Use Sonar dataset from sklearn.datasets, which contains sonar signals for classifying objects as either "rock" or "mine."

In [1]:
```
from sklearn.datasets import fetch_openml
import numpy as np
import pandas as pd
sonar = fetch_openml(name="sonar", version=1)

X = sonar.data  # Features
y = sonar.target  # Target (rock or mine)
```

a) Begin by creating a training and testing datasest from the dataset, with a 80-20 ratio, and random_state=1. **1 pt**

In [2]:
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_stat
```

b) Train a KNN classifier on the training set to classify sonar signals as either "Rock" or "Mine." Use cross-validation to find an appropriate value of K. Evaluate and print the model's performance on the testing set using accuracy. **-- 9 points**

In [3]:
```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
from sklearn.model_selection import cross_val_score

# Define the range of k values
k_values = list(range(1, 21))

# Dictionary to store cross-validation accuracies for each k
cv_accuracies = {}

# Loop through different k values and perform cross-validation
for k in k_values:
    knn = KNeighborsClassifier(n_neighbors=k)
    cv_scores = cross_val_score(knn, X_train, y_train, cv=5)  # 5-fold cross-validat
    cv_accuracies[k] = cv_scores.mean()

# Find the best k based on cross-validation accuracy
best_k = max(cv_accuracies, key=cv_accuracies.get)
best_accuracy = cv_accuracies[best_k]

print("Cross-validation accuracies for different k values:")
for k, accuracy in cv_accuracies.items():
    print(f"K = {k}: Mean Accuracy = {accuracy:.4f}")

print(f"\nBest k value based on cross-validation: {best_k} with accuracy {best_accur

# Training the KNN model with the best k on the entire training set
best_knn = KNeighborsClassifier(n_neighbors=best_k)
best_knn.fit(X_train, y_train)

# Predicting on the test set using the best model
y_pred = best_knn.predict(X_test)

# Calculate accuracy on the test set
test_accuracy = accuracy_score(y_test, y_pred)
print(f"\nAccuracy on the test set using the best k ({best_k}): {test_accuracy:.4f}"
```

```
Cross-validation accuracies for different k values:
K = 1: Mean Accuracy = 0.8137
K = 2: Mean Accuracy = 0.7891
```

```
K = 3: Mean Accuracy = 0.8012
K = 4: Mean Accuracy = 0.7652
K = 5: Mean Accuracy = 0.7717
K = 6: Mean Accuracy = 0.7234
K = 7: Mean Accuracy = 0.7358
K = 8: Mean Accuracy = 0.6873
K = 9: Mean Accuracy = 0.6875
K = 10: Mean Accuracy = 0.6934
K = 11: Mean Accuracy = 0.6811
K = 12: Mean Accuracy = 0.6451
K = 13: Mean Accuracy = 0.6512
K = 14: Mean Accuracy = 0.6148
K = 15: Mean Accuracy = 0.6390
K = 16: Mean Accuracy = 0.6209
K = 17: Mean Accuracy = 0.6449
K = 18: Mean Accuracy = 0.6267
K = 19: Mean Accuracy = 0.6330
K = 20: Mean Accuracy = 0.6269

