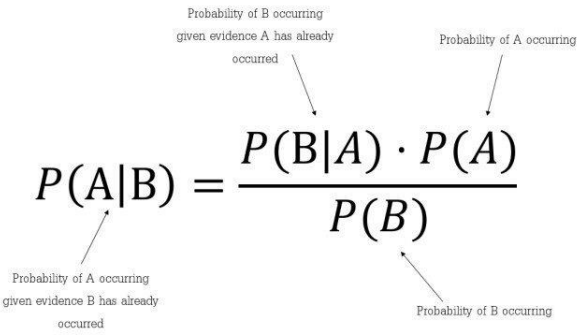
Experiment-5: Implementation of Naive Bayesian Classifier for a sample training data set stored as a .csv file. Calculate accuracy, precision and recall for your dataset.

Description :

Naive Bayes classifiers are a collection of classification algorithms based on Bayes' Theorem. It is not a single algorithm but a family of algorithms where all of them share a common principle, i.e. every pair of features being classified is independent of each other.

Bayes' Theorem finds the probability of an event occurring given the probability of another event that has already occurred. Bayes' theorem is stated mathematically as the following equation:



Code:

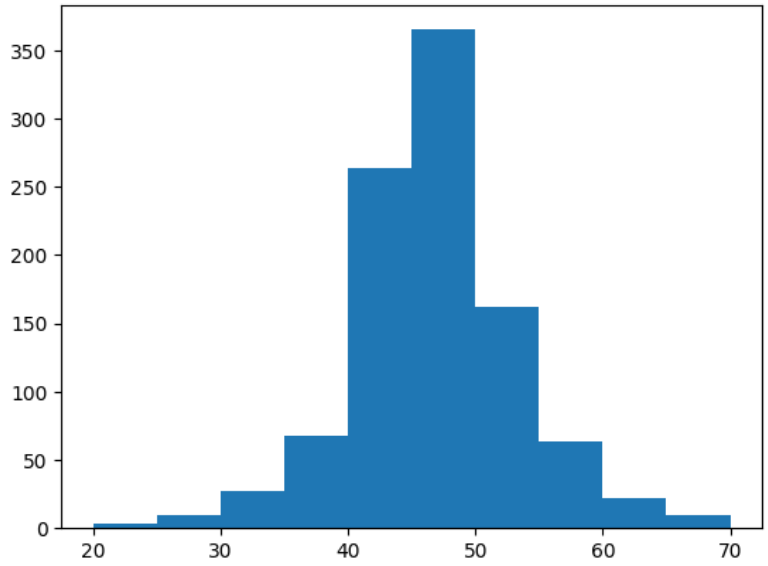
```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
import matplotlib.pyplot as plt
```

```
df = pd.read_csv('Naive-Bayes-Classification-Data.csv')
df.head()
```

	glucose	bloodpressure	diabetes
0	40	85	0
1	40	92	0
2	45	63	1

	glucose	bloodpressure	diabetes
3	45	80	0
4	40	73	1

```
plt.hist(df.glucose)
```




```
X = df.iloc[:, :-1].values
y = df.iloc[:, 2].values

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.
33, random_state=42)

model = GaussianNB()
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
print(model.predict([[45, 100]]))
```

```
[0]
```

```
from sklearn.metrics import accuracy_score, precision_score, recall_sc
ore
accuracy = accuracy_score(y_test, y_pred) * 100
precision = precision_score(y_test, y_pred) * 100
recall = recall_score(y_test, y_pred) * 100
```

```
print(accuracy)
print(precision)
print(recall)
```

Output:

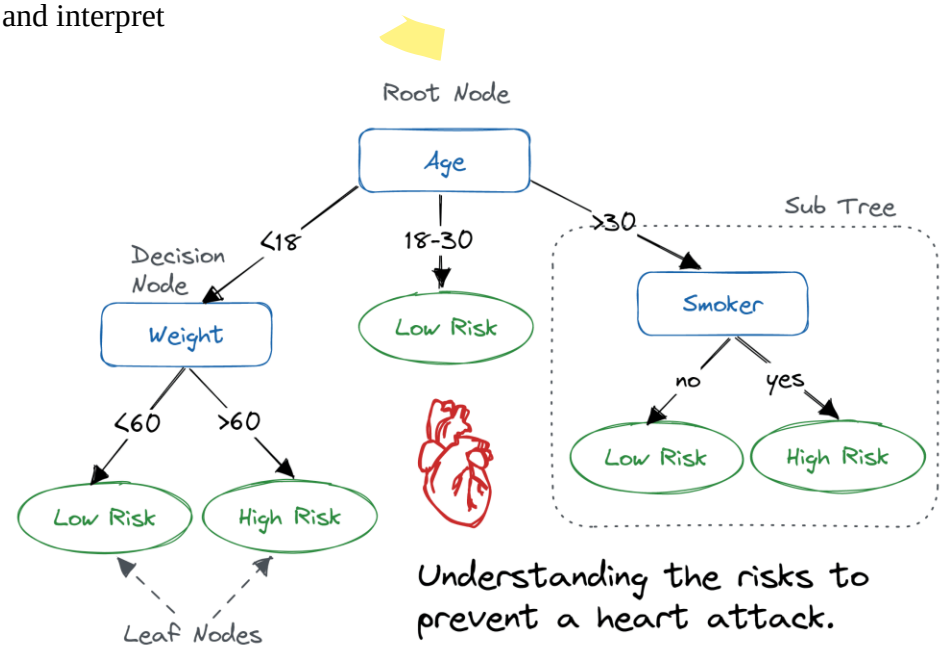
```
92.40121580547113
93.64161849710982
92.04545454545455
```

Experiment-6: Build a decision tree classifier compare its performance with ensemble techniques like random forest, bagging, boosting and voting. Demonstrate it with different decision trees.

Description:

A decision tree is a flowchart-like tree structure where an internal node represents a feature(or attribute), the branch represents a decision rule, and each leaf node represents the outcome.

The topmost node in a decision tree is known as the root node. It learns to partition on the basis of the attribute value. It partitions the tree in a recursive manner called recursive partitioning. This flowchart-like structure helps you in decision-making. It's visualization like a flowchart diagram which easily mimics the human level thinking. That is why decision trees are easy to understand and interpret



Code:

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, BaggingClassifier
, GradientBoostingClassifier,AdaBoostClassifier,VotingClassifier
from sklearn import datasets # import inbuild datasets

from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, accuracy_score, precisio
n_score
from sklearn.model_selection import cross_val_score, cross_val_predict


iris = datasets.load_iris()
```

```
X = iris.data
y = iris.target


X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.
3)

dtc = DecisionTreeClassifier()
rfc = RandomForestClassifier(n_estimators=100)
bc = BaggingClassifier(estimator = rfc, n_estimators=100, bootstrap=Tr
ue, n_jobs=-1, random_state=42)
gbc = GradientBoostingClassifier(n_estimators=100)
abc = AdaBoostClassifier(n_estimators=100, learning_rate=0.03)
vot = VotingClassifier(estimators = [('rfc', rfc), ('gbc', gbc)], voti
ng = 'soft')


clfs = {
    'DT': dtc,
    'RF': rfc,
    'BGC': bc,
    'GBC': gbc,
    'ABC': abc,
    'VOTING': vot
}

def train_classifier(clf, X_train, X_test, y_train, y_test):
  clf.fit(X_train, y_train)
  y_pred = clf.predict(X_test)
  score = clf.score(X_train, y_train)
  accuracy = accuracy_score(y_test, y_pred)
  precision = precision_score(y_test, y_pred, average='micro')

  return accuracy, precision, score


accuracy_scores = []
precision_scores = []
scores = []

for name, clf in clfs.items():
  acc, pre, score = train_classifier(clf, X_train, X_test, y_train, y_
test)

  accuracy_scores.append(acc)
  precision_scores.append(pre)
```

```
  scores.append(score)


import pandas as pd
df = pd.DataFrame({'Algorithm': clfs.keys(),'Score': scores, 'Accuracy
': accuracy_scores, 'Precision': precision_scores})
```

	Algorithm	Score	Accuracy	Precision
0	DT	1.000000	1.000000	1.000000
1	RF	1.000000	1.000000	1.000000
2	BGC	0.971429	1.000000	1.000000
3	GBC	1.000000	1.000000	1.000000
4	ABC	0.904762	0.977778	0.977778
5	VOTING	1.000000	1.000000	1.000000

Experiment-7: implementation of gradient descent Algorithm using tensor flow.

Description:

Gradient descent is a popular optimization algorithm used in machine learning and deep learning to minimize the cost or loss function. It works by iteratively adjusting the parameters of a model in the direction of steepest descent of the loss function until convergence or a specified number of iterations is reached. TensorFlow is a popular open-source machine learning library that provides various functions and tools for implementing gradient descent algorithms.

Overall, gradient descent is a powerful optimization algorithm that is widely used in machine learning and deep learning. TensorFlow provides a convenient and efficient way to implement gradient descent and tune its hyperparameters for optimal performance.

Code:

```
import tensorflow as tf
import numpy as np

x_train = np.random.rand(100, 1) * 10
y_train = 2 * x_train - 3 + np.random.randn(100, 1)

model = tf.keras.Sequential([
    tf.keras.layers.Dense(units=1, input_shape=[1])])
loss_fn = tf.keras.losses.MeanSquaredError()
optimizer = tf.keras.optimizers.SGD(learning_rate=0.01)
train_dataset = tf.data.Dataset.from_tensor_slices((x_train, y_train))
.batch(32)

for epoch in range(1000):
    for x_batch, y_batch in train_dataset:
        with tf.GradientTape() as tape:
            y_pred = model(x_batch)
            loss = loss_fn(y_batch, y_pred)
        gradients = tape.gradient(loss, model.trainable_variables)
        optimizer.apply_gradients(zip(gradients, model.trainable_varia
bles))
    if epoch % 100 == 0:
        print("Epoch {}: w = {}, b = {}".format(epoch, model.get_weigh
ts()[0], model.get_weights()[1]))
```

Output:

```
Epoch 0: w = [[1.4046725]], b = [0.36599326]
Epoch 100: w = [[1.9614199]], b = [-2.6434197]
Epoch 200: w = [[2.0052927]], b = [-2.9071152]
Epoch 300: w = [[2.0091255]], b = [-2.9301534]
Epoch 400: w = [[2.0094607]], b = [-2.9321659]
Epoch 500: w = [[2.0094898]], b = [-2.9323428]
Epoch 600: w = [[2.0094917]], b = [-2.9323537]
Epoch 700: w = [[2.0094917]], b = [-2.9323537]
Epoch 800: w = [[2.0094917]], b = [-2.9323537]
Epoch 900: w = [[2.0094917]], b = [-2.9323537]
```

Experiment-8: Give a case study on supervised algorithms.

Description:

Supervised learning is a machine learning paradigm where an algorithm is trained on a labeled dataset to predict outputs for unseen data. In this case study, we will cover the following supervised learning algorithms.

1. **Linear Regression:** Linear regression is a simple but powerful algorithm used to model the relationship between a dependent variable and one or more independent variables. For example, predicting the price of a house based on its size and location.

2. **Logistic Regression:** Logistic regression is used to model the probability of a binary outcome based on one or more independent variables. For example, predicting whether a customer will purchase a product or not based on their age, income, and gender.

3. **Decision Tree:** Decision trees are used to classify or predict outcomes by splitting data into subsets based on the values of input features. For example, predicting whether a customer will churn based on their age, income, and tenure.

4. **Random Forest**: A random forest is an ensemble of decision trees that improve accuracy by aggregating the predictions of multiple models. For example, predicting the likelihood of default on a loan based on a variety of demographic and financial data.

5. **Support Vector Machine (SVM):** SVM is used for binary classification by finding the hyperplane that best separates the classes. For example, predicting whether a customer will purchase a product or not based on their browsing history and demographics.

6. **Naive Bayes**: Naive Bayes is used for classification and is based on Bayes' theorem. For example, predicting whether an email is spam or not based on its contents.

7. **K-Nearest Neighbors (KNN)**: KNN is used for classification by finding the k-nearest neighbors to a given data point and assigning the label that appears most frequently. For example, predicting whether a customer will buy a product based on their shopping history and browsing behavior**.**

Experiment-9: Demonstration of clustering algorithms - k means, agglomerative, dbscan to classify for standard datasets.

Description:

K-means: K-means is a popular clustering algorithm that partitions a dataset into k clusters, with k specified beforehand. The algorithm works by iteratively assigning data points to the nearest cluster centroid and updating the centroids based on the mean of the points in the cluster. K-means is sensitive to the initial choice of centroids, and its performance can depend on the choice of k. It is best suited for datasets where the clusters are spherical and of similar size.

DBSCAN: DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is a clustering algorithm that groups together points that are close to each other in a high-density region of the dataset. It works by defining a radius around each point, and grouping together points that are within that radius and have a certain minimum number of neighbors. Points that are not within the radius of any other point are considered outliers. DBSCAN is robust to noise and can handle clusters of arbitrary shape and size.

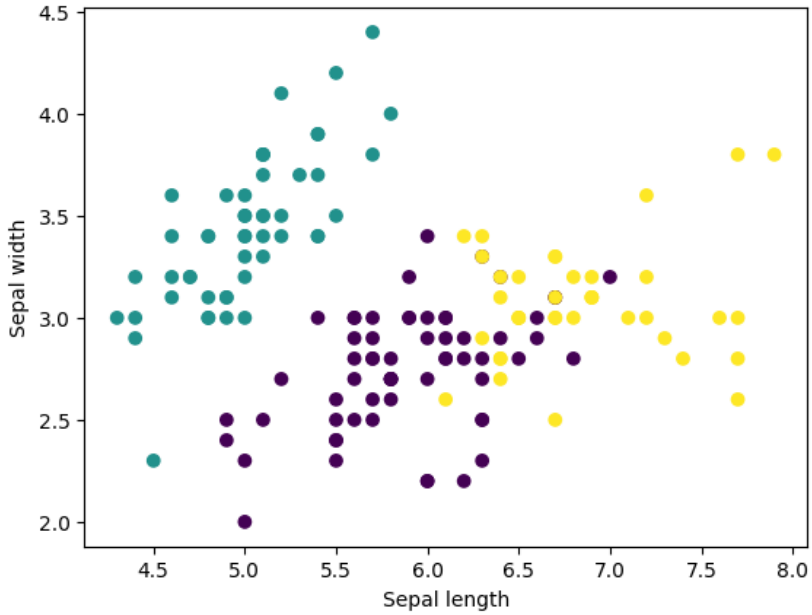
Agglomerative: Agglomerative clustering is a hierarchical clustering algorithm that starts by treating each point as its own cluster and iteratively merging the two closest clusters until a desired number of clusters is reached. The algorithm works by defining a distance metric between clusters (e.g., average linkage, complete linkage), and merging the two closest clusters based on that metric. Agglomerative clustering is versatile and can handle clusters of different shapes and sizes, but its performance can be slow for large datasets.

Code:

K means:

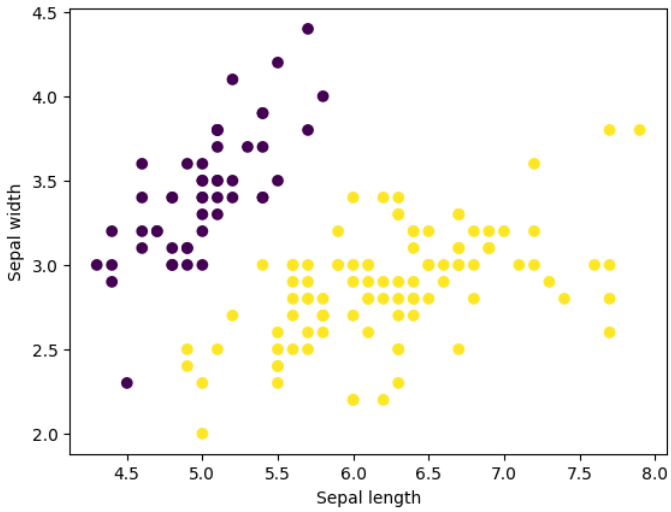
```
import pandas as pd
from sklearn.cluster import KMeans
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt

iris = load_iris()
X = pd.DataFrame(iris.data, columns=iris.feature_names)

# inertia = []
# for i in range(1, 11):
#     kmeans = KMeans(n_clusters=i, random_state=0)
#     kmeans.fit(X)
#     inertia.append(kmeans.inertia_)


# K-means clustering with 3 clusters
kmeans = KMeans(n_clusters=3, random_state=0)
```

```
kmeans.fit(X)
labels = kmeans.predict(X)

# Plotting the clusters
plt.scatter(X.iloc[:, 0], X.iloc[:, 1], c=labels)
plt.xlabel('Sepal length')
plt.ylabel('Sepal width')
plt.show()
```



DBSCAN:

```
import pandas as pd
from sklearn.cluster import DBSCAN
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt

iris = load_iris()
X = pd.DataFrame(iris.data, columns=iris.feature_names)

# DBSCAN clustering
dbscan = DBSCAN(eps=1, min_samples=3)
labels = dbscan.fit_predict(X)

# Plotting the clusters
plt.scatter(X.iloc[:, 0], X.iloc[:, 1], c=labels)
plt.xlabel('Sepal length')
plt.ylabel('Sepal width')
plt.show()
```



Agglomerative:

```
import pandas as pd
from sklearn.cluster import AgglomerativeClustering
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt

iris = load_iris()
X = pd.DataFrame(iris.data, columns=iris.feature_names)

# Agglomerative clustering with 3 clusters
agg_clustering = AgglomerativeClustering(n_clusters=3)
labels = agg_clustering.fit_predict(X)

# Plotting the clusters
plt.scatter(X.iloc[:, 0], X.iloc[:, 1], c=labels)
plt.xlabel('Sepal length')
plt.ylabel('Sepal width')
plt.show()
```