Learning Type**: **Regularization-based learning** (penalizing large coefficients to avoid overfitting).

6. **Feature-Based Learning**

Logistic Regression learns based on the **features** provided in the dataset. In this sense, it is **feature-based learning** because it assumes that the relationship between the features and the target variable is linear.

- **How it works**: Logistic regression computes a **weighted sum of features** and then applies the **sigmoid function** to estimate the probability of the outcome. The model learns which features are most important for predicting the target class by adjusting the weights during training.
- **Learning Type**: **Feature-based learning** (learning from the features and their relationships to the target).

7. **Gradient-Based Learning**

Logistic Regression can be trained using gradient-based methods such as **Gradient Descent**, which is the most common optimization technique for training the model.

- **How it works**: The gradient of the cost function is calculated with respect to the model's parameters, and the parameters are updated iteratively to minimize the cost function. This is typically done using either **Batch Gradient Descent**, **Stochastic Gradient Descent (SGD)**, or **Mini-Batch Gradient Descent**.
- **Learning Type**: **Gradient-based learning** (updating model parameters using gradients of the cost function).

Summary of Learning Types in Logistic Regression:

Learning Type	Description	Key Concepts
Optimization-Based Learning	Model parameters are learned through optimization (e.g., gradient descent).	Gradient Descent, Cost Function, Weights
Probabilistic Learning	The model outputs probabilities of class membership, using the logistic function.	Sigmoid, Log-Odds, Probabilities
Linear Decision Boundary	Logistic Regression learns a linear boundary to separate classes.	Linear Model, Decision Boundary, Hyperplane
Distance-Based Learning	Logistic Regression isn't distance-based but the decision boundary can be related to distances in feature space.	Decision Boundary, Feature Space
Regularized Logistic Regression	Uses regularization (L1 or L2) to prevent overfitting by penalizing large coefficients.	L1 Regularization (Lasso), L2 Regularization (Ridge)
Feature-Based Learning	Learning from features and their relationship to the target.	Feature Weights, Feature Selection
Gradient-Based Learning	Uses gradient-based optimization methods (e.g., Gradient Descent) to update model parameters.	Gradient Descent, Stochastic Gradient Descent

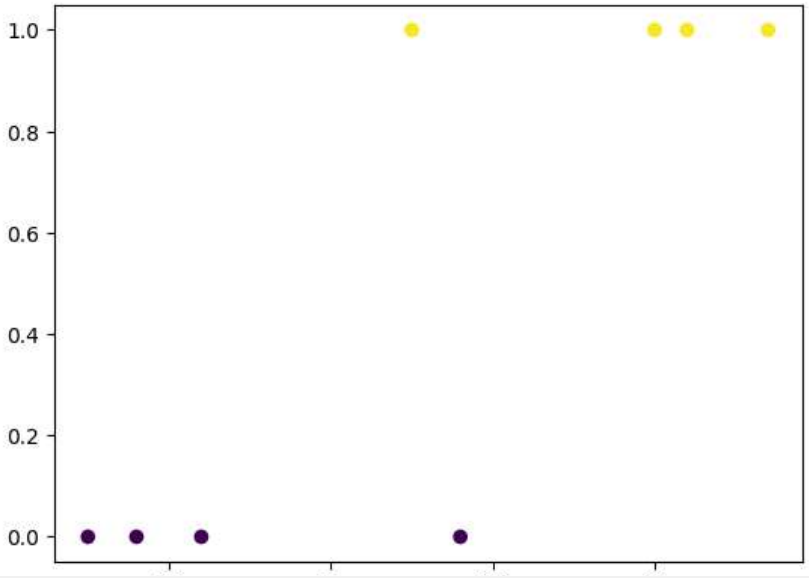
```
import pyforest
```

```
data = pd.DataFrame({'W': (62,58,78,75,92,55,90,97), 'HD': (0,0,0,1,1,0,1,1)})

data
```

	W	HD
0	62	0
1	58	0
2	78	0
3	75	1
4	92	1
5	55	0
6	90	1

```
plt.scatter(data.W, data.HD, c=data.HD)
```

```
<matplotlib.collections.PathCollection at 0x7d5e276117b0>
```



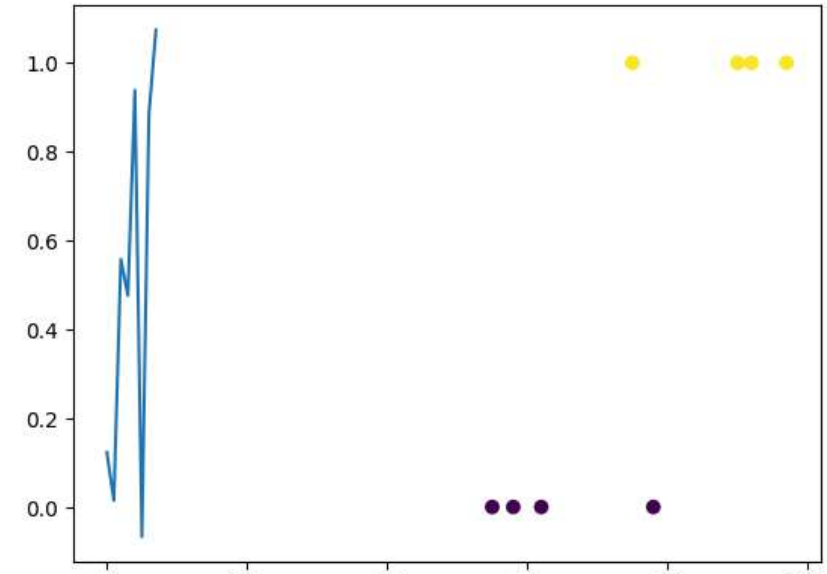

```
x = data.W
y = data.HD

b1 = np.sum((x - x.mean())* (y-y.mean())) / np.sum((x-x.mean())**2)
b0 = y.mean() - b1 * x.mean()
b1, b0
```

```
y_reg = b1 * x + b0
print(y_reg)
```

```
0    0.123058
1    0.014390
2    0.557730
3    0.476229
4    0.938067
5   -0.067110
6    0.883733
7    1.073902
Name: W, dtype: float64
```

```
plt.scatter(data.W, data.HD, c=data.HD)
plt.plot(y_reg)
```

```
[<matplotlib.lines.Line2D at 0x7d5e253c6ce0>]
```



Admissions Data Set

```
admissions = pd.read_csv("https://raw.githubusercontent.com/HarpyTech/DSAI-MTech/refs/heads/main/ML2/Admission_Predict.csv")

admissions.head()
```

	Serial No.	GRE Score	TOEFL Score	University Rating	SOP	LOR	CGPA	Research	Chance of Admit
0	1	337	118	4	4.5	4.5	9.65	1	1
1	2	324	107	4	4.0	4.5	8.87	1	1
2	3	316	104	3	3.0	3.5	8.00	1	0
3	4	322	110	3	3.5	2.5	8.67	1	1

```
admissions = admissions.drop(columns="Serial No.")
admissions.head()
```

	GRE Score	TOEFL Score	University Rating	SOP	LOR	CGPA	Research	Chance of Admit
0	337	118	4	4.5	4.5	9.65	1	1
1	324	107	4	4.0	4.5	8.87	1	1
2	316	104	3	3.0	3.5	8.00	1	0
3	322	110	3	3.5	2.5	8.67	1	1

```
admissions.isnull().sum()
```

	0
GRE Score	0
TOEFL Score	0
University Rating	0
SOP	0
LOR	0
CGPA	0
Research	0
Chance of Admit	0

```
out = admissions["Chance of Admit"]
inp = admissions.drop(columns='Chance of Admit')
```

```
from sklearn.preprocessing import StandardScaler

sc = StandardScaler()
inp_sc = sc.fit_transform(inp.iloc[:, :-1])
inp_sc = pd.DataFrame(inp_sc, columns=inp.columns[:-1])

inp_sc.head()
```

	GRE Score	TOEFL Score	University Rating	SOP	LOR	CGPA
0	1.762107	1.746971	0.798829	1.093864	1.167321	1.764818
1	0.627656	-0.067635	0.798829	0.596653	1.167321	0.455151
2	-0.070467	-0.562528	-0.076600	-0.397769	0.052933	-1.005631
3	0.453126	0.427257	-0.076600	0.099442	-1.061454	0.119339

```
inp_sc['Research'] = inp.Research
inp_sc.head()
```

	GRE Score	TOEFL Score	University Rating	SOP	LOR	CGPA	Research
0	1.762107	1.746971	0.798829	1.093864	1.167321	1.764818	1
1	0.627656	-0.067635	0.798829	0.596653	1.167321	0.455151	1
2	-0.070467	-0.562528	-0.076600	-0.397769	0.052933	-1.005631	1
3	0.453126	0.427257	-0.076600	0.099442	-1.061454	0.119339	1

```
inp_sc.describe()
```

	GRE Score	TOEFL Score	University Rating	SOP	LOR	CGPA	Research
count	4.000000e+02	4.000000e+02	4.000000e+02	4.000000e+02	4.000000e+02	4.000000e+02	400.000000
mean	-4.174439e-16	5.595524e-16	7.105427e-17	7.993606e-17	-1.332268e-16	7.771561e-16	0.547500
std	1.001252e+00	1.001252e+00	1.001252e+00	1.001252e+00	1.001252e+00	1.001252e+00	0.498362
min	-2.339367e+00	-2.542098e+00	-1.827457e+00	-2.386613e+00	-2.733036e+00	-3.020504e+00	0.000000
25%	-7.685900e-01	-7.274920e-01	-9.520286e-01	-8.949798e-01	-5.042604e-01	-7.201909e-01	0.000000
50%	1.679859e-02	-6.763531e-02	-7.660001e-02	9.944220e-02	5.293342e-02	1.859559e-02	1.000000
75%	7.149218e-01	7.571856e-01	7.988286e-01	5.966532e-01	6.101273e-01	7.783704e-01	1.000000

```
import statsmodels.api as sm
from statsmodels.stats.outliers_influence import variance_inflation_factor

vif = pd.DataFrame()

vif["VIF"] = [variance_inflation_factor(inp_sc.values, i ) for i in range(inp_sc.shape[1]) ]
vif["feature"] = inp_sc.columns

vif.sort_values("VIF", ascending=False )
```