Best k value based on cross-validation: 1 with accuracy 0.8137

Accuracy on the test set using the best k (1): 0.7619
```

In [4]:
```
X_train
```

Out[4]:
```
array([[0.0139, 0.0222, 0.0089, ..., 0.0059, 0.0039, 0.0048],
       [0.0411, 0.0277, 0.0604, ..., 0.005 , 0.0085, 0.0044],
       [0.0731, 0.1249, 0.1665, ..., 0.0194, 0.0207, 0.0057],
       ...,
       [0.0208, 0.0186, 0.0131, ..., 0.0019, 0.0049, 0.0023],
       [0.0412, 0.1135, 0.0518, ..., 0.0225, 0.0098, 0.0085],
       [0.0333, 0.0221, 0.027 , ..., 0.0132, 0.0051, 0.0041]])
```

In [5]:
```
sonar
```

Out[5]:
```
{'data': array([[0.02  , 0.0371, 0.0428, ..., 0.0084, 0.009 , 0.0032],
        [0.0453, 0.0523, 0.0843, ..., 0.0049, 0.0052, 0.0044],
        [0.0262, 0.0582, 0.1099, ..., 0.0164, 0.0095, 0.0078],
        ...,
        [0.0522, 0.0437, 0.018 , ..., 0.0138, 0.0077, 0.0031],
        [0.0303, 0.0353, 0.049 , ..., 0.0079, 0.0036, 0.0048],
        [0.026 , 0.0363, 0.0136, ..., 0.0036, 0.0061, 0.0115]]),
 'target': array(['Rock', 'Rock', 'Rock', 'Rock', 'Rock', 'Rock', 'Rock', 'Rock',
        'Rock', 'Rock', 'Rock', 'Rock', 'Rock', 'Rock', 'Rock', 'Rock',
        'Rock', 'Rock', 'Rock', 'Rock', 'Rock', 'Rock', 'Rock', 'Rock',
        'Rock', 'Rock', 'Rock', 'Rock', 'Rock', 'Rock', 'Rock', 'Rock',
        'Rock', 'Rock', 'Rock', 'Rock', 'Rock', 'Rock', 'Rock', 'Rock',
        'Rock', 'Rock', 'Rock', 'Rock', 'Rock', 'Rock', 'Rock', 'Rock',
        'Rock', 'Rock', 'Rock', 'Rock', 'Rock', 'Rock', 'Rock', 'Rock',
        'Rock', 'Rock', 'Rock', 'Rock', 'Rock', 'Rock', 'Rock', 'Rock',
        'Rock', 'Rock', 'Rock', 'Rock', 'Rock', 'Rock', 'Rock', 'Rock',
        'Rock', 'Rock', 'Rock', 'Rock', 'Rock', 'Rock', 'Rock', 'Rock',
        'Rock', 'Rock', 'Rock', 'Rock', 'Rock', 'Rock', 'Rock', 'Rock',
        'Rock', 'Rock', 'Rock', 'Rock', 'Rock', 'Rock', 'Rock', 'Rock',
        'Rock', 'Mine', 'Mine', 'Mine', 'Mine', 'Mine', 'Mine', 'Mine',
        'Mine', 'Mine', 'Mine', 'Mine', 'Mine', 'Mine', 'Mine', 'Mine',
        'Mine', 'Mine', 'Mine', 'Mine', 'Mine', 'Mine', 'Mine', 'Mine',
        'Mine', 'Mine', 'Mine', 'Mine', 'Mine', 'Mine', 'Mine', 'Mine',
        'Mine', 'Mine', 'Mine', 'Mine', 'Mine', 'Mine', 'Mine', 'Mine',
        'Mine', 'Mine', 'Mine', 'Mine', 'Mine', 'Mine', 'Mine', 'Mine',
        'Mine', 'Mine', 'Mine', 'Mine', 'Mine', 'Mine', 'Mine', 'Mine',
        'Mine', 'Mine', 'Mine', 'Mine', 'Mine', 'Mine', 'Mine', 'Mine',
        'Mine', 'Mine', 'Mine', 'Mine', 'Mine', 'Mine', 'Mine', 'Mine',
        'Mine', 'Mine', 'Mine', 'Mine', 'Mine', 'Mine', 'Mine', 'Mine',
        'Mine', 'Mine', 'Mine', 'Mine', 'Mine', 'Mine', 'Mine', 'Mine',
        'Mine', 'Mine', 'Mine', 'Mine', 'Mine', 'Mine', 'Mine', 'Mine',
        'Mine', 'Mine', 'Mine', 'Mine', 'Mine', 'Mine', 'Mine', 'Mine',
        'Mine', 'Mine', 'Mine', 'Mine', 'Mine', 'Mine', 'Mine', 'Mine'],
       dtype=object),
```

