Ensemble Techniques in Machine Learning

Ensemble techniques in machine learning are methods that combine multiple models to improve the overall performance of predictive models. The idea is that by combining several models, especially those that are weak learners (models with slightly better accuracy than random chance), you can create a stronger predictive model. Ensembles help reduce the variance, bias, or improve the accuracy of machine learning models.

Types of Ensemble Techniques

There are primarily three types of ensemble methods:

1. **Bagging (Bootstrap Aggregating)**
2. **Boosting**
3. **Stacking**

Let’s go into detail about each type along with examples.

---

1. Bagging (Bootstrap Aggregating)

Objective: Reduce variance and avoid overfitting by creating multiple instances of the same model on different subsets of the data.

- **How it works:**
  - Bagging involves training multiple models (often of the same type) on different random subsets of the training data, generated through bootstrapping (sampling with replacement).
  - After training, the predictions of all the models are combined, typically by averaging for regression tasks or by majority voting for classification tasks.
- **Example:**
  - **Random Forest** is the most well-known example of the bagging technique. It is a collection of decision trees, where each tree is trained on a random subset of the data and features. The trees' predictions are aggregated by majority voting (classification) or averaging (regression).
- **Advantages:**
  - Reduces variance and prevents overfitting in models prone to it, like decision trees.
  - Works well when individual models have high variance.
- **Disadvantages:**
  - Can lead to loss of interpretability.
  - Computationally expensive due to the training of multiple models.

---

2. **Boosting**

Objective: Reduce bias by building models sequentially, where each subsequent model tries to correct the errors of the previous one.

- **How it works:**
  - Boosting algorithms train models sequentially. Each new model is trained to correct the errors made by the previous models, typically by giving more weight to the data points that were misclassified or poorly predicted.
  - The final prediction is a weighted combination of all models.
- **Types of Boosting:**
  - **AdaBoost (Adaptive Boosting):**
    - **Process:** Models are trained sequentially. The first model is trained on the original data, then weights are assigned to all data points. Misclassified data points are given more weight, and the next model focuses more on them. The process repeats, and the final prediction is a weighted sum of all models.
    - **Example:** An AdaBoost classifier with decision stumps (simple decision trees with one node) as weak learners. The decision stumps are combined into a final strong classifier.
  - **Gradient Boosting:**
    - **Process:** Models are trained sequentially. However, instead of adjusting weights, each new model is trained on the residual errors (differences between the actual and predicted values) of the previous model. It is a gradient descent approach to minimize a loss function.
    - **Example: XGBoost**, **LightGBM**, and **CatBoost** are popular gradient boosting algorithms that are highly optimized for speed and performance.
- **Advantages:**
  - Often results in state-of-the-art performance for many machine learning tasks.
  - Boosting works well for complex data and can handle both regression and classification tasks effectively.
- **Disadvantages:**
  - Prone to overfitting if not properly regularized.
  - Computationally expensive due to the sequential training of models.

---

3. **Stacking**

Objective: Improve predictive accuracy by combining different models (base models and meta-models) in a two-layered structure.

- **How it works:**
  - Stacking involves training multiple different models (referred to as base models) on the same dataset.
  - A meta-model (or second-level model) is trained to make the final predictions based on the predictions of the base models.
  - For example, you could have three different algorithms as base models (e.g., a decision tree, a logistic regression, and a neural network). The meta-model would then combine their predictions in some way (e.g., using linear regression, logistic regression, etc.) to make a final prediction.
- **Example:**
  - Suppose you have three base models: a Decision Tree, a Support Vector Machine (SVM), and a KNN classifier. You use their predictions as features to train a meta-model like a Logistic Regression, which makes the final prediction.
- **Advantages:**
  - Can provide better predictive performance than any single model.
  - Effective when base models are diverse (e.g., different algorithms or different subsets of the data).
- **Disadvantages:**
  - Complex to implement and interpret.
  - Can be computationally expensive and prone to overfitting if not carefully managed.

Example of Ensemble Technique: Random Forest (Bagging)

Random Forest uses the bagging approach to combine multiple decision trees. The process looks like this:

1. Take multiple bootstrap samples from the dataset (with replacement).
2. Train a decision tree on each sample.
3. At each split in the decision tree, a random subset of features is chosen, and the best feature is used for the split.
4. Combine the trees' predictions by majority voting for classification tasks or averaging for regression tasks.