	VIF	feature
5	5.205309	CGPA
0	4.358514	GRE Score
1	4.282118	TOEFL Score
3	3.063188	SOP
2	2.918556	University Rating
4	2.430409	LOR

```
inp_sc.shape
```

```
(400, 7)
```

```
from sklearn.model_selection import train_test_split

xtrain, xtest, ytrain, ytest = train_test_split(inp_sc, out, test_size=0.2, random_state=10)
```

```
xtrainc = sm.add_constant(xtrain) # To Add B0 constant
mod_stat = sm.Logit(ytrain, xtrainc)

logit_mod = mod_stat.fit()
logit_mod.summary()
```

```
Optimization terminated successfully.
         Current function value: 0.241326
         Iterations 8
```

Logit Regression Results

Dep. Variable:	Chance of Admit	**No. Observations:**	320
Model:	Logit	**Df Residuals:**	312
Method:	MLE	**Df Model:**	7
Date:	Sat, 11 Jan 2025	**Pseudo R-squ.:**	0.6486
Time:	08:48:53	**Log-Likelihood:**	-77.224
converged:	True	**LL-Null:**	-219.78
Covariance Type:	nonrobust	**LLR p-value:**	9.137e-58

	coef	std err	z	P>|z|	[0.025	0.975]
const	-0.7119	0.330	-2.157	0.031	-1.359	-0.065
GRE Score	0.6095	0.447	1.365	0.172	-0.266	1.485
TOEFL Score	0.1989	0.403	0.493	0.622	-0.592	0.990
University Rating	0.5883	0.383	1.535	0.125	-0.163	1.339
SOP	0.1768	0.374	0.473	0.636	-0.555	0.909
LOR	0.5118	0.308	1.662	0.096	-0.092	1.115
CGPA	2.6273	0.544	4.832	0.000	1.562	3.693

```
xtrain1 = xtrain.copy()

while(len(xtrain1.columns) > 0):
    xtrain1c = sm.add_constant(xtrain1)
    logit_mod = sm.Logit(ytrain,xtrain1c).fit()

    f = logit_mod.pvalues[1:].idxmax()
    if logit_mod.pvalues[1:].max() > 0.05:
        xtrain1 = xtrain1.drop(columns=f)
    else:
        break

print(xtrain1.columns)
```

```
Optimization terminated successfully.
         Current function value: 0.241326
         Iterations 8
 Optimization terminated successfully.
         Current function value: 0.241676
         Iterations 8
 Optimization terminated successfully.
         Current function value: 0.242082
         Iterations 8
 Optimization terminated successfully.
         Current function value: 0.244507
         Iterations 8
 Index(['GRE Score', 'University Rating', 'LOR', 'CGPA'], dtype='object')
```

```
from sklearn.linear_model import LogisticRegression
model = LogisticRegression()
model = model.fit(xtrain, ytrain)

ypred = model.predict(xtest)

ypred
```

```
array([0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1,
       1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0,
       0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1,
       1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0])
```

```
np.round(model.predict_proba(xtest), 3)[:5]
```

```
array([[0.934, 0.066],
       [0.205, 0.795],
       [0.978, 0.022],
       [0.999, 0.001]
```

```
model.coef_
```

```
array([[0.66801172, 0.29734361, 0.54609258, 0.23830659, 0.49665543,
        2.11910362, 0.49882117]])
```

```
from sklearn.metrics import confusion_matrix, accuracy_score, classification_report

cm = confusion_matrix(ytest, ypred)
print(cm)
```

```
[[34  8]
 [ 5 33]]
```

```
ytest.value_counts()
```

Chance of Admit	count
0	42
1	38

```
tp = cm[1,1]
tn = cm[0,0]
fp = cm[0,1]
fn = cm[1,0]
```

```
print(tp, tn, fp, fn)
```

```
33 34 8 5
```

```
accuracy = (tp + tn) / (tp + tn + fp + fn)
recall = tp / (tp + fn)
precision = tp / (tp + fp)
f1 = 2 * (precision * recall) / (precision + recall)

print(accuracy, recall, precision, f1)
```

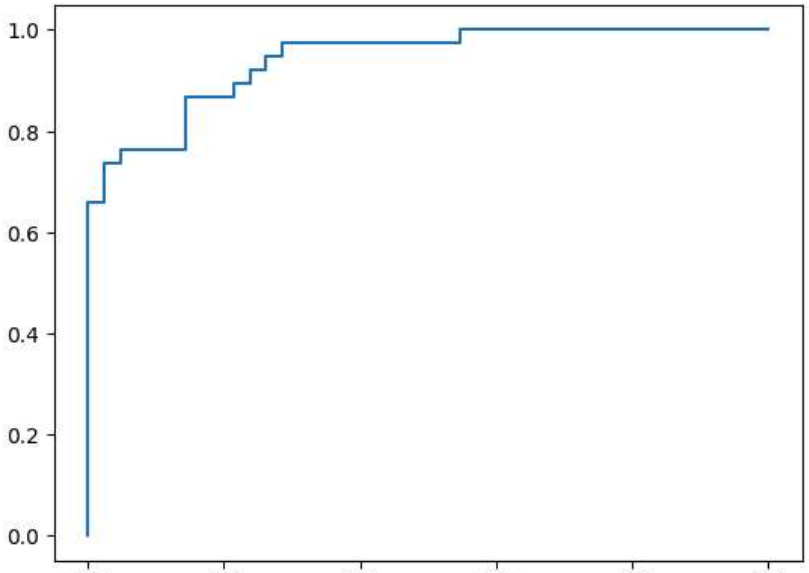
```
0.8375 0.868421052631579 0.8048780487804879 0.8354430379746836
```

```
from sklearn.metrics import recall_score, roc_curve, precision_score, f1_score, roc_auc_score

print("Accuracy", accuracy_score(ytest, ypred))
print("Recall", recall_score(ytest, ypred))
print("Precision", precision_score(ytest, ypred))
print("F1 Score", f1_score(ytest, ypred))
print(classification_report(ytest, ypred))
```

```
Accuracy 0.8375
Recall 0.868421052631579
Precision 0.8048780487804879
F1 Score 0.8354430379746836
              precision    recall  f1-score   support

           0       0.87      0.81      0.84        42
           1       0.80      0.87      0.84        38

    accuracy                           0.84        80
   macro avg       0.84      0.84      0.84        80
weighted avg       0.84      0.84      0.84        80
```

```
yprob = model.predict_proba(xtest)
yprob1= yprob[:,1]
fpr, tpr, th = roc_curve(ytest, yprob1)

plt.plot(fpr, tpr)
```

```
[<matplotlib.lines.Line2D at 0x7d5e17d91a50>]
```




```
roc_auc_score(ytest, yprob1)
```

```
0.9411027568922306
```

```
from sklearn.metrics import cohen_kappa_score

cohen_kappa_score(ytest, ypred)
```

```
0.6754057428214731
```

```
logit_mod.aic
```

```
166.48425548448398
```

```
logit_mod.bic
```

```
185.32586046345284
```

21 Dec 2024

```
ypred_th = np.zeros([len(ytest), 1])
ypred_th[ypred > 0.7] = 1
```

```
print(classification_report(ytest, ypred_th))
```

```
              precision    recall  f1-score   support

           0       0.87      0.81      0.84        42
           1       0.80      0.87      0.84        38

    accuracy                           0.84        80
   macro avg       0.84      0.84      0.84        80
weighted avg       0.84      0.84      0.84        80
```

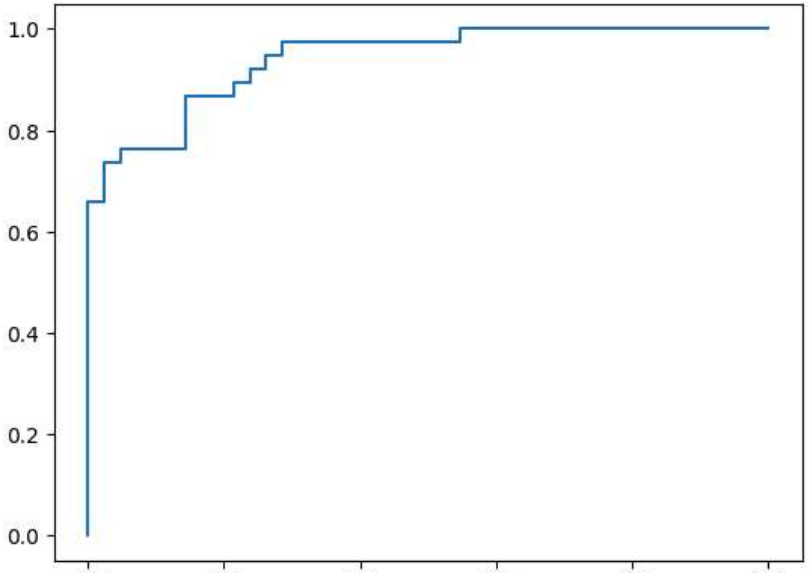
```
th1 = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
acc = []
rec = []
prec = []
f1 = []

for i in th1:
  ypred_th = np.zeros([len(ytest), 1])
  ypred_th[yprob1 > i] = 1

  acc.append(accuracy_score(ytest, ypred_th))
  rec.append(recall_score(ytest, ypred_th))
  prec.append(precision_score(ytest, ypred_th))
  f1.append(f1_score(ytest, ypred_th))


result_df = pd.DataFrame({'Threshold': th1, 'Accuracy': acc, 'Recall': rec, 'Precision': prec, 'F1 Score': f1})
result_df
```

	Threshold	Accuracy	Recall	Precision	F1 Score
0	0.1	0.7375	0.973684	0.649123	0.778947
1	0.2	0.8125	0.973684	0.725490	0.831461
2	0.3	0.8250	0.947368	0.750000	0.837209
3	0.4	0.8250	0.894737	0.772727	0.829268
4	0.5	0.8375	0.868421	0.804878	0.835443
5	0.6	0.8625	0.868421	0.846154	0.857143
6	0.7	0.8250	0.789474	0.833333	0.810811
7	0.8	0.8375	0.763158	0.878788	0.816901

Note: The More F1_Score determines the better model, Multiple Answers based on the Client / Quality requirements, i.e based on the requirements the F1 Score selection might differ.

```
fpr, tpr, th = roc_curve(ytest, yprob1)

plt.plot(fpr, tpr)
```

```
[<matplotlib.lines.Line2D at 0x7d5e17586590>]
```




```
result = pd.DataFrame({'Threshold': th, 'FPR': fpr, 'TPR': tpr})
result
```