```
 'frame': None,
 'categories': {},
 'feature_names': ['attribute_1',
  'attribute_2',
  'attribute_3',
  'attribute_4',
  'attribute_5',
  'attribute_6',
  'attribute_7',
  'attribute_8',
  'attribute_9',
  'attribute_10',
  'attribute_11',
  'attribute_12',
  'attribute_13',
  'attribute_14',
  'attribute_15',
  'attribute_16',
  'attribute_17',
  'attribute_18',
  'attribute_19',
  'attribute_20',
  'attribute_21',
  'attribute_22',
  'attribute_23',
  'attribute_24',
  'attribute_25',
  'attribute_26',
  'attribute_27',
  'attribute_28',
  'attribute_29',
  'attribute_30',
  'attribute_31',
  'attribute_32',
  'attribute_33',
  'attribute_34',
  'attribute_35',
  'attribute_36',
  'attribute_37',
  'attribute_38',
  'attribute_39',
  'attribute_40',
  'attribute_41',
  'attribute_42',
  'attribute_43',
  'attribute_44',
  'attribute_45',
  'attribute_46',
  'attribute_47',
  'attribute_48',
  'attribute_49',
  'attribute_50',
  'attribute_51',
  'attribute_52',
  'attribute_53',
  'attribute_54',
  'attribute_55',
  'attribute_56',
  'attribute_57',
  'attribute_58',
  'attribute_59',
  'attribute_60'],
 'target_names': ['Class'],
 'DESCR': '**Author**:   \n**Source**: Unknown -   \n**Please cite**:   \n\nNAME: So
nar, Mines vs. Rocks\n \n SUMMARY: This is the data set used by Gorman and Sejnowski
in their study\n of the classification of sonar signals using a neural network [1].
The\n task is to train a network to discriminate between sonar signals bounced\n off
a metal cylinder and those bounced off a roughly cylindrical rock.\n \n SOURCE: The
data set was contributed to the benchmark collection by Terry\n Sejnowski, now at th
```

e Salk Institute and the University of California at\n San Deigo. The data set was developed in collaboration with R. Paul\n Gorman of Allied-Signal Aerospace Technology Center.\n \n MAINTAINER: Scott E. Fahlman\n \n PROBLEM DESCRIPTION:\n \n The file "sonar.mines" contains 111 patterns obtained by bouncing sonar\n signals off a metal cylinder at various angles and under various\n conditions. The file "sonar.rocks" contains 97 patterns obtained from\n rocks under similar conditions. The transmitted sonar signal is a\n frequency-modulated chirp, rising in frequency. The data set contains\n signals obtained from a variety of different aspect angles, spanning 90\n degrees for the cylinder and 180 degrees for the rock.\n \n Each pattern is a set of 60 numbers in the range 0.0 to 1.0. Each number\n represents the energy within a particular frequency band, integrated over\n a certain period of time. The integration aperture for higher frequencies\n occur later in time, since these frequencies are transmitted later during\n the chirp.\n \n The label associated with each record contains the letter "R" if the object\n is a rock and "M" if it is a mine (metal cylinder). The numbers in the\n labels are in increasing order of aspect angle, but they do not encode the\n angle directly.\n \n METHODOLOGY: \n \n This data set can be used in a number of different ways to test learning\n speed, quality of ultimate learning, ability to generalize, or combinations\n of these factors.\n \n In [1], Gorman and Sejnowski report two series of experiments: an\n "aspect-angle independent" series, in which the whole data set is used\n without controlling for aspect angle, and an "aspect-angle dependent"\n series in which the training and testing sets were carefully controlled to\n ensure that each set contained cases from each aspect angle in\n appropriate proportions.\n \n For the aspect-angle independent experiments the combined set of 208 cases\n is divided randomly into 13 disjoint sets with 16 cases in each. For each\n experiment, 12 of these sets are used as training data, while the 13th is\n reserved for testing. The experiment is repeated 13 times so that every\n case appears once as part of a test set. The reported performance is an\n average over the entire set of 13 different test sets, each run 10 times.\n \n It was observed that this random division of the sample set led to rather\n uneven performance. A few of the splits gave poor results, presumably\n because the test set contains some samples from aspect angles that are\n under-represented in the corresponding training set. This motivated Gorman\n and Sejnowski to devise a different set of experiments in which an attempt\n was made to balance the training and test sets so that each would have a\n representative number of samples from all aspect angles. Since detailed\n aspect angle information was not present in the data base of samples, the\n 208 samples were first divided into clusters, using a 60-dimensional\n Euclidian metric; each of these clusters was then divided between the\n 104-member training set and the 104-member test set. \n \n The actual training and testing samples used for the "aspect angle\n dependent" experiments are marked in the data files. The reported\n performance is an average over 10 runs with this single division of the\n data set.\n \n A standard back-propagation network was used for all experiments. The\n network had 60 inputs and 2 output units, one indicating a cylinder and the\n other a rock. Experiments were run with no hidden units (direct\n connections from each input to each output) and with a single hidden layer\n with 2, 3, 6, 12, or 24 units. Each network was trained by 300 epochs over\n the entire training set.\n \n The weight-update formulas used in this study were slightly different from\n the standard form. A learning rate of 2.0 and momentum of 0.0 was used.\n Errors less than 0.2 were treated as zero. Initial weights were uniform\n random values in the range -0.3 to +0.3.\n \n RESULTS: \n \n For the angle independent experiments, Gorman and Sejnowski report the\n following results for networks with different numbers of hidden units:\n \n Hidden\t% Right on\tStd.\t% Right on\tStd.\n Units\tTraining set\tDev.\tTest Set\tDev.\n ------\t------------\t----\t----------\t----\n 0\t89.4\t\t2.1\t77.1\t\t8.3\n 2\t96.5\t\t0.7\t81.9\t\t6.2\n 3\t98.8\t\t0.4\t82.0\t\t7.3\n 6\t99.7\t\t0.2\t83.5\t\t5.6\n 12\t99.8\t\t0.1\t84.7\t\t5.7\n 24\t99.8\t\t0.1\t84.5\t\t5.7\n \n For the angle-dependent experiments Gorman and Sejnowski report the\n following results:\n \n Hidden\t% Right on\tStd.\t% Right on\tStd.\n Units\tTraining set\tDev.\tTest Set\tDev.\n ------\t------------\t----\t----------\t----\n 0\t79.3\t\t3.4\t73.1\t\t4.8\n 2\t96.2\t\t2.2\t85.7\t\t6.3\n 3\t98.1\t\t1.5\t87.6\t\t3.0\n 6\t99.4\t\t0.9\t89.3\t\t2.4\n 12\t99.8\t\t0.6\t90.4\t\t1.8\n 24   100.0\t\t0.0\t89.2\t\t1.4\n \n Not surprisingly, the network\'s performance on the test set was somewhat\n better when the aspect angles in the training and test sets were balanced.\n \n Gorman and Sejnowski further report that a nearest neighbor classifier on\n the same data gave an 82.7% probability of correct classification.\n \n Three trained human subjects were each tested on 100 signals, chosen at\n random from the set of 208 returns used to create this data set. Their\n responses ranged between 88% and 97% correct. However, they may have been\n using information from the raw sonar signal that is not preserved in the\n processed data sets presented here.\n \n REFERENCES: \n \n 1. Gorman, R. P., and Sejnowski, T. J. (1988). "Analysis of Hidden Units\n in a Layered Network Trained to Classify Sonar Targets" in Neural Networks,\n Vol. 1, pp. 75-89.\n\n\n\n\n