Why Ensemble Techniques Work?

Ensemble methods leverage the power of combining different models (or the same model on different data subsets) to reduce the weaknesses of individual models. The models compensate for each other's weaknesses, leading to improved performance. It's like a team of experts providing different perspectives to solve a problem—where individual mistakes are balanced out by the overall wisdom of the group.

Applications of Ensemble Methods:

- **Fraud Detection:** Combining models to accurately identify fraudulent transactions.
- **Stock Market Prediction:** Using ensembles to combine predictions from multiple models for better forecasting.
- **Medical Diagnosis:** Creating robust models by combining weak learners to reduce error rates in critical medical predictions.
- **Recommendation Systems:** Using ensemble methods to combine different algorithms to improve recommendation quality.

Conclusion

Ensemble techniques are one of the most effective ways to improve the accuracy and robustness of machine learning models. They are widely used in both classification and regression tasks across various domains. By combining multiple models, ensemble methods help mitigate the limitations of individual models and provide more reliable and accurate predictions.

Ensemble techniques offer several advantages, but they also come with certain drawbacks. Below is a breakdown of the pros and cons of ensemble methods in machine learning.

Pros of Ensemble Techniques

1. **Improved Accuracy**:
    - By combining multiple models, ensemble methods often provide better predictive performance compared to individual models. This is because they capitalize on the strengths of each model while minimizing individual weaknesses.
2. **Reduced Overfitting (Variance)**:
    - Methods like bagging (e.g., Random Forest) help reduce overfitting by training multiple models on different subsets of the data, averaging the results to smooth out variance.
3. **Increased Robustness**:
    - Ensemble methods provide more reliable predictions by balancing out errors from individual models. For example, if one model misclassifies an instance, the other models may correct that mistake.
4. **Flexibility**:
    - Ensemble techniques can combine models of different types (e.g., decision trees, neural networks, SVMs), making them highly adaptable to different types of data and problems.

5. **Handling Complex Data**:
    - In many cases, ensemble methods can better handle complex datasets with a large number of features or high variability in the data (e.g., non-linear patterns, noisy data).
6. **Applicability in Many Scenarios**:
    - Ensemble methods can be used for both regression and classification problems. They are also highly applicable to real-world scenarios such as fraud detection, medical diagnosis, stock prediction, etc.
7. **Stability**:
    - By averaging out predictions, ensemble methods often provide more stable and consistent results over time, reducing the impact of outliers or noise in the data.

Cons of Ensemble Techniques

1. **Increased Computational Cost**:
    - Training multiple models simultaneously or sequentially (as in boosting) requires significant computational power and memory. Ensemble methods like Random Forest or Gradient Boosting may require more resources than individual models.
2. **Interpretability**:
    - While individual models like decision trees are often easy to interpret, ensembles of models (especially in boosting or stacking) are much harder to explain. This lack of transparency can be problematic in applications where model interpretability is important (e.g., healthcare).
3. **Longer Training Times**:
    - Because multiple models need to be trained, ensemble methods can take much longer to train compared to single models. Techniques like boosting, where models are trained sequentially, can especially suffer from long training times.
4. **Risk of Overfitting (in Boosting)**:
    - Boosting techniques are prone to overfitting if not properly regularized. By continuously focusing on correcting the errors of previous models, they can become overly complex and fit noise in the training data.
5. **Complexity in Implementation**:
    - Ensemble methods, especially stacking or custom ensembles, can be complex to implement. They often require careful tuning of hyperparameters and a deep understanding of the models being combined.
6. **Diminishing Returns**:

- After a certain point, adding more models to the ensemble may not significantly improve performance. There can be diminishing returns, where increasing complexity adds computational cost but no tangible improvement in accuracy.

7. **Requires More Data**:
   - In some cases, ensemble methods require more data to be effective. For example, bagging techniques like Random Forest work best when there's a large dataset to generate varied training subsets.

Summary

Pros:

- Better accuracy and generalization.
- Reduced overfitting (variance).
- Robustness to noise and outliers.
- Flexibility across models and problem types.
- Suitable for complex data.

Cons:

- High computational cost and longer training times.
- Harder to interpret.
- Risk of overfitting (in boosting).
- Complex to implement and maintain.
- Possible diminishing returns with more models.