	Threshold	FPR	TPR
0	inf	0.000000	0.000000
1	0.999000	0.000000	0.026316
2	0.883078	0.000000	0.657895
3	0.880071	0.023810	0.657895
4	0.848147	0.023810	0.736842
5	0.844904	0.047619	0.736842
6	0.840400	0.047619	0.763158
7	0.780229	0.142857	0.763158
8	0.609693	0.142857	0.868421
9	0.481694	0.214286	0.868421
10	0.474650	0.214286	0.894737
11	0.449281	0.238095	0.894737
12	0.398504	0.238095	0.921053
13	0.382682	0.261905	0.921053
14	0.373589	0.261905	0.947368
15	0.307261	0.285714	0.947368
16	0.255786	0.285714	0.973684
17	0.087713	0.547619	0.973684
18	0.066045	0.547619	1.000000

```
result["Diff_Tpr_Fpr"] = result["TPR"] - result["FPR"]
result.sort_values("Diff_Tpr_Fpr", ascending=False)
```

	Threshold	FPR	TPR	Diff_Tpr_Fpr
8	0.609693	0.142857	0.868421	0.725564
6	0.840400	0.047619	0.763158	0.715539
4	0.848147	0.023810	0.736842	0.713033
5	0.844904	0.047619	0.736842	0.689223
16	0.255786	0.285714	0.973684	0.687970
14	0.373589	0.261905	0.947368	0.685464
12	0.398504	0.238095	0.921053	0.682957
10	0.474650	0.214286	0.894737	0.680451
15	0.307261	0.285714	0.947368	0.661654
13	0.382682	0.261905	0.921053	0.659148
2	0.883078	0.000000	0.657895	0.657895
11	0.449281	0.238095	0.894737	0.656642
9	0.481694	0.214286	0.868421	0.654135
3	0.880071	0.023810	0.657895	0.634085
7	0.780229	0.142857	0.763158	0.620301
18	0.066045	0.547619	1.000000	0.452381
17	0.087713	0.547619	0.973684	0.426065
1	0.999000	0.000000	0.026316	0.026316
0	inf	0.000000	0.000000	0.000000

```
# tot_cost = cfp * fp + cfn *fn
tot_cost = []
cfp = 4
cfn = 2
for i in th:
  ypred_th = np.zeros([len(ytest), 1])
  ypred_th[yprob1 > i] = 1
  tn = cm[0,0]
  fp = cm[0,1]
  fn = cm[1,0]
  tp = cm[1,1]

  tot_cost.append(cfp * fp + cfn * fn)
```

```
result["Total_Cost"] = tot_cost
result
```

	Threshold	FPR	TPR	Diff_Tpr_Fpr	Total_Cost
0	inf	0.000000	0.000000	0.000000	42
1	0.999000	0.000000	0.026316	0.026316	42
2	0.883078	0.000000	0.657895	0.657895	42
3	0.880071	0.023810	0.657895	0.634085	42
4	0.848147	0.023810	0.736842	0.713033	42
5	0.844904	0.047619	0.736842	0.689223	42
6	0.840400	0.047619	0.763158	0.715539	42
7	0.780229	0.142857	0.763158	0.620301	42
8	0.609693	0.142857	0.868421	0.725564	42
9	0.481694	0.214286	0.868421	0.654135	42
10	0.474650	0.214286	0.894737	0.680451	42
11	0.449281	0.238095	0.894737	0.656642	42
12	0.398504	0.238095	0.921053	0.682957	42
13	0.382682	0.261905	0.921053	0.659148	42
14	0.373589	0.261905	0.947368	0.685464	42
15	0.307261	0.285714	0.947368	0.661654	42
16	0.255786	0.285714	0.973684	0.687970	42
17	0.087713	0.547619	0.973684	0.426065	42
18	0.066045	0.547619	1.000000	0.452381	42

Naive Bayes Technique

Bayes' Theorem is a fundamental concept in probability theory and statistics. It provides a way to update the probability of an event based on new evidence or information.

Key Concepts:

- **Prior Probability (P(A)):** The initial belief or probability of an event A before any new evidence is considered.
- **Likelihood (P(B|A)):** The probability of observing evidence B given that event A is true.
- **Posterior Probability (P(A|B)):** The updated probability of event A after considering the evidence B.

- **Marginal Likelihood (P(B)):** The probability of observing evidence B, regardless of whether A is true or not.

Formula:

> P(A|B) = (P(B|A) * P(A)) / P(B)

In simpler terms:

Posterior Probability = (Likelihood * Prior Probability) / Marginal Likelihood (Evidence)

- **Naive** ---> This called as Naive Bayes Theorem because we Assume that the features are independent to each other, but they might have dependency each other. Even though there might be Dependency we are going to get good results.

```
b_cancer = pd.read_csv("https://raw.githubusercontent.com/HarpyTech/DSAI-MTech/refs/heads/main/ML2/bcancer.csv")

b_cancer.head()
```

	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_mean	concave points_mean	symmetry_mean	...	radius_wor
0	M	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001	0.14710	0.2419	...	25.
1	M	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869	0.07017	0.1812	...	24.
2	M	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974	0.12790	0.2069	...	23.
3	M	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414	0.10520	0.2597	...	14.
4	M	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980	0.10430	0.1809	...	22.

5 rows × 31 columns

```
cross_prob = b_cancer.diagnosis.value_counts(normalize=True)

cross_prob
```

diagnosis	proportion
B	0.627417
M	0.372583

```
P_B = cross_prob["B"]
P_M = cross_prob["M"]

print(P_B, P_M)
```

```
0.6274165202108963 0.37258347978910367
```

```
data = b_cancer[["diagnosis", "radius_mean", "texture_mean", "perimeter_mean", "area_mean"]]
data.head()
```

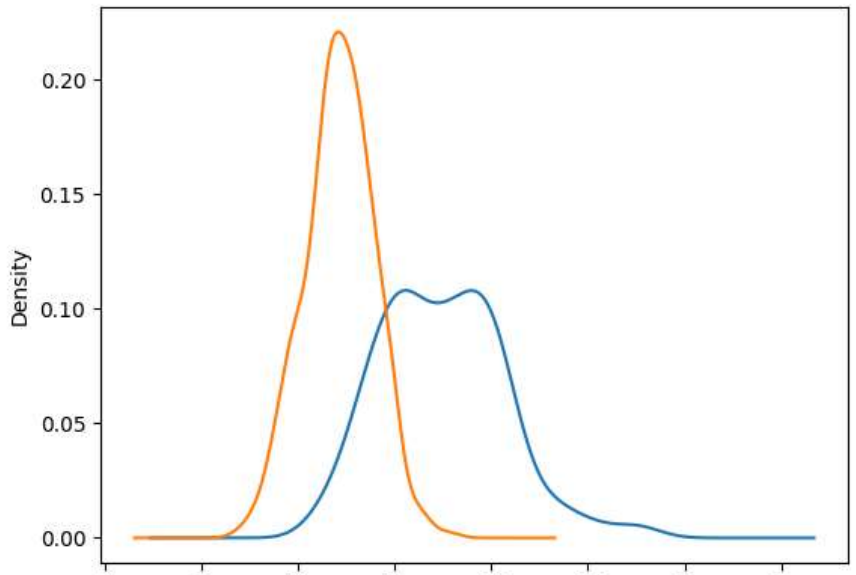
	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean
0	M	17.99	10.38	122.80	1001.0
1	M	20.57	17.77	132.90	1326.0
2	M	19.69	21.25	130.00	1203.0
3	M	11.42	20.38	77.58	386.1

```
data1 = data[data.diagnosis == "M"]
data2 = data[data.diagnosis == "B"]

print(data1.shape, data2.shape)
```

```
(212, 5) (357, 5)
```

```
from scipy import stats

data1['radius_mean'].plot(kind='kde')
data2['radius_mean'].plot(kind='kde')
```

```
<Axes: ylabel='Density'>
```




```
stats.gaussian_kde(data1['radius_mean']).pdf(15) # Probability Density of radius mean at 15
```

```
array([0.10534476])
```

```
data1.head(2)
```

	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean
0	M	17.99	10.38	122.8	1001.0

```
test = np.array(data1.iloc[0:1, 1:])[0]
test
```

```
num_m = stats.gaussian_kde(data1['radius_mean']).pdf(test[0]) * stats.gaussian_kde(data1['texture_mean']).pdf(test[1]) * stats.gaussian_kde(data1['perimeter_mean']).
num_b = stats.gaussian_kde(data2['radius_mean']).pdf(test[0]) * stats.gaussian_kde(data2['texture_mean']).pdf(test[1]) * stats.gaussian_kde(data2['perimeter_mean']).

print(num_m, num_b)

num_m > num_b
```

```
[1.29869883e-09] [1.00561393e-14]
```

```
array([ True])
```

```
test = np.array(data2.iloc[0:1, 1:])[0]
num_m = stats.gaussian_kde(data1['radius_mean']).pdf(test[0]) * stats.gaussian_kde(data1['texture_mean']).pdf(test[1]) * stats.gaussian_kde(data1['perimeter_mean']).
num_b = stats.gaussian_kde(data2['radius_mean']).pdf(test[0]) * stats.gaussian_kde(data2['texture_mean']).pdf(test[1]) * stats.gaussian_kde(data2['perimeter_mean']).

print(num_m, num_b)

num_m > num_b
```

```
[3.22339838e-09] [4.47645249e-07]
```

We observe that the probability of malignant is greater than the Probability of Benign, then the above test can be concluded as Good Probability (the test data belongs to Malignant) with the data1 test data for Malignant and vice versa for Beign.