```
Relabeled values in attribute \'Class\'\n    From: R                       To: Rock
\n    From: M                       To: Mine\n\nDownloaded from openml.org.',
 'details': {'id': '40',
  'name': 'sonar',
  'version': '1',
  'description_version': '1',
  'format': 'ARFF',
  'creator': 'R. Paul Gorman',
  'collection_date': '1988-10-30',
  'upload_date': '2014-04-06T23:22:24',
  'language': 'English',
  'licence': 'Public',
  'url': 'https://api.openml.org/data/v1/download/40/sonar.arff',
  'parquet_url': 'http://openml1.win.tue.nl/dataset40/dataset_40.pq',
  'file_id': '40',
  'default_target_attribute': 'Class',
  'version_label': '1',
  'citation': 'https://archive.ics.uci.edu/ml/citation_policy.html',
  'tag': ['mythbusting_1',
   'study_1',
   'study_123',
   'study_15',
   'study_20',
   'study_29',
   'study_30',
   'study_41',
   'study_50',
   'study_52',
   'study_7',
   'study_88',
   'uci'],
  'visibility': 'public',
  'original_data_url': 'https://archive.ics.uci.edu/ml/datasets/Connectionist+Bench+
(Sonar,+Mines+vs.+Rocks)',
  'paper_url': 'https://www.sciencedirect.com/science/article/abs/pii/08936080889002
38',
  'minio_url': 'http://openml1.win.tue.nl/dataset40/dataset_40.pq',
  'status': 'active',
  'processing_date': '2020-11-20 19:08:05',
  'md5_checksum': '3ab630fbbfe25ab48b9bb47ce5759203'},
 'url': 'https://www.openml.org/d/40'}
```


In [6]:
```
X
```

Out[6]:
```
array([[0.02  , 0.0371, 0.0428, ..., 0.0084, 0.009 , 0.0032],
       [0.0453, 0.0523, 0.0843, ..., 0.0049, 0.0052, 0.0044],
       [0.0262, 0.0582, 0.1099, ..., 0.0164, 0.0095, 0.0078],
       ...,
       [0.0522, 0.0437, 0.018 , ..., 0.0138, 0.0077, 0.0031],
       [0.0303, 0.0353, 0.049 , ..., 0.0079, 0.0036, 0.0048],
       [0.026 , 0.0363, 0.0136, ..., 0.0036, 0.0061, 0.0115]])
```

c) Using any combination of the classification tools we've discussed in class:

- KNN
- Naive Bayes
- SVM
- Decision Tree (including Random Forests)
- Ensemble Methods (AdaBoost, Bagging)

You may also use feature extraction tools like PCA. Train and tune a model on the training set and evaluate its performance on the test set using accuracy. **-- 30 points**

- accuracy > .95 **-- 30 points**
- accuracy between 0.94 and 0.95 **-- 25 points**

- accuracy between 0.92 and 0.94 **-- 20 points**
- accuracy between 0.9 and 0.92 **-- 15 points**
- accuracy between 0.85 and 0.9 **-- 10 points**
- accuracy between 0.8 and 0.85 **-- 7 points**
- accuracy between 0.7 and 0.8 **-- 5 points**
- accuracy < 0.7 **-- 3 points**

In [7]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import GridSearchCV
import pandas as pd
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import VotingClassifier
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier, BaggingClas
import xgboost as xgboost
from sklearn.naive_bayes import MultinomialNB
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

from sklearn.decomposition import PCA
pca = PCA(n_components = 0.98)
X_train_pca = pca.fit_transform(X_train_scaled)
X_test_pca = pca.transform(X_test_scaled)
```