In summary, ensemble methods are powerful tools in machine learning that improve model performance at the cost of increased complexity and computational demand. Choosing to use them depends on the specific problem, the size of the dataset, the need for interpretability, and available computational resources.

Bagging Implementation

Bagging (Bootstrap Aggregating) is an ensemble learning technique that aims to improve the accuracy and robustness of models, particularly decision trees. It does so by training multiple instances of a model on different subsets of the data (bootstrapped samples) and then averaging their predictions (for regression) or using majority voting (for classification).

Here, I will demonstrate how to implement Bagging using Python and Scikit-learn. This example uses the `BaggingClassifier` and `BaggingRegressor` provided by Scikit-learn.

Bagging Implementation for Classification

In this example, we will use the Bagging technique with decision trees to classify data.

```
# Import libraries
import numpy as np
import pandas as pd
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris
from sklearn.metrics import accuracy_score

# Load the iris dataset
iris = load_iris()
X = iris.data
y = iris.target

# Split the data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Initialize the Bagging Classifier with Decision Tree as the
base estimator
bagging_clf = BaggingClassifier(
    base_estimator=DecisionTreeClassifier(),
    n_estimators=50,  # Number of trees
    random_state=42,
    bootstrap=True  # Whether to use bootstrapped datasets
)

# Fit the model
bagging_clf.fit(X_train, y_train)

# Predict on the test set
y_pred = bagging_clf.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Bagging Classifier Accuracy: {accuracy:.2f}")
```

Bagging Implementation for Regression

In this example, we will use Bagging with decision trees for regression purposes.

```
# Import libraries
from sklearn.ensemble import BaggingRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn.datasets import load_boston  # Boston dataset
deprecated, use fetch_california_housing or another dataset
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

# Load the dataset (use California Housing or your dataset)
from sklearn.datasets import fetch_california_housing
data = fetch_california_housing()
```

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

# Split the data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Initialize the Bagging Regressor with Decision Tree as the base
estimator
bagging_reg = BaggingRegressor(
    base_estimator=DecisionTreeRegressor(),
    n_estimators=50,  # Number of trees
    random_state=42,
    bootstrap=True  # Whether to use bootstrapped datasets
)

# Fit the model
bagging_reg.fit(X_train, y_train)

# Predict on the test set
y_pred = bagging_reg.predict(X_test)

# Calculate Mean Squared Error
mse = mean_squared_error(y_test, y_pred)
print(f"Bagging Regressor Mean Squared Error: {mse:.2f}")
```

Explanation

- **Base Estimator**: The individual model used within the bagging ensemble, which is a decision tree in this case. You can replace it with any other model (like `SVC`, `LogisticRegression`, etc.).
- **n_estimators**: The number of base estimators (e.g., decision trees) to be used.
- **bootstrap**: Whether to use bootstrapping (sampling with replacement). If `False`, all samples are used for training each base model.
- **random_state**: This ensures reproducibility.

How Bagging Works

1. **Bootstrapping**: Random subsets of the original training data are created, with replacement, for training individual models. This means that some data points will appear multiple times in the training set, while others may be left out.
2. **Training**: Multiple models (here, decision trees) are trained on different bootstrapped datasets.
3. **Aggregation**: For classification, the final prediction is made by majority voting. For regression, the final prediction is the average of predictions from individual models.

Pros of Bagging

- **Reduces Overfitting**: By averaging predictions from many models, bagging smooths out predictions and reduces overfitting.
- **Improved Accuracy**: By combining multiple models, bagging can achieve higher accuracy than a single model, particularly for unstable models like decision trees.

Cons of Bagging

- **Increased Complexity**: Training and maintaining multiple models requires more computational resources.
- **Interpretability**: The ensemble model can be harder to interpret than individual models, particularly when using decision trees which are otherwise easy to understand.

This approach is widely used in classification and regression problems to improve model performance and robustness.

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.ensemble import BaggingClassifier
from sklearn.neighbors import KNeighborsClassifier
```

In [2]:
```
from sklearn.datasets import load_breast_cancer
data = load_breast_cancer()
```