SKLearn Usage

```
inp = data.drop('diagnosis', axis=1)
out = data.diagnosis

out.replace({"M": 1, "B": 0}, inplace=True)
inp.head(1), out.head(1)
```

```
<ipython-input-50-4136637c096d>:4: FutureWarning: Downcasting behavior in `replace` is deprecated and will be removed in a future version. To retain the old beha
  out.replace({"M": 1, "B": 0}, inplace=True)
<ipython-input-50-4136637c096d>:4: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  out.replace({"M": 1, "B": 0}, inplace=True)
(   radius_mean  texture_mean  perimeter_mean  area_mean
 0        17.99         10.38           122.8     1001.0,
 0    1
 Name: diagnosis, dtype: int64)
```

```
from sklearn.model_selection import train_test_split

xtrain, xtest, ytrain, ytest = train_test_split(inp, out, test_size=0.2, random_state=48)
```

- Different Types of Naive bayes Models used for diffrent data Input
  - Gaussian Naive Bayes --> Continuous Input data
  - Bernoulli NB --> Boolean input data
  - Multinominal NB --> Frequency Count input data

```
from sklearn.naive_bayes import GaussianNB, BernoulliNB, MultinomialNB

modelGB = GaussianNB()
modelGB.fit(xtrain, ytrain)

ypredGB = modelGB.predict(xtest)
```

```
print(classification_report(ytest, ypredGB))
```

```
              precision    recall  f1-score   support

           0       0.88      0.94      0.91        68
           1       0.90      0.80      0.85        46

    accuracy                           0.89       114
   macro avg       0.89      0.87      0.88       114
weighted avg       0.89      0.89      0.88       114
```

```
# Check the Overfittingg nature of the Model
# When Bias Error is Low, Variance error is High then the model is OverFitting
ypred_trainGB = modelGB.predict(xtrain)
ypred_testGB = modelGB.predict(xtest)

print("Performance of the Model for Trained Input. ", f1_score(ytrain, ypred_trainGB))
print("Performance of the Model for Test Input. ", f1_score(ytest, ypred_testGB))

print(classification_report(ytrain, ypred_trainGB))
print(classification_report(ytest, ypred_testGB))
```

```
Performance of the Model for Trained Input.  0.8376623376623377
Performance of the Model for Test Input.  0.8505747126436781
              precision    recall  f1-score   support

           0       0.88      0.96      0.92       289
           1       0.91      0.78      0.84       166

    accuracy                           0.89       455
   macro avg       0.90      0.87      0.88       455
weighted avg       0.89      0.89      0.89       455

              precision    recall  f1-score   support

           0       0.88      0.94      0.91        68
```

```
              1       0.90      0.80      0.85        46

       accuracy                           0.89       114
      macro avg       0.89      0.87      0.88       114
   weighted avg       0.89      0.89      0.88       114
```

```
score = cross_val_score(modelGB, inp, out, cv=5, scoring="f1")
score
```

```
np.mean(score)
```

```
np.std(score)
```

Cross-validation is a technique used to assess how well a model generalizes to unseen data. It involves splitting the data into multiple folds, training the model on a subset of the folds, and evaluating it on the remaining fold. This process is repeated multiple times, with each fold serving as the validation set once.

Standard Deviation and Overfitting

The standard deviation of the cross-validation scores can be a valuable indicator of overfitting. Here's how:

High Standard Deviation:

- ***Overfitting***: A high standard deviation suggests that the model's performance varies significantly across different folds. This could indicate that the model is sensitive to the specific data it's trained on, implying overfitting. In other words, the model's performance is not consistent across different subsets of the data.

Possible Causes:

- **Complex Model**: The model might be too complex for the given dataset, leading to overfitting.
- **Insufficient Data**: A small dataset can also lead to high variance, as the model might not have enough data to learn robust patterns.

  Low Standard Deviation:

Good Generalization: A low standard deviation generally indicates that the model's performance is consistent across different folds. This suggests that the model is likely to generalize well to unseen data.

NOTE Model is not overfitted, since the Standard Deviation too Low

```
Start coding or generate with AI.
```

Distance Based Learning

KNeighborsClassifier

```
admissions.head()
```

	GRE Score	TOEFL Score	University Rating	SOP	LOR	CGPA	Research	Chance of Admit
0	337	118	4	4.5	4.5	9.65	1	1
1	324	107	4	4.0	4.5	8.87	1	1
2	316	104	3	3.0	3.5	8.00	1	0
3	322	110	3	3.5	2.5	8.67	1	1

```
inp = admissions.drop(columns="Chance of Admit")
out = admissions["Chance of Admit"]
```

```
inp_sc = sc.fit_transform(inp)
inp_sc = pd.DataFrame(inp_sc, columns=inp.columns)
inp_sc.head()
```

	GRE Score	TOEFL Score	University Rating	SOP	LOR	CGPA	Research
0	1.762107	1.746971	0.798829	1.093864	1.167321	1.764818	0.909112
1	0.627656	-0.067635	0.798829	0.596653	1.167321	0.455151	0.909112
2	-0.070467	-0.562528	-0.076600	-0.397769	0.052933	-1.005631	0.909112
3	0.453126	0.427257	-0.076600	0.099442	-1.061454	0.119339	0.909112

```
xtrain, xtest, ytrain, ytest = train_test_split(inp_sc, out, test_size=0.2, random_state=48, stratify=out)
```

```
out.value_counts(normalize=True)
```

Chance of Admit	proportion
0	0.55
1	0.45

```
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(xtrain, ytrain)

ypred = knn.predict(xtest)

ypred
```

```
array([1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1,
       1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0,
       0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0,
       1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1])
```

```
print(classification_report(ytest, ypred))
```

```
              precision    recall  f1-score   support

           0       0.88      0.86      0.87        44
           1       0.84      0.86      0.85        36

    accuracy                           0.86        80
   macro avg       0.86      0.86      0.86        80
weighted avg       0.86      0.86      0.86        80
```

```
np.mean(cross_val_score(knn, inp_sc, out, cv=5, scoring="f1"))
```

If we are splitting the the data into 5 groups we are getting average F1 Score as ***0.83***

```
np.std(cross_val_score(knn, inp_sc, out, cv=5, scoring="f1"))
```

```
np.mean(cross_val_score(knn, inp_sc, out, cv=10, scoring="f1"))
```

```
np.mean(cross_val_score(knn, inp_sc, out, cv=15, scoring="f1"))
```

```
np.std(cross_val_score(knn, inp_sc, out, cv=5, scoring="f1")) / np.mean(cross_val_score(knn, inp_sc, out, cv=5, scoring="f1")) * 100
```

We are getting Coefficient of Variation less than ***5%***, which is accepted and good model.

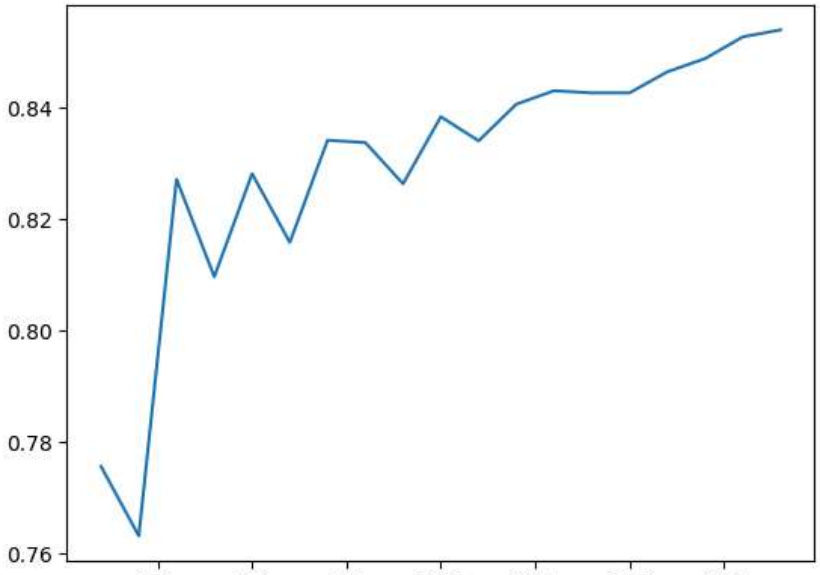
```
np.std(cross_val_score(knn, inp_sc, out, cv=15, scoring="f1")) / np.mean(cross_val_score(knn, inp_sc, out, cv=15, scoring="f1")) * 100
```

```
mean_score = []
std_score = []
var_score = []
k = np.arange(1, 20)

for i in k:
  knn = KNeighborsClassifier(n_neighbors=i)
  score = cross_val_score(knn, inp_sc, out, cv=5, scoring="f1")
  mean_score.append(np.mean(score))
  std_score.append(np.std(score))
  var_score.append(np.std(score) / np.mean(score) * 100)

result = pd.DataFrame({"K": k, "Mean_Score": mean_score, "Std_Score": std_score, "Var_Score": var_score})
result.sort_values("Var_Score", ascending=False)
```

	K	Mean_Score	Std_Score	Var_Score
1	2	0.763174	0.076188	9.982987
3	4	0.809594	0.056433	6.970498
2	3	0.827071	0.057303	6.928450
9	10	0.838269	0.055344	6.602202
13	14	0.842536	0.054476	6.465691
8	9	0.826258	0.053408	6.463868
11	12	0.840508	0.050206	5.973269
17	18	0.852582	0.050717	5.948619
5	6	0.815757	0.048253	5.915127
6	7	0.834054	0.048983	5.872822
18	19	0.853824	0.048642	5.697011
7	8	0.833635	0.046107	5.530821
15	16	0.846313	0.045654	5.394489
14	15	0.842551	0.045235	5.368829
12	13	0.842920	0.044712	5.304370
16	17	0.848690	0.044261	5.215237
0	1	0.775616	0.039878	5.141473
10	11	0.833941	0.039862	4.780010

```
plt.plot(k, mean_score)
```

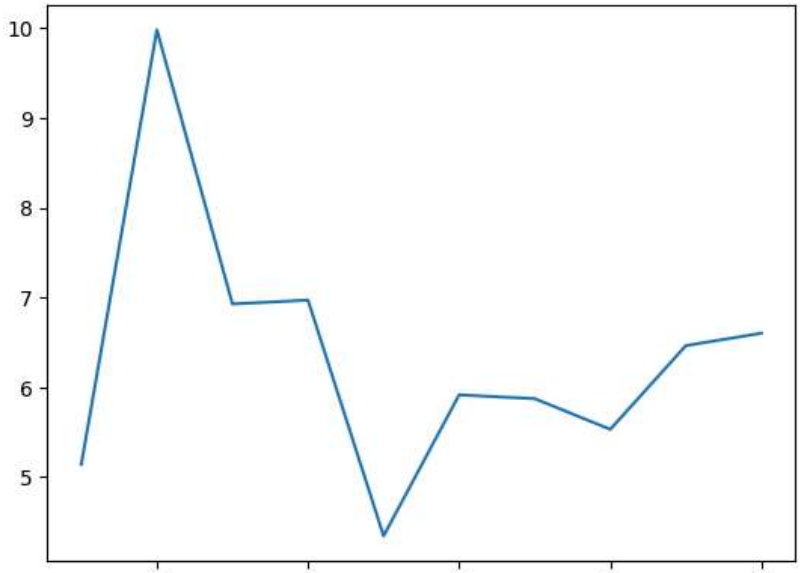
```
[<matplotlib.lines.Line2D at 0x7d5e1420b190>]
```