In [8]:
```
X_train_pca
```

Out[8]:
```
array([[ 0.08238329, -0.8415542 , -0.37129052, ...,  0.01706236,
         0.08270075, -0.02232914],
       [ 1.31114804,  0.04786938,  0.35541227, ..., -0.00252458,
        -0.12711949,  0.02057906],
       [ 0.18601859,  0.20724731,  0.68373819, ...,  0.04693734,
        -0.11561944,  0.17685122],
       ...,
       [ 0.03711126, -0.94687492, -0.41512858, ..., -0.15250896,
        -0.06377815,  0.00890625],
       [ 0.76279092,  0.54315745,  1.44873934, ..., -0.12771044,
         0.08777093,  0.04116873],
       [ 0.60750661, -1.05395462, -0.31667639, ..., -0.0421534 ,
        -0.06452961, -0.0213803 ]])
```

The one with the highest accuracy has been put bewlow this comment, which is an accuracy of 0.9524

In [9]:
```
# Training the KNN model with the best k on the entire training set
best_k = 1
best_knn = KNeighborsClassifier(n_neighbors=best_k)
best_knn.fit(X_train_pca, y_train)

# Predicting on the test set using the best model
y_pred = best_knn.predict(X_test_pca)

# Calculate accuracy on the test set
test_accuracy = accuracy_score(y_test, y_pred)
print(f"\nAccuracy on the test set using the knn with k = {best_k}: {test_accuracy:.
```

```
Accuracy on the test set using the knn with k = 1: 0.9524
```

In [10]:
```
# Create a KNN base classifier
base_knn = KNeighborsClassifier(n_neighbors=5)

# Create Bagging classifier with KNN as base estimator
bagging_knn = BaggingClassifier(base_estimator=base_knn, n_estimators=100, random_st

# Train the Bagging model
bagging_knn.fit(X_train, y_train)

# Make predictions on the test set
y_pred = bagging_knn.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy of Bagging with KNN Classifier: {accuracy:.4f}")
```

```
Accuracy of Bagging with KNN Classifier: 0.6905
```

In [11]:
```
# Create a Multinomial Naive Bayes classifier
nb_classifier = MultinomialNB()

# Define the grid of parameters to search
param_grid = {'alpha': [0.1, 0.5, 1.0, 1.5, 2.0]}

# Perform GridSearchCV to tune the 'alpha' parameter
grid_search = GridSearchCV(nb_classifier, param_grid, cv=5, scoring='accuracy')
grid_search.fit(X_train, y_train)

# Get the best parameter value
best_alpha = grid_search.best_params_['alpha']

# Train the model using the best alpha value
best_nb_classifier = MultinomialNB(alpha=best_alpha)
best_nb_classifier.fit(X_train, y_train)

# Make predictions on the test set
y_pred = best_nb_classifier.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy of Tuned Naive Bayes Classifier: {accuracy:.4f}")
print(f"Best alpha value: {best_alpha}")
```

```
Accuracy of Tuned Naive Bayes Classifier: 0.6667
Best alpha value: 0.1
```

In [12]:
```
# Create a base classifier (e.g., Decision Tree)
base_classifier_nb = MultinomialNB()

# Create AdaBoost classifier with a base estimator and set random_state
adaboost_classifier = AdaBoostClassifier(base_estimator = base_classifier_nb, n_esti

# Train the AdaBoost model
adaboost_classifier.fit(X_train, y_train)

# Make predictions on the test set
y_pred = adaboost_classifier.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy of AdaBoost Classifier: {accuracy:.4f}")
```

```
Accuracy of AdaBoost Classifier: 0.7143
```

In [13]:
```
# Create Bagging classifier with nb as base estimator
bagging_nb = BaggingClassifier(base_estimator=base_classifier_nb, n_estimators=100,

# Train the Bagging model
bagging_nb.fit(X_train, y_train)

# Make predictions on the test set
y_pred = bagging_nb.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy of Bagging with KNN Classifier: {accuracy:.4f}")
```

```
Accuracy of Bagging with KNN Classifier: 0.6667
```

In [14]:
```
# Create an SVM classifier
svm_classifier = SVC()

# Define the grid of parameters to search
param_grid = {
    'C': [0.1, 1, 10, 100],  # Penalty parameter C
    'gamma': [1, 0.1, 0.01, 0.001],  # Kernel coefficient for 'rbf'
    'kernel': ['rbf', 'poly', 'sigmoid']  # Kernel type
}

# Perform GridSearchCV to tune hyperparameters
grid_search = GridSearchCV(svm_classifier, param_grid, cv=5, scoring='accuracy')
grid_search.fit(X_train, y_train)

# Get the best parameter values
best_C = grid_search.best_params_['C']
best_gamma = grid_search.best_params_['gamma']
best_kernel = grid_search.best_params_['kernel']
# Train the model using the best parameters
best_svm_classifier = SVC(C=best_C, gamma=best_gamma, kernel = 'poly')
best_svm_classifier.fit(X_train, y_train)

# Make predictions on the test set
y_pred = best_svm_classifier.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy of Tuned SVM Classifier: {accuracy:.4f}")
print(f"Best C value: {best_C}")
print(f"Best gamma value: {best_gamma}")
```

```
Accuracy of Tuned SVM Classifier: 0.8095
Best C value: 10
Best gamma value: 0.1
```