In [3]:
```
data
```

Out[3]:
```
{'data': array([[1.799e+01, 1.038e+01, 1.228e+02, ..., 2.654e-01, 4.601e
-01,
         1.189e-01],
        [2.057e+01, 1.777e+01, 1.329e+02, ..., 1.860e-01, 2.750e-01,
         8.902e-02],
        [1.969e+01, 2.125e+01, 1.300e+02, ..., 2.430e-01, 3.613e-01,
         8.758e-02],
        ...,
        [1.660e+01, 2.808e+01, 1.083e+02, ..., 1.418e-01, 2.218e-01,
         7.820e-02],
        [2.060e+01, 2.933e+01, 1.401e+02, ..., 2.650e-01, 4.087e-01,
         1.240e-01],
        [7.760e+00, 2.454e+01, 4.792e+01, ..., 0.000e+00, 2.871e-01,
         7.039e-02]]),
 'target': array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 1, 1, 1,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
0,
        0, 0, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0,
0,
        1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0,
0,
        1, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0,
1,
        1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1,
0,
        0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1,
1,
        1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1,
1,
        1, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0,
0,
        0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0,
0,
        1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 1,
1,
        1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0,
        0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1,
1,
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1,
1,
        1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0,
0,
        0, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1,
0,
        0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0,
0,
        1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1,
1,
        1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1,
0,
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1,
1,
        1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0,
0,
        1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1,
1,
        1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1,
1,
```

```
       1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1,
1,
       1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1]),
 'frame': None,
 'target_names': array(['malignant', 'benign'], dtype='<U9'),
 'DESCR': '.. _breast_cancer_dataset:\n\nBreast cancer wisconsin (diagno
stic) dataset\n--------------------------------------------\n\n**Data Se
t Characteristics:**\n\n:Number of Instances: 569\n\n:Number of Attribut
es: 30 numeric, predictive attributes and the class\n\n:Attribute Inform
ation:\n    - radius (mean of distances from center to points on the per
imeter)\n    - texture (standard deviation of gray-scale values)\n    -
perimeter\n    - area\n    - smoothness (local variation in radius lengt
hs)\n    - compactness (perimeter^2 / area - 1.0)\n    - concavity (seve
rity of concave portions of the contour)\n    - concave points (number o
f concave portions of the contour)\n    - symmetry\n    - fractal dimens
ion ("coastline approximation" - 1)\n\n    The mean, standard error, and
"worst" or largest (mean of the three\n    worst/largest values) of thes
e features were computed for each image,\n    resulting in 30 features.
For instance, field 0 is Mean Radius, field\n    10 is Radius SE, field
20 is Worst Radius.\n\n    - class:\n            - WDBC-Malignant\n
- WDBC-Benign\n\n:Summary Statistics:\n\n===============================
====== ====== ======\n                                        Min    Max
\n===================================== ====== ======\nradius (mean):
6.981  28.11\ntexture (mean):                        9.71   39.28\nperime
ter (mean):                     43.79  188.5\narea (mean):
143.5  2501.0\nsmoothness (mean):                    0.053  0.163\ncompa
ctness (mean):                   0.019  0.345\nconcavity (mean):
0.0    0.427\nconcave points (mean):                0.0    0.201\nsymmet
ry (mean):                      0.106  0.304\nfractal dimension (mean):
0.05   0.097\nradius (standard error):              0.112  2.873\ntextur
e (standard error):             0.36   4.885\nperimeter (standard erro
r):           0.757  21.98\narea (standard error):                6.802
542.2\nsmoothness (standard error):          0.002  0.031\ncompactness
(standard error):         0.002  0.135\nconcavity (standard error):
0.0    0.396\nconcave points (standard error):      0.0    0.053\nsymmet
ry (standard error):            0.008  0.079\nfractal dimension (standar
d error):   0.001  0.03\nradius (worst):                       7.93   3
6.04\ntexture (worst):                       12.02  49.54\nperimeter (wor
st):                     50.41  251.2\narea (worst):