If we have to pick high values, identify the peak points, i.e 3, 5, 11

```
plt.plot(k[:10], var_score[:10])
```

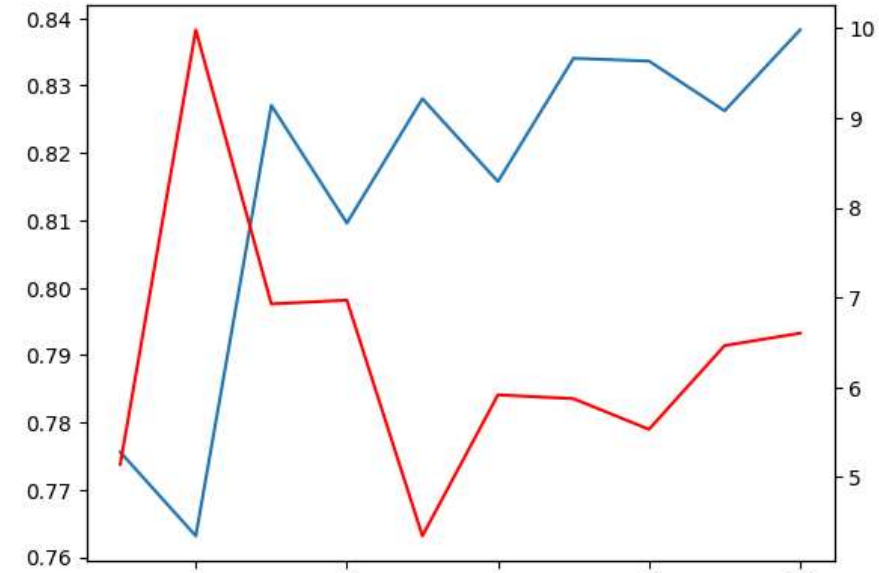
```
[<matplotlib.lines.Line2D at 0x7d5e6460e2f0>]
```



In the Variance Score we have to look at the Lesser Values and which have downfall in nature, i.e 3, 5, 8

```
fig, ax1 = plt.subplots()
ax1.plot(k[:10], mean_score[:10])
ax2 = ax1.twinx()
ax2.plot(k[:10], var_score[:10], color="red")
```

```
[<matplotlib.lines.Line2D at 0x7d5e14639b40>]
```




```
from sklearn.model_selection import GridSearchCV
model = KNeighborsClassifier()
params = {
    "n_neighbors": [3,5,7, 9, 11, 13, 15],
    "weights": ["uniform", "distance"],
    "p": [1,2,3,4,5]
}

grid = GridSearchCV(model, param_grid=params, cv=5, scoring="f1")
grid.fit(xtrain, ytrain)

grid.best_params_
```

```
{'n_neighbors': 3, 'p': 4, 'weights': 'uniform'}
```

```
result = pd.DataFrame(grid.cv_results_)
result.sort_values("std_test_score", ascending=False).head()
```

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_n_neighbors	param_p	param_weights	params	split0_test_score	split1_test_score	s
32	0.004398	0.000259	0.017629	0.005817	9	2	uniform	{'n_neighbors': 9, 'p': 2, 'weights': 'uniform'}	0.866667	0.827586	
42	0.010680	0.006963	0.040165	0.004953	11	2	uniform	{'n_neighbors': 11, 'p': 2, 'weights': 'uniform'}	0.881356	0.862069	
33	0.004287	0.000163	0.009428	0.001262	9	2	distance	{'n_neighbors': 9, 'p': 2, 'weights': 'distance'}	0.847458	0.821429	
40	0.008162	0.004484	0.037805	0.017856	11	1	uniform	{'n_neighbors': 11, 'p': 1, 'weights': 'uniform'}	0.819672	0.813559	
68	0.004531	0.000255	0.019943	0.002473	15	5	uniform	{'n_neighbors': 15, 'p': 5, 'weights': 'uniform'}	0.847458	0.821429	

```
grid = GridSearchCV(model, param_grid=params, cv=10, scoring="f1")
grid.fit(xtrain, ytrain)
result = pd.DataFrame(grid.cv_results_)
result.sort_values("std_test_score", ascending=False)
```

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_n_neighbors	param_p	param_weights	params	split0_test_score	split1_test_score	.
36	0.004100	0.000084	0.013543	0.001229	9	4	uniform	{'n_neighbors': 9, 'p': 4, 'weights': 'uniform'}	0.896552	0.81250	
34	0.004227	0.000236	0.013806	0.000842	9	3	uniform	{'n_neighbors': 9, 'p': 3, 'weights': 'uniform'}	0.896552	0.81250	
38	0.004128	0.000168	0.012866	0.000988	9	5	uniform	{'n_neighbors': 9, 'p': 5, 'weights': 'uniform'}	0.896552	0.81250	
37	0.004005	0.000135	0.009090	0.000192	9	4	distance	{'n_neighbors': 9, 'p': 4, 'weights': 'distance'}	0.896552	0.81250	
20	0.004462	0.001401	0.011498	0.001304	7	1	uniform	{'n_neighbors': 7, 'p': 1, 'weights': 'uniform'}	0.866667	0.83871	
...	...	...	...	...	...	...	...	...	...	...	
41	0.002617	0.000083	0.005112	0.000215	11	1	distance	{'n_neighbors': 11, 'p': 1, 'weights': 'distan...	0.827586	0.81250	
59	0.002789	0.000237	0.006268	0.000601	13	5	distance	{'n_neighbors': 13, 'p': 5, 'weights': 'distan...	0.857143	0.83871	
51	0.002505	0.000039	0.005006	0.000130	13	1	distance	{'n_neighbors': 13, 'p': 1, 'weights': 'distan...	0.827586	0.83871	
1	0.007557	0.005270	0.014012	0.008077	3	1	distance	{'n_neighbors': 3, 'p': 1, 'weights': 'distance'}	0.857143	0.81250	
0	0.005136	0.002548	0.017672	0.011901	3	1	uniform	{'n_neighbors': 3, 'p': 1, 'weights': 'uniform'}	0.857143	0.81250	

70 rows × 21 columns

```
test = np.array(xtest.loc[82, :])
test
```

```
array([0.2785948 , 0.42725722, 1.67425725, 1.59107523, 1.16732114,
```

```
test = test.reshape(1, -1)
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(xtrain, ytrain)

ypred = knn.predict(xtest)

knn.kneighbors(test)
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/utils/validation.py:2739: UserWarning: X does not have valid feature names, but KNeighborsClassifier was fitted w
  warnings.warn(
(array([[0.61767557, 0.70647406, 0.77228162, 0.7826315 , 0.84669372]]),
 array([[ 85, 255,  49, 194,   9]]))
```

Decision Tree Learning Technique

1. To Identify the Root Nodes
    - Gini Impurity - Lesser the Impurity to be eliminated.
    - Entropy -

How to calculate Gini Impurity:

$Gini_t = 1 - \sum(P_i)^2$

t -> dependent Feature sub Class

i -> Target Feature sub Class

```
SH = [Y, N, Y, N, Y]
Qual = [B, M, M, B, M]
cancer = [M, B, M, B, M]

Gini_for_SH = weighted_sum * (Gini_Imp_of_SH_Y + Gini_Imp_of_SH_N) # over target Variable
Gini_Imp_of_SH_Y = 1 - (Pm^2 + Pβ^2)
                 = 1 - ( (3/3)^2 + (0/3)^2 )
                 = 1 - (1-0)
                 = 0 # (Homogenious)
```

```
# Similarly for SH_N also 0 since all the N type observations are Benign and Y type are Malignant
```

Note: If we have got Gini Value 0 then we dont' need Machine Learning Algorithm for the respective DataSet or the Perticular Feature can be eliminated from the Feature Selection.

How To Calculate Entropy:

Formula:

```
Entropy (P_x) = ∑ -P_i * log(P_i)

i -> sub class of Target Feature
x -> sub class of Dependent Feature (Input Feature)
```

Identify the Threshold: - used for Numerical Features

1. arrage the column is ascending order
2. Find Moving Average with Windows as 1
3. For Each Level calculate the Gini Value
4. Among all the Gini Values, take the Least as Root Node

Important Inferences:

- Lesser the Gini value the Featrue is Good
- Lesser the Entropy the Feature is Good

Fully Grown Tree

A fully grown tree is one where every possible split has been made according to the training data, and each leaf node contains as few samples as necessary, potentially even just one sample.

Key Characteristics of a Fully Grown Decision Tree:

- ***No Pruning:*** The tree continues to split nodes until each node (leaf) in the tree contains only one unique sample or the data is perfectly classified. This means that no post-processing or pruning is performed to simplify the tree after it has been built.
- ***Overfitting Risk:*** A fully grown tree tends to overfit the training data. Since the tree perfectly fits the training data, it captures even the noise and small fluctuations in the data, leading to poor generalization to unseen data. This is a classic example of overfitting.
- ***Maximum Depth:*** There is no limit on the depth of the tree. The tree continues to grow deeper and deeper until every training sample is perfectly classified.
- ***Leaf Nodes with Few Samples:*** Each leaf node typically contains only a very small number of samples (or even a single sample), which can result in high variance in predictions and a lack of robustness when applied to new data.

Consider a dataset with 3 features, X1, X2, and X3. A fully grown tree might split as follows:

```
                Root
              X1 <= 2.5
             /         \
        X2 <= 1.5   X3 <= 0.5
          /    \         /    \
        X3<=1   X1<=3   X2<=0.3  X3<=0.7
```

At the leaves, you may have very specific classes (e.g., leaf 1 might represent Class A, leaf 2 Class B, etc.). Every internal node is a perfect split based on some threshold, leading to very high accuracy on the training data but potentially poor generalization.

Why Fully Grown Trees are Problematic:

- ***Overfitting:*** The most significant issue with fully grown trees is that they often overfit the training data. Since the tree learns every detail of the training data, it fails to generalize well to new, unseen data.
- ***High Variance:*** Because the tree has become so complex, it is highly sensitive to slight variations or noise in the training data.
- ***Inefficiency:*** A fully grown tree may require significant computational resources (memory, processing time) to store and compute predictions, especially for large datasets.