In [15]:
```
# Create an SVM classifier
svm_classifier = SVC()

# Define the grid of parameters to search
param_grid = {
    'C': [0.1, 1, 10, 100],  # Penalty parameter C
    'gamma': [1, 0.1, 0.01, 0.001],  # Kernel coefficient for 'rbf'
    'kernel': ['rbf', 'poly', 'sigmoid']  # Kernel type
}

# Perform GridSearchCV to tune hyperparameters
grid_search = GridSearchCV(svm_classifier, param_grid, cv=5, scoring='accuracy')
grid_search.fit(X_train_pca, y_train)
```

```
# Get the best parameter values
best_C = grid_search.best_params_['C']
best_gamma = grid_search.best_params_['gamma']
best_kernel = grid_search.best_params_['kernel']
# Train the model using the best parameters
best_svm_classifier = SVC(C=best_C, gamma=best_gamma, kernel = 'poly')
best_svm_classifier.fit(X_train_pca, y_train)

# Make predictions on the test set
y_pred = best_svm_classifier.predict(X_test_pca)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy of Tuned SVM Classifier: {accuracy:.4f}")
print(f"Best C value: {best_C}")
print(f"Best gamma value: {best_gamma}")
```

```
Accuracy of Tuned SVM Classifier: 0.5238
Best C value: 10
Best gamma value: 0.1
```

In [16]:

```
# Create a base classifier (e.g., Decision Tree)
base_classifier_svm = SVC(probability= True)

# Create AdaBoost classifier with a base estimator and set random_state
adaboost_classifier = AdaBoostClassifier(base_estimator = base_classifier_svm, n_est

# Train the AdaBoost model
adaboost_classifier.fit(X_train, y_train)

# Make predictions on the test set
y_pred = adaboost_classifier.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy of AdaBoost Classifier: {accuracy:.4f}")
```

```
Accuracy of AdaBoost Classifier: 0.7381
```

In [17]:

```
# Create a base classifier (e.g., Decision Tree)
base_classifier_svm = SVC(probability= True)

# Create AdaBoost classifier with a base estimator and set random_state
adaboost_classifier = AdaBoostClassifier(base_estimator = base_classifier_svm, n_est

# Train the AdaBoost model
adaboost_classifier.fit(X_train_pca, y_train)

# Make predictions on the test set
y_pred = adaboost_classifier.predict(X_test_pca)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy of AdaBoost Classifier: {accuracy:.4f}")
```

```
Accuracy of AdaBoost Classifier: 0.8571
```

In [18]:

```
# Create Bagging classifier with nb as base estimator
bagging_svm = BaggingClassifier(base_estimator=base_classifier_svm, n_estimators=100

# Train the Bagging model
bagging_svm.fit(X_train, y_train)

# Make predictions on the test set
```

```
y_pred = bagging_svm.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy of Bagging with KNN Classifier: {accuracy:.4f}")
```

```
Accuracy of Bagging with KNN Classifier: 0.8333
```

In [19]:
```
# Create Bagging classifier with nb as base estimator
bagging_svm = BaggingClassifier(base_estimator=base_classifier_svm, n_estimators=100

# Train the Bagging model
bagging_svm.fit(X_train_pca, y_train)

# Make predictions on the test set
y_pred = bagging_svm.predict(X_test_pca)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy of Bagging with KNN Classifier: {accuracy:.4f}")
```

```
Accuracy of Bagging with KNN Classifier: 0.8571
```

In [20]:
```
tree_classifier = DecisionTreeClassifier()

# Define the grid of parameters to search
param_grid = {
    'criterion': ['gini', 'entropy'],  # Function to measure the quality of a split
    'max_depth': [None, 5, 10, 15, 20, 25],  # Maximum depth of the tree
    'min_samples_split': [2, 5, 10]  # Minimum number of samples required to split a
}

# Perform GridSearchCV to tune hyperparameters
grid_search = GridSearchCV(tree_classifier, param_grid, cv=5, scoring='accuracy')
grid_search.fit(X_train, y_train)

# Get the best parameter values
best_criterion = grid_search.best_params_['criterion']
best_max_depth = grid_search.best_params_['max_depth']
best_min_samples_split = grid_search.best_params_['min_samples_split']

# Train the model using the best parameters
best_tree_classifier = DecisionTreeClassifier(criterion = best_criterion,
                                              max_depth=best_max_depth,
                                              min_samples_split = best_min_samples_s
best_tree_classifier.fit(X_train, y_train)

# Make predictions on the test set
y_pred = best_tree_classifier.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy of Tuned Decision Tree Classifier: {accuracy:.4f}")
print(f"Best criterion: {best_criterion}")
print(f"Best max_depth: {best_max_depth}")
print(f"Best min_samples_split: {best_min_samples_split}")
```

```
Accuracy of Tuned Decision Tree Classifier: 0.7143
Best criterion: entropy
Best max_depth: 15
Best min_samples_split: 5
```