185.2  4254.0\nsmoothness (worst):                    0.071  0.223\ncompa
ctness (worst):                   0.027  1.058\nconcavity (worst):
0.0    1.252\nconcave points (worst):               0.0    0.291\nsymmet
ry (worst):                     0.156  0.664\nfractal dimension (worst):
0.055  0.208\n===================================== ====== ======\n\n:Mi
ssing Attribute Values: None\n\n:Class Distribution: 212 - Malignant, 35
7 - Benign\n\n:Creator:  Dr. William H. Wolberg, W. Nick Street, Olvi L.
Mangasarian\n\n:Donor: Nick Street\n\n:Date: November, 1995\n\nThis is a
copy of UCI ML Breast Cancer Wisconsin (Diagnostic) datasets.\nhttps://g
oo.gl/U2Uwz2\n\nFeatures are computed from a digitized image of a fine n
eedle\naspirate (FNA) of a breast mass.  They describe\ncharacteristics
of the cell nuclei present in the image.\n\nSeparating plane described a
bove was obtained using\nMultisurface Method-Tree (MSM-T) [K. P. Bennet
t, "Decision Tree\nConstruction Via Linear Programming." Proceedings of
the 4th\nMidwest Artificial Intelligence and Cognitive Science Societ
y,\npp. 97-101, 1992], a classification method which uses linear\nprogra
mming to construct a decision tree.  Relevant features\nwere selected us
ing an exhaustive search in the space of 1-4\nfeatures and 1-3 separatin
g planes.\n\nThe actual linear program used to obtain the separating pla
```

```
ne\nin the 3-dimensional space is that described in:\n[K. P. Bennett and
O. L. Mangasarian: "Robust Linear\nProgramming Discrimination of Two Lin
early Inseparable Sets",\nOptimization Methods and Software 1, 1992, 23-
34].\n\nThis database is also available through the UW CS ftp server:\n
\nftp ftp.cs.wisc.edu\ncd math-prog/cpo-dataset/machine-learn/WDBC/\n
\n.. dropdown:: References\n\n  - W.N. Street, W.H. Wolberg and O.L. Man
gasarian. Nuclear feature extraction\n    for breast tumor diagnosis. IS
&T/SPIE 1993 International Symposium on\n    Electronic Imaging: Science
and Technology, volume 1905, pages 861-870,\n    San Jose, CA, 1993.\n
- O.L. Mangasarian, W.N. Street and W.H. Wolberg. Breast cancer diagnosi
s and\n    prognosis via linear programming. Operations Research, 43(4),
pages 570-577,\n    July-August 1995.\n  - W.H. Wolberg, W.N. Street, an
d O.L. Mangasarian. Machine learning techniques\n    to diagnose breast
cancer from fine-needle aspirates. Cancer Letters 77 (1994)\n    163-17
1.\n',
 'feature_names': array(['mean radius', 'mean texture', 'mean perimete
r', 'mean area',
        'mean smoothness', 'mean compactness', 'mean concavity',
        'mean concave points', 'mean symmetry', 'mean fractal dimensio
n',
        'radius error', 'texture error', 'perimeter error', 'area erro
r',
        'smoothness error', 'compactness error', 'concavity error',
        'concave points error', 'symmetry error',
        'fractal dimension error', 'worst radius', 'worst texture',
        'worst perimeter', 'worst area', 'worst smoothness',
        'worst compactness', 'worst concavity', 'worst concave points',
        'worst symmetry', 'worst fractal dimension'], dtype='<U23'),
 'filename': 'breast_cancer.csv',
 'data_module': 'sklearn.datasets.data'}
```

In [4]:
```
## Split the data into input and target feature
X = data.data
y = data.target
```

In [5]:
```
## Split the data into train and test set
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y)
```

In [7]:
```
## Create a model of KNeighoursClassifer
knn = KNeighborsClassifier()
knn.fit(X_train, y_train)
knn.score(X_test, y_test)
```

Out[7]:
```
0.916083916083916
```

In [8]:
```
y_pred = knn.predict(X_test)
y_pred
```

Out[8]:
```
array([0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0,
       1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0,
       1, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1,
       1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0,
       1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1,
       1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1,
       1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1])
```

In [9]:
```
## Create a model of BaggingClassifier
bag_knn = BaggingClassifier(knn, n_estimators=10)
```

```
bag_knn.fit(X_train, y_train)
bag_knn.score(X_test, y_test)
```

Out[9]:
```
0.916083916083916
```

In [10]:
```
y_pred_bag_knn = bag_knn.predict(X_test)
y_pred_bag_knn
```

Out[10]:
```
array([0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0,
       1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0,
       1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1,
       1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0,
       1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1,
       1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1,
       1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1])
```