Solutions to Address Overfitting:

- ***Pruning:*** To prevent overfitting, decision trees are often pruned after they are fully grown. Pruning involves removing branches that contribute little to predictive power, effectively simplifying the tree. There are two main approaches to pruning:
  - **Pre-pruning (Early Stopping):** Limits the tree growth during training, such as by restricting the maximum depth, minimum samples per leaf, or minimum impurity decrease.
  - **Post-pruning:** After the tree has been fully grown, branches that don't contribute much to the accuracy of the model are removed.
- ***Regularization:*** Setting constraints such as limiting the depth of the tree, requiring a minimum number of samples per split, or limiting the number of leaf nodes can help control the complexity of the tree.
- ***Ensemble Methods:*** Using methods like Random Forests or Gradient Boosting can reduce overfitting by combining multiple decision trees, each trained on a different subset of the data or in a way that reduces variance.

```
out = admissions["Chance of Admit"]
inp = admissions.drop(columns='Chance of Admit')
```

```
inp_sc = sc.fit_transform(inp)
inp_sc = pd.DataFrame(inp_sc, columns=inp.columns)
inp_sc.head()

xtrain, xtest, ytrain, ytest = train_test_split(inp_sc, out, test_size=0.2, random_state=48, stratify=out)
```

```
ytest.value_counts(normalize=True)
```

Chance of Admit	proportion
0	0.55
1	0.45

```
from sklearn.tree import DecisionTreeClassifier

decision_tree = DecisionTreeClassifier()
decision_tree.fit(xtrain, ytrain)

ypred = decision_tree.predict(xtest)
ypred
```

```
array([1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1,
       1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0,
       1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0,
       1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0])
```

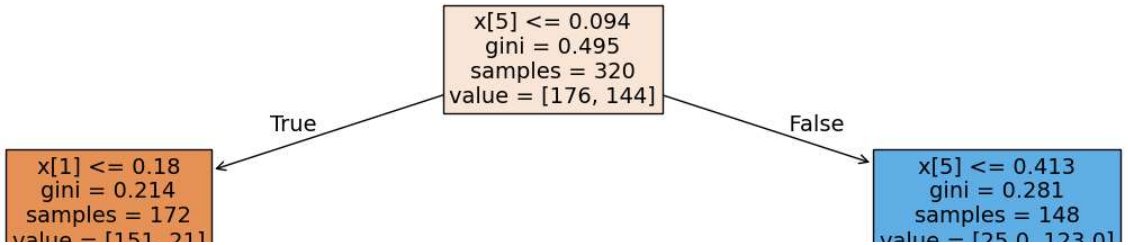
```
print(classification_report(ytest, ypred))
```

```
              precision    recall  f1-score   support

           0       0.89      0.91      0.90        44
           1       0.89      0.86      0.87        36

    accuracy                           0.89        80
   macro avg       0.89      0.89      0.89        80
weighted avg       0.89      0.89      0.89        80
```

```
from sklearn import tree

plt.figure(figsize=(20, 20))
tree.plot_tree(decision_tree, filled=True, fontsize=14)
```

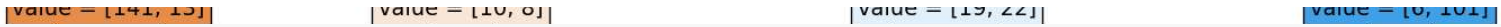
```
[Text(0.5096153846153846, 0.9583333333333334, 'x[5] <= 0.094\ngini = 0.495\nsamples = 320\nvalue = [176, 144]'),
 Text(0.2548076923076923, 0.875, 'x[1] <= 0.18\ngini = 0.214\nsamples = 172\nvalue = [151, 21]'),
 Text(0.38221153846153844, 0.9166666666666667, 'True  '),
 Text(0.15384615384615385, 0.7916666666666666, 'x[5] <= -0.275\ngini = 0.155\nsamples = 154\nvalue = [141, 13]'),
 Text(0.07692307692307693, 0.7083333333333334, 'x[4] <= 0.889\ngini = 0.064\nsamples = 120\nvalue = [116, 4]'),
 Text(0.038461538461538464, 0.625, 'x[1] <= -0.315\ngini = 0.034\nsamples = 115\nvalue = [113, 2]'),
 Text(0.019230769230769232, 0.5416666666666666, 'gini = 0.0\nsamples = 88\nvalue = [88, 0]'),
 Text(0.057692307692307696, 0.5416666666666666, 'x[6] <= -0.095\ngini = 0.137\nsamples = 27\nvalue = [25, 2]'),
 Text(0.038461538461538464, 0.4583333333333333, 'gini = 0.0\nsamples = 18\nvalue = [18, 0]'),
 Text(0.07692307692307693, 0.4583333333333333, 'x[0] <= -0.463\ngini = 0.346\nsamples = 9\nvalue = [7, 2]'),
 Text(0.057692307692307696, 0.375, 'x[1] <= -0.15\ngini = 0.444\nsamples = 3\nvalue = [1, 2]'),
 Text(0.038461538461538464, 0.2916666666666667, 'gini = 0.0\nsamples = 2\nvalue = [0, 2]'),
 Text(0.07692307692307693, 0.2916666666666667, 'gini = 0.0\nsamples = 1\nvalue = [1, 0]'),
 Text(0.09615384615384616, 0.375, 'gini = 0.0\nsamples = 6\nvalue = [6, 0]'),
 Text(0.11538461538461539, 0.625, 'x[5] <= -0.477\ngini = 0.48\nsamples = 5\nvalue = [3, 2]'),
 Text(0.09615384615384616, 0.5416666666666666, 'gini = 0.0\nsamples = 1\nvalue = [0, 1]'),
 Text(0.1346153846153846, 0.5416666666666666, 'x[2] <= 0.361\ngini = 0.375\nsamples = 4\nvalue = [3, 1]'),
 Text(0.11538461538461539, 0.4583333333333333, 'gini = 0.0\nsamples = 3\nvalue = [3, 0]'),
 Text(0.15384615384615385, 0.4583333333333333, 'gini = 0.0\nsamples = 1\nvalue = [0, 1]'),
 Text(0.23076923076923078, 0.7083333333333334, 'x[5] <= -0.258\ngini = 0.389\nsamples = 34\nvalue = [25, 9]'),
 Text(0.21153846153846154, 0.625, 'gini = 0.0\nsamples = 2\nvalue = [0, 2]'),
 Text(0.25, 0.625, 'x[3] <= 1.342\ngini = 0.342\nsamples = 32\nvalue = [25, 7]'),
 Text(0.21153846153846154, 0.5416666666666666, 'x[1] <= -1.305\ngini = 0.285\nsamples = 29\nvalue = [24, 5]'),
 Text(0.19230769230769232, 0.4583333333333333, 'gini = 0.0\nsamples = 1\nvalue = [0, 1]'),
 Text(0.23076923076923078, 0.4583333333333333, 'x[5] <= -0.124\ngini = 0.245\nsamples = 28\nvalue = [24, 4]'),
 Text(0.21153846153846154, 0.375, 'x[5] <= -0.149\ngini = 0.426\nsamples = 13\nvalue = [9, 4]'),
 Text(0.19230769230769232, 0.2916666666666667, 'x[2] <= -0.514\ngini = 0.375\nsamples = 12\nvalue = [9, 3]'),
 Text(0.17307692307692307, 0.20833333333333334, 'gini = 0.0\nsamples = 5\nvalue = [5, 0]'),
 Text(0.21153846153846154, 0.20833333333333334, 'x[0] <= -0.463\ngini = 0.49\nsamples = 7\nvalue = [4, 3]'),
 Text(0.19230769230769232, 0.125, 'gini = 0.0\nsamples = 3\nvalue = [3, 0]'),
 Text(0.23076923076923078, 0.125, 'x[0] <= 0.235\ngini = 0.375\nsamples = 4\nvalue = [1, 3]'),
 Text(0.21153846153846154, 0.041666666666666664, 'gini = 0.0\nsamples = 3\nvalue = [0, 3]'),
 Text(0.25, 0.041666666666666664, 'gini = 0.0\nsamples = 1\nvalue = [1, 0]'),
 Text(0.23076923076923078, 0.2916666666666667, 'gini = 0.0\nsamples = 1\nvalue = [0, 1]'),
 Text(0.25, 0.375, 'gini = 0.0\nsamples = 15\nvalue = [15, 0]'),
 Text(0.28846153846153844, 0.5416666666666666, 'x[2] <= 0.799\ngini = 0.444\nsamples = 3\nvalue = [1, 2]'),
 Text(0.2692307692307692, 0.4583333333333333, 'gini = 0.0\nsamples = 1\nvalue = [0, 1]'),
 Text(0.3076923076923077, 0.4583333333333333, 'x[0] <= -1.161\ngini = 0.5\nsamples = 2\nvalue = [1, 1]'),
 Text(0.28846153846153844, 0.375, 'gini = 0.0\nsamples = 1\nvalue = [0, 1]'),
 Text(0.3269230769230769, 0.375, 'gini = 0.0\nsamples = 1\nvalue = [1, 0]'),
 Text(0.3557692307692308, 0.7916666666666666, 'x[0] <= -0.114\ngini = 0.494\nsamples = 18\nvalue = [10, 8]'),
 Text(0.3076923076923077, 0.7083333333333334, 'x[3] <= 0.348\ngini = 0.245\nsamples = 7\nvalue = [6, 1]'),
 Text(0.28846153846153844, 0.625, 'gini = 0.0\nsamples = 6\nvalue = [6, 0]'),
 Text(0.3269230769230769, 0.625, 'gini = 0.0\nsamples = 1\nvalue = [0, 1]'),
 Text(0.40384615384615385, 0.7083333333333334, 'x[6] <= -0.095\ngini = 0.463\nsamples = 11\nvalue = [4, 7]'),
 Text(0.36538461538461536, 0.625, 'x[0] <= 0.104\ngini = 0.375\nsamples = 4\nvalue = [3, 1]'),
 Text(0.34615384615384615, 0.5416666666666666, 'gini = 0.0\nsamples = 1\nvalue = [0, 1]'),
 Text(0.38461538461538464, 0.5416666666666666, 'gini = 0.0\nsamples = 3\nvalue = [3, 0]'),
 Text(0.4423076923076923, 0.625, 'x[1] <= 1.005\ngini = 0.245\nsamples = 7\nvalue = [1, 6]'),
 Text(0.4230769230769231, 0.5416666666666666, 'gini = 0.0\nsamples = 5\nvalue = [0, 5]'),
 Text(0.46153846153846156, 0.5416666666666666, 'x[5] <= -0.535\ngini = 0.5\nsamples = 2\nvalue = [1, 1]'),
 Text(0.4423076923076923, 0.4583333333333333, 'gini = 0.0\nsamples = 1\nvalue = [0, 1]'),
 Text(0.4807692307692308, 0.4583333333333333, 'gini = 0.0\nsamples = 1\nvalue = [1, 0]'),
 Text(0.7644230769230769, 0.875, 'x[5] <= 0.413\ngini = 0.281\nsamples = 148\nvalue = [25.0, 123.0]'),
 Text(0.6370192307692307, 0.9166666666666667, '  False'),
 Text(0.6346153846153846, 0.7916666666666666, 'x[0] <= 0.409\ngini = 0.497\nsamples = 41\nvalue = [19, 22]'),
```

```
Text(0.5576923076923077, 0.7083333333333334, 'x[4] <= -1.34\ngini = 0.463\nsamples = 22\nvalue = [14, 8]'),
Text(0.5384615384615384, 0.625, 'gini = 0.0\nsamples = 1\nvalue = [0, 1]'),
Text(0.5769230769230769, 0.625, 'x[5] <= 0.363\ngini = 0.444\nsamples = 21\nvalue = [14, 7]'),
Text(0.5576923076923077, 0.5416666666666666, 'x[1] <= 0.015\ngini = 0.42\nsamples = 20\nvalue = [14, 6]'),
Text(0.5192307692307693, 0.4583333333333333, 'x[3] <= -0.149\ngini = 0.496\nsamples = 11\nvalue = [6, 5]'),
Text(0.5, 0.375, 'gini = 0.0\nsamples = 3\nvalue = [3, 0]'),
Text(0.5384615384615384, 0.375, 'x[5] <= 0.145\ngini = 0.469\nsamples = 8\nvalue = [3, 5]'),
Text(0.5192307692307693, 0.2916666666666667, 'gini = 0.0\nsamples = 3\nvalue = [0, 3]'),
Text(0.5576923076923077, 0.2916666666666667, 'x[1] <= -0.15\ngini = 0.48\nsamples = 5\nvalue = [3, 2]'),
Text(0.5384615384615384, 0.20833333333333334, 'x[3] <= 0.348\ngini = 0.444\nsamples = 3\nvalue = [1, 2]'),
Text(0.5192307692307693, 0.125, 'gini = 0.0\nsamples = 1\nvalue = [1, 0]'),
Text(0.5576923076923077, 0.125, 'gini = 0.0\nsamples = 2\nvalue = [0, 2]'),
Text(0.5769230769230769, 0.20833333333333334, 'gini = 0.0\nsamples = 2\nvalue = [2, 0]'),
Text(0.5961538461538461, 0.4583333333333333, 'x[0] <= -0.245\ngini = 0.198\nsamples = 9\nvalue = [8, 1]'),
Text(0.5769230769230769, 0.375, 'gini = 0.0\nsamples = 1\nvalue = [0, 1]'),
Text(0.6153846153846154, 0.375, 'gini = 0.0\nsamples = 8\nvalue = [8, 0]'),
Text(0.5961538461538461, 0.5416666666666666, 'gini = 0.0\nsamples = 1\nvalue = [0, 1]'),
Text(0.7115384615384616, 0.7083333333333334, 'x[3] <= -0.149\ngini = 0.388\nsamples = 19\nvalue = [5, 14]'),
Text(0.6730769230769231, 0.625, 'x[4] <= -0.504\ngini = 0.48\nsamples = 5\nvalue = [3, 2]'),
Text(0.6538461538461539, 0.5416666666666666, 'x[4] <= -1.897\ngini = 0.444\nsamples = 3\nvalue = [1, 2]'),
Text(0.6346153846153846, 0.4583333333333333, 'gini = 0.0\nsamples = 1\nvalue = [1, 0]'),
Text(0.6730769230769231, 0.4583333333333333, 'gini = 0.0\nsamples = 2\nvalue = [0, 2]'),
Text(0.6923076923076923, 0.5416666666666666, 'gini = 0.0\nsamples = 2\nvalue = [2, 0]'),
Text(0.75, 0.625, 'x[1] <= 1.087\ngini = 0.245\nsamples = 14\nvalue = [2, 12]'),
Text(0.7307692307692307, 0.5416666666666666, 'x[5] <= 0.136\ngini = 0.142\nsamples = 13\nvalue = [1, 12]'),
Text(0.7115384615384616, 0.4583333333333333, 'x[0] <= 0.671\ngini = 0.375\nsamples = 4\nvalue = [1, 3]'),
Text(0.6923076923076923, 0.375, 'gini = 0.0\nsamples = 2\nvalue = [0, 2]'),
Text(0.7307692307692307, 0.375, 'x[1] <= 0.675\ngini = 0.5\nsamples = 2\nvalue = [1, 1]'),
Text(0.7115384615384616, 0.2916666666666667, 'gini = 0.0\nsamples = 1\nvalue = [1, 0]'),
Text(0.75, 0.2916666666666667, 'gini = 0.0\nsamples = 1\nvalue = [0, 1]'),
Text(0.75, 0.4583333333333333, 'gini = 0.0\nsamples = 9\nvalue = [0, 9]'),
Text(0.7692307692307693, 0.5416666666666666, 'gini = 0.0\nsamples = 1\nvalue = [1, 0]'),
Text(0.8942307692307693, 0.7916666666666666, 'x[1] <= -0.15\ngini = 0.106\nsamples = 107\nvalue = [6, 101]'),
Text(0.8461538461538461, 0.7083333333333334, 'x[1] <= -0.398\ngini = 0.48\nsamples = 5\nvalue = [3, 2]'),
Text(0.8269230769230769, 0.625, 'x[1] <= -1.14\ngini = 0.444\nsamples = 3\nvalue = [1, 2]'),
Text(0.8076923076923077, 0.5416666666666666, 'gini = 0.0\nsamples = 1\nvalue = [1, 0]'),
Text(0.8461538461538461, 0.5416666666666666, 'gini = 0.0\nsamples = 2\nvalue = [0, 2]'),
Text(0.8653846153846154, 0.625, 'gini = 0.0\nsamples = 2\nvalue = [2, 0]'),
Text(0.9423076923076923, 0.7083333333333334, 'x[3] <= 0.348\ngini = 0.057\nsamples = 102\nvalue = [3, 99]'),
Text(0.9230769230769231, 0.625, 'x[6] <= -0.095\ngini = 0.278\nsamples = 18\nvalue = [3, 15]'),
Text(0.8846153846153846, 0.5416666666666666, 'x[1] <= 0.592\ngini = 0.48\nsamples = 5\nvalue = [2, 3]'),
Text(0.8653846153846154, 0.4583333333333333, 'gini = 0.0\nsamples = 3\nvalue = [0, 3]'),
Text(0.9038461538461539, 0.4583333333333333, 'gini = 0.0\nsamples = 2\nvalue = [2, 0]'),
Text(0.9615384615384616, 0.5416666666666666, 'x[0] <= 0.235\ngini = 0.142\nsamples = 13\nvalue = [1, 12]'),
Text(0.9423076923076923, 0.4583333333333333, 'gini = 0.0\nsamples = 1\nvalue = [1, 0]'),
Text(0.9807692307692307, 0.4583333333333333, 'gini = 0.0\nsamples = 12\nvalue = [0, 12]'),
Text(0.9615384615384616, 0.625, 'gini = 0.0\nsamples = 84\nvalue = [0, 84]')]
```



```
len(xtrain[xtrain.CGPA <= 0.094]) # because the column at x[5] is CGPA
```

```
172
```