In [21]:
```
tree_classifier = DecisionTreeClassifier()

# Define the grid of parameters to search
```

```
param_grid = {
    'criterion': ['gini', 'entropy'],  # Function to measure the quality of a split
    'max_depth': [None, 5, 10, 15, 20, 25],  # Maximum depth of the tree
    'min_samples_split': [2, 5, 10]  # Minimum number of samples required to split a
}

# Perform GridSearchCV to tune hyperparameters
grid_search = GridSearchCV(tree_classifier, param_grid, cv=5, scoring='accuracy')
grid_search.fit(X_train_pca, y_train)

# Get the best parameter values
best_criterion = grid_search.best_params_['criterion']
best_max_depth = grid_search.best_params_['max_depth']
best_min_samples_split = grid_search.best_params_['min_samples_split']

# Train the model using the best parameters
best_tree_classifier = DecisionTreeClassifier(criterion = best_criterion,
                                              max_depth=best_max_depth,
                                              min_samples_split = best_min_samples_s
best_tree_classifier.fit(X_train_pca, y_train)

# Make predictions on the test set
y_pred = best_tree_classifier.predict(X_test_pca)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy of Tuned Decision Tree Classifier: {accuracy:.4f}")
print(f"Best criterion: {best_criterion}")
print(f"Best max_depth: {best_max_depth}")
print(f"Best min_samples_split: {best_min_samples_split}")
```

```
Accuracy of Tuned Decision Tree Classifier: 0.5714
Best criterion: gini
Best max_depth: 10
Best min_samples_split: 5
```

In [22]:

```
# Create a base classifier (e.g., Decision Tree)
base_classifier_dc = DecisionTreeClassifier(max_depth=1)

# Create AdaBoost classifier with a base estimator and set random_state
adaboost_classifier = AdaBoostClassifier(base_estimator=base_classifier_dc, random_s

# Train the AdaBoost model
adaboost_classifier.fit(X_train, y_train)

# Make predictions on the test set
y_pred = adaboost_classifier.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy of AdaBoost Classifier: {accuracy:.4f}")
```

```
Accuracy of AdaBoost Classifier: 0.8571
```

In [23]:

```
# Create a base classifier (e.g., Decision Tree)
base_classifier_dc = DecisionTreeClassifier(max_depth=1)

# Create AdaBoost classifier with a base estimator and set random_state
adaboost_classifier = AdaBoostClassifier(base_estimator=base_classifier_dc, random_s

# Train the AdaBoost model
adaboost_classifier.fit(X_train_pca, y_train)

# Make predictions on the test set
y_pred = adaboost_classifier.predict(X_test_pca)
```

```
# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy of AdaBoost Classifier: {accuracy:.4f}")
```

```
Accuracy of AdaBoost Classifier: 0.6667
```

In [24]:
```
# Create Bagging classifier with nb as base estimator
bagging_dc = BaggingClassifier(base_estimator=base_classifier_dc, n_estimators=100,

# Train the Bagging model
bagging_dc.fit(X_train, y_train)

# Make predictions on the test set
y_pred = bagging_dc.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy of Bagging with bagging Classifier: {accuracy:.4f}")
```

```
Accuracy of Bagging with bagging Classifier: 0.7381
```

In [25]:
```
# Create Bagging classifier with nb as base estimator
bagging_dc = BaggingClassifier(base_estimator=base_classifier_dc, n_estimators=100,

# Train the Bagging model
bagging_dc.fit(X_train_pca, y_train)

# Make predictions on the test set
y_pred = bagging_dc.predict(X_test_pca)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy of Bagging with bGGING Classifier: {accuracy:.4f}")
```

```
Accuracy of Bagging with bGGING Classifier: 0.7619
```