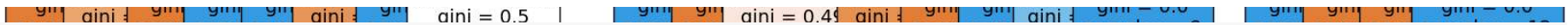



```
from sklearn.model_selection import cross_val_score, KFold

dt = DecisionTreeClassifier(random_state=48)
kf = KFold(n_splits=5, shuffle=True, random_state=48)

score = cross_val_score(dt, inp_sc, out, cv=kf, scoring="f1")
score
```

```
array([0.78378378, 0.8125    , 0.7826087 , 0.78378378, 0.75675676])
```

```
# co-efficient of variation
covar = score.std() / score.mean() * 100
covar, score.mean(), score.std()
```

```
(2.250731321627394, 0.7838866039952996, 0.017643181322163505)
```

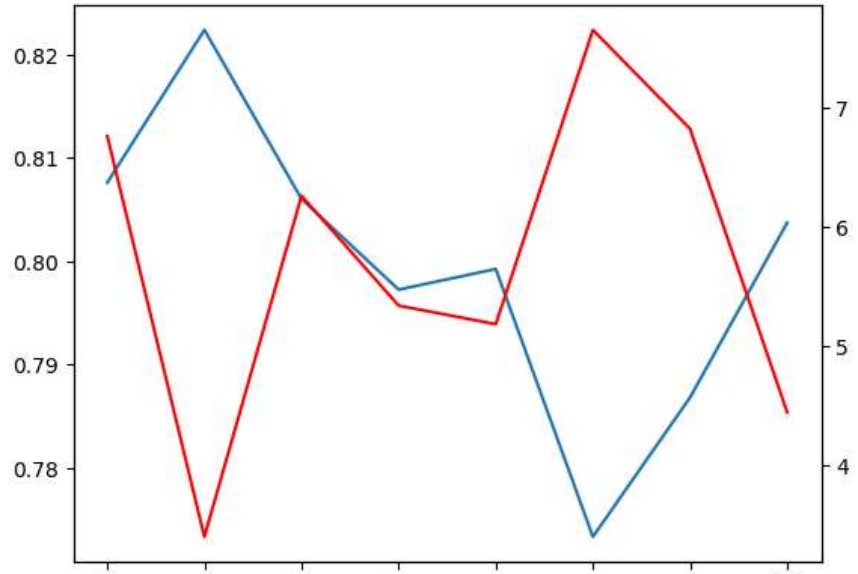



```
# Tuning the model for better performace with max-Depth
mean_score = []
std_score = []
covar_score = []
max_depth = [3,4,5,6,7,8,9,10]

for k in max_depth:
  dt_mod = DecisionTreeClassifier(max_depth=k, random_state=48)
  score = cross_val_score(dt_mod, inp, out, cv=5, scoring="f1")
  mean_score.append(score.mean())
  std_score.append(score.std())
  covar_score.append(score.std() / score.mean()*100)

result = pd.DataFrame({"Max_Depth": max_depth, "Mean_Score": mean_score, "Std_Score": std_score, "Covar_Score": covar_score})
result
```

	Max_Depth	Mean_Score	Std_Score	Covar_Score
0	3	0.807605	0.054586	6.759004
1	4	0.822368	0.028005	3.405410
2	5	0.806024	0.050444	6.258401
3	6	0.797252	0.042570	5.339550
4	7	0.799247	0.041437	5.184557
5	8	0.773348	0.059136	7.646731
6	9	0.786820	0.053659	6.819759

```
fig, ax = plt.subplots()
ax.plot(max_depth, mean_score)
ax2 = ax.twinx()
ax2.plot(max_depth, covar_score, color="red")
```

```
[<matplotlib.lines.Line2D at 0x7d5e173d0670>]
```



At Max Depth we are getting better model, compared to others.

```
params = {
    "max_depth": [3,4,5,6,7,8,9,10], # Pre pruning Condition
```

```
    "min_samples_split": [10, 15, 20, 25], #
    "criterion": ["gini", "entropy"]
}
dt = DecisionTreeClassifier()
kf = KFold(n_splits=5, shuffle=True, random_state=48)

grid = GridSearchCV(dt, param_grid=params, cv=kf, scoring="f1")
grid.fit(xtrain, ytrain)

grid.best_params_
```

```
{'criterion': 'entropy', 'max_depth': 9, 'min_samples_split': 10}
```

```
res = pd.DataFrame(grid.cv_results_)
res.sort_values("mean_test_score", ascending=False)
```

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_criterion	param_max_depth	param_min_samples_split	params	split0_test_score	s
56	0.002907	0.000047	0.003861	0.000025	entropy	9	10	{'criterion': 'entropy', 'max_depth': 9, 'min_...	0.811594	
48	0.003727	0.000851	0.004857	0.000783	entropy	7	10	{'criterion': 'entropy', 'max_depth': 7, 'min_...	0.800000	
52	0.003085	0.000091	0.004116	0.000130	entropy	8	10	{'criterion': 'entropy', 'max_depth': 8, 'min_...	0.811594	
44	0.003147	0.000139	0.004288	0.000111	entropy	6	10	{'criterion': 'entropy', 'max_depth': 6, 'min_...	0.833333	
60	0.003021	0.000120	0.004123	0.000247	entropy	10	10	{'criterion': 'entropy', 'max_depth': 10, 'min...	0.811594	
...	...	...	...	...	...	...	...	...	...	
22	0.003613	0.000604	0.004755	0.001178	gini	8	20	{'criterion': 'gini', 'max_depth': 8, 'min_sam...	0.852941	
21	0.003215	0.000723	0.004767	0.001092	gini	8	15	{'criterion': 'gini', 'max_depth': 8, 'min_sam...	0.865672	
6	0.002919	0.000295	0.004090	0.000197	gini	4	20	{'criterion': 'gini', 'max_depth': 4, 'min_sam...	0.840580	
9	0.002971	0.000223	0.004118	0.000152	gini	5	15	{'criterion': 'gini', 'max_depth': 5, 'min_sam...	0.852941	
10	0.002722	0.000110	0.003935	0.000180	gini	5	20	{'criterion': 'gini', 'max_depth': 5, 'min_sam...	0.852941	

64 rows × 16 columns

```
grid.best_params_
```

```
{'criterion': 'entropy', 'max_depth': 9, 'min_samples_split': 10}
```

```
final_dt = DecisionTreeClassifier(max_depth=4, min_samples_split=10, criterion="entropy")
final_dt.fit(xtrain, ytrain)
```

```
DecisionTreeClassifier
```

```
final_dt = DecisionTreeClassifier(max_depth=7, min_samples_split=10, criterion="entropy")
final_dt.fit(xtrain, ytrain)

ypred = final_dt.predict(xtest) # Best parameters from the above Code
print(classification_report(ytest, ypred))
```

```
              precision    recall  f1-score   support

           0       0.87      0.93      0.90        44
           1       0.91      0.83      0.87        36

    accuracy                           0.89        80
   macro avg       0.89      0.88      0.89        80
weighted avg       0.89      0.89      0.89        80
```

```
ypred_best = grid.best_estimator_.predict(xtest)
print(classification_report(ytest, ypred_best))
```

```
              precision    recall  f1-score   support

           0       0.89      0.93      0.91        44
           1       0.91      0.86      0.89        36

    accuracy                           0.90        80
   macro avg       0.90      0.90      0.90        80
weighted avg       0.90      0.90      0.90        80
```

```
ypred_dt = decision_tree.predict(xtest) # Base Model Performace
print(classification_report(ytest, ypred_dt))
```

```
              precision    recall  f1-score   support

           0       0.89      0.91      0.90        44
           1       0.89      0.86      0.87        36

    accuracy                           0.89        80
```

```
          macro avg       0.89      0.89      0.89        80
       weighted avg       0.89      0.89      0.89        80
```

```
final_dt = DecisionTreeClassifier(max_depth=4, min_samples_split=10, criterion="entropy")
final_dt.fit(xtrain, ytrain)

ypred = final_dt.predict(xtest) # Best parameters from the Faculty Code
print(classification_report(ytest, ypred))
```

```
              precision    recall  f1-score   support

           0       0.91      0.93      0.92        44
           1       0.91      0.89      0.90        36

    accuracy                           0.91        80
   macro avg       0.91      0.91      0.91        80
weighted avg       0.91      0.91      0.91        80
```

11 Jan 2025

Ensembling Algorithm

1. Bagging
2. Random Forest
3. Boosting
4. Voting
5. Stacking

1. **Bagging (Bootstrap Aggregating)**

Bagging is an ensemble technique where multiple models (typically the same type, like decision trees) are trained independently on different subsets of the training data. The subsets are created by **bootstrapping**, which means randomly sampling the training data with replacement.

- **How it works**:
  - Multiple models are trained in parallel on different random subsets of the training dataset.
  - For classification, the final prediction is made by **voting** (majority vote) across the individual models.
  - For regression, the final prediction is the **average** of the individual model predictions.
- **Key characteristic**: Bagging reduces **variance** by averaging out the predictions from multiple models.
- **Example**: **Random Forest** (which is a specific instance of bagging with decision trees) is one of the most popular examples of bagging.

2. **Random Forest**

Random Forest is a specific implementation of bagging where multiple decision trees are trained on random subsets of the data, but with an additional randomness introduced in the feature selection process.

- **How it works**:
  - Each tree in the forest is trained on a different bootstrapped sample of the data (like in bagging).
  - When splitting nodes in each tree, a random subset of features is considered, instead of all features, to create more diverse trees.
- **Key characteristic**: Random Forests reduce both **variance** and **bias** by combining the predictions of many diverse trees.
- **Example**: Random Forest is widely used for classification and regression tasks due to its robustness and ability to handle large datasets with many features.

3. **Boosting**

Boosting is an ensemble technique that builds models sequentially. Each subsequent model attempts to correct the errors made by the previous models, giving more weight to the instances that were previously misclassified.

- **How it works**:
  - Boosting trains models **sequentially**, where each model is trained to correct the mistakes of the previous one.
  - The models are weighted based on their performance, and the final prediction is a **weighted vote** or **weighted average** of all the models.
- **Key characteristic**: Boosting reduces **bias** and typically increases the accuracy of weak models by focusing on harder-to-predict instances.
- **Examples**:
  - **AdaBoost**: Adjusts the weights of misclassified samples to focus learning on difficult cases.
  - **Gradient Boosting** (e.g., **XGBoost**, **LightGBM**, **CatBoost**): Builds trees sequentially, with each new tree correcting the errors of the previous one by optimizing the residual errors.

4. **Voting**

Voting is a simple ensembling method where multiple models are trained independently, and their predictions are combined by **voting** (for classification) or **averaging** (for regression).

- **How it works**:
  - In classification, each model makes a prediction, and the final prediction is determined by the **majority vote** (for hard voting) or by averaging the predicted probabilities (for soft voting).
  - In regression, the final prediction is the **average** of the individual model predictions.
- **Key characteristic**: Voting is typically used with a mix of different model types (e.g., decision trees, SVMs, neural networks) to leverage the strengths of each.

- **Example**: A **Voting Classifier** may combine the outputs of logistic regression, decision trees, and SVMs to make a final prediction.

5. **Stacking**

Stacking (or **stacked generalization**) is an advanced ensembling technique where multiple models are trained, and their predictions are used as input for a **meta-model** (also called a **stacker**), which makes the final prediction.

- **How it works**:
  - First, multiple base-level models are trained on the training data.
  - Then, a **meta-model** (usually a logistic regression or another model) is trained on the predictions of the base models.
  - The meta-model learns how to combine the predictions of the base models to improve accuracy.
- **Key characteristic**: Stacking can handle different types of models, and the meta-model can learn how to best combine the predictions from the base models to reduce both **bias** and **variance**.
- **Example**: A stacking ensemble might combine the predictions of decision trees, SVMs, and neural networks, with a logistic regression model acting as the meta-model to combine their outputs effectively.

Summary of Key Differences:

Method	Training Process	Focus	Main Use Case
Bagging	Train models independently on bootstrapped data	Reduces variance	High variance models (e.g., decision trees)
Random Forest	Bagging with random feature selection for each tree	Reduces variance and bias	Robust classification and regression tasks
Boosting	Sequential model training, correcting errors	Reduces bias	Tasks requiring high accuracy, often with weak learners
Voting	Combine predictions via voting (classification) or averaging (regression)	Leverages multiple model types	Combining diverse models for improved generalization
Stacking	Train base models, then use their predictions for a meta-model	Combines predictions of multiple models for optimal result	Complex models with diverse base models and strong meta-model

Note:

- In Bagging only Row Sampling is Available
- In Random Forest, both Row Sampling and Column Sampling is available

```
All the Trees are constucted parallely
```

```
plant_data = pd.read_csv("plant_csv.csv")
plant_data.head()
```

	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	...	F15	F16	F17	F18	F19	F20
0	0.559318	0.168862	7.257456	0.013264	2.442463	0.338663	0.909578	0.124971	0.863168	0.124971	...	10.004906	107.249273	2.179754	3.523551	0.282158	0.655862
1	0.579257	0.159956	7.120799	0.297232	2.166805	0.314408	0.908118	0.150460	0.871048	0.150460	...	10.235738	111.299146	2.063268	3.340540	0.262196	0.634552
2	0.597987	0.189701	7.337628	-0.579806	2.710970	0.367551	0.922431	0.119970	0.858656	0.119970	...	10.558140	120.576853	2.224622	3.616883	0.307822	0.675735