In [26]:
```
# Define the grid of parameters to search

forest_classifier = RandomForestClassifier(random_state = 42)
param_grid = {
    'n_estimators': [100, 200, 300],  # Number of trees in the forest
    'criterion': ['gini', 'entropy'],  # Function to measure the quality of a split
    'max_depth': [None, 5, 10, 15],  # Maximum depth of the trees
    'min_samples_split': [2, 5, 10]  # Minimum number of samples required to split a
}

# Perform GridSearchCV to tune hyperparameters
grid_search = GridSearchCV(forest_classifier, param_grid, cv=5, scoring='accuracy')
grid_search.fit(X_train, y_train)

# Get the best parameter values
best_n_estimators = grid_search.best_params_['n_estimators']
best_criterion = grid_search.best_params_['criterion']
best_max_depth = grid_search.best_params_['max_depth']
best_min_samples_split = grid_search.best_params_['min_samples_split']

# Train the model using the best parameters
best_forest_classifier = RandomForestClassifier(n_estimators=best_n_estimators, crit
                                                max_depth=best_max_depth,
                                                min_samples_split=best_min_samples_s
best_forest_classifier.fit(X_train, y_train)

# Make predictions on the test set
```

```
y_pred = best_forest_classifier.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy of Tuned Random Forest Classifier: {accuracy:.4f}")
print(f"Best n_estimators: {best_n_estimators}")
print(f"Best criterion: {best_criterion}")
print(f"Best max_depth: {best_max_depth}")
print(f"Best min_samples_split: {best_min_samples_split}")
```

```
Accuracy of Tuned Random Forest Classifier: 0.7857
Best n_estimators: 100
Best criterion: entropy
Best max_depth: None
Best min_samples_split: 5
```

In [27]:

```
# Define the grid of parameters to search

forest_classifier = RandomForestClassifier(random_state = 42)
param_grid = {
    'n_estimators': [100, 200, 300],  # Number of trees in the forest
    'criterion': ['gini', 'entropy'],  # Function to measure the quality of a split
    'max_depth': [None, 5, 10, 15],  # Maximum depth of the trees
    'min_samples_split': [2, 5, 10]  # Minimum number of samples required to split a
}

# Perform GridSearchCV to tune hyperparameters
grid_search = GridSearchCV(forest_classifier, param_grid, cv=5, scoring='accuracy')
grid_search.fit(X_train_pca, y_train)

# Get the best parameter values
best_n_estimators = grid_search.best_params_['n_estimators']
best_criterion = grid_search.best_params_['criterion']
best_max_depth = grid_search.best_params_['max_depth']
best_min_samples_split = grid_search.best_params_['min_samples_split']

# Train the model using the best parameters
best_forest_classifier = RandomForestClassifier(n_estimators=best_n_estimators, crit
                                                max_depth=best_max_depth,
                                                min_samples_split=best_min_samples_s
best_forest_classifier.fit(X_train_pca, y_train)

# Make predictions on the test set
y_pred = best_forest_classifier.predict(X_test_pca)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy of Tuned Random Forest Classifier: {accuracy:.4f}")
print(f"Best n_estimators: {best_n_estimators}")
print(f"Best criterion: {best_criterion}")
print(f"Best max_depth: {best_max_depth}")
print(f"Best min_samples_split: {best_min_samples_split}")
```

```
Accuracy of Tuned Random Forest Classifier: 0.7381
Best n_estimators: 200
Best criterion: gini
Best max_depth: None
Best min_samples_split: 2
```

In [28]:

```
# Create a base classifier (e.g., Decision Tree)
base_classifier_rfm = RandomForestClassifier()

# Create AdaBoost classifier with a base estimator and set random_state
adaboost_classifier = AdaBoostClassifier(base_estimator=base_classifier_rfm, random_
```

```
# Train the AdaBoost model
adaboost_classifier.fit(X_train, y_train)

# Make predictions on the test set
y_pred = adaboost_classifier.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy of AdaBoost Classifier: {accuracy:.4f}")
```

```
Accuracy of AdaBoost Classifier: 0.8333
```

In [29]:
```
# Create a base classifier (e.g., Decision Tree)
base_classifier_rfm = RandomForestClassifier()

# Create AdaBoost classifier with a base estimator and set random_state
adaboost_classifier = AdaBoostClassifier(base_estimator=base_classifier_rfm, random_

# Train the AdaBoost model
adaboost_classifier.fit(X_train_pca, y_train)

# Make predictions on the test set
y_pred = adaboost_classifier.predict(X_test_pca)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy of AdaBoost Classifier: {accuracy:.4f}")
```

```
Accuracy of AdaBoost Classifier: 0.7857
```

In [30]:
```
bagging_rfm = BaggingClassifier(base_estimator = base_classifier_rfm, n_estimators=1

# Train the Bagging model
bagging_rfm.fit(X_train, y_train)

# Make predictions on the test set
y_pred = bagging_rfm.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy of Bagging with rfm Classifier: {accuracy:.4f}")
```

```
Accuracy of Bagging with rfm Classifier: 0.7381
```

In [31]:
```
bagging_rfm = BaggingClassifier(base_estimator = base_classifier_rfm, n_estimators=1

# Train the Bagging model
bagging_rfm.fit(X_train_pca, y_train)

# Make predictions on the test set
y_pred = bagging_rfm.predict(X_test_pca)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy of Bagging with rfm Classifier: {accuracy:.4f}")
```

```
Accuracy of Bagging with rfm Classifier: 0.7619
```