WHAT IS BAGGING?

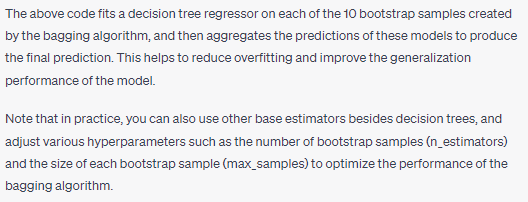
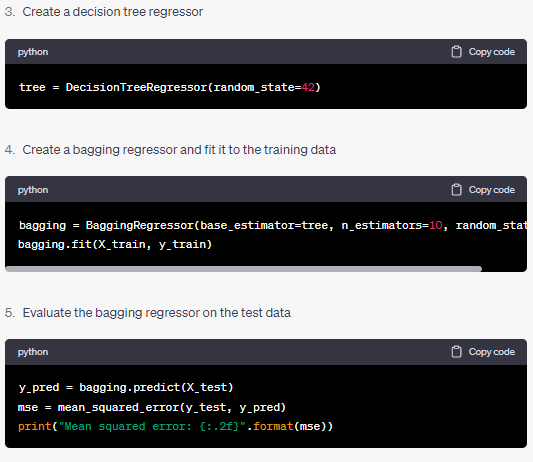
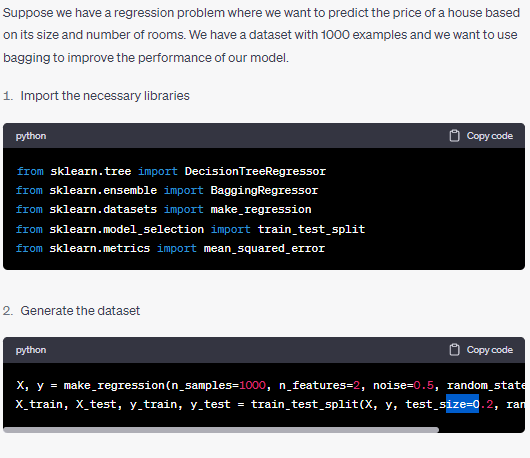
Bagging, short for Bootstrap Aggregating, is a popular ensemble learning technique in machine learning. The idea behind bagging is to reduce the variance of a machine learning model by combining the predictions of multiple models trained on different subsets of the training data.

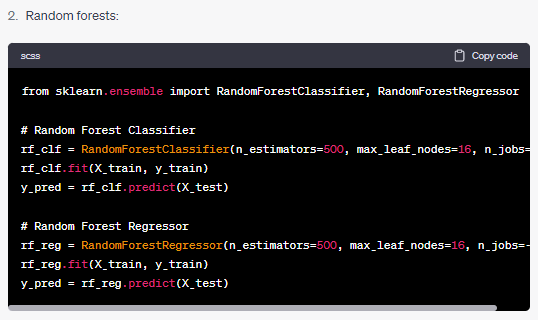
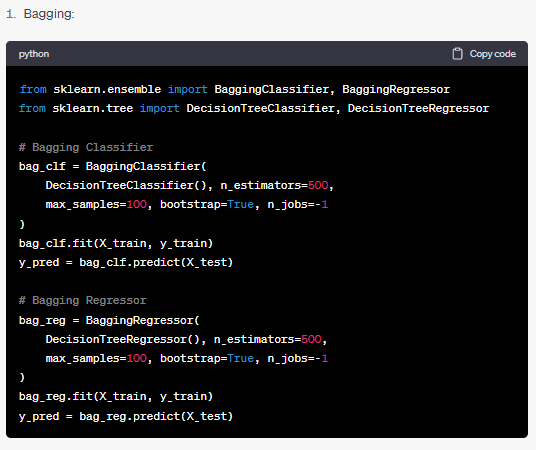
The basic idea of bagging is to create several different training sets by randomly sampling the original dataset with replacement, which is known as ***bootstrapping***. Each of these training sets is used to train a separate model, often using the same learning algorithm but with different random initialization or hyperparameters. Finally, the predictions of these models are combined by taking a simple average (in the case of regression) or by taking a majority vote (in the case of classification).

The advantage of bagging is that it reduces the variance of the model and prevents overfitting. This is because each model is trained on a different subset of the data, so each model is less likely to be biased towards a particular set of training examples. Furthermore, by combining the predictions of multiple models, bagging can often lead to better generalization performance and improved accuracy.

The most commonly used bagging algorithm is the Random Forest algorithm, which is a type of decision tree algorithm that uses bagging to improve performance. In Random Forest, the decision trees are trained on randomly sampled subsets of the features and training examples, and the final prediction is made by aggregating the predictions of all the trees.

You can create your own bagging algorithm using the steps below. In scikit-learn, you can implement bagging with decision trees using the **BaggingRegressor** or **BaggingClassifier** classes. These classes take a base estimator (in this case, a decision tree regressor or classifier) and use it to train multiple models on different random subsets of the training data.





3. Boosting

from sklearn.ensemble import AdaBoostClassifier, AdaBoostRegressor, GradientBoostingClassifier, GradientBoostingRegressor

**# AdaBoost Classifier**

ada\_clf = AdaBoostClassifier(

DecisionTreeClassifier(max\_depth=1), n\_estimators=200,

algorithm="SAMME.R", learning\_rate=0.5

)

ada\_clf.fit(X\_train, y\_train)

y\_pred = ada\_clf.predict(X\_test)

**# AdaBoost Regressor**

ada\_reg = AdaBoostRegressor(

DecisionTreeRegressor(max\_depth=1), n\_estimators=200,

learning\_rate=0.5

)

ada\_reg.fit(X\_train, y\_train)

y\_pred = ada\_reg.predict(X\_test)

**# Gradient Boosting Classifier**

gb\_clf = GradientBoostingClassifier(n\_estimators=100, learning\_rate=1.0, max\_depth=1, random\_state=0)

gb\_clf.fit(X\_train, y\_train)

y\_pred = gb\_clf.predict(X\_test)

**# Gradient Boosting Regressor**

gb\_reg = GradientBoostingRegressor(n\_estimators=100, learning\_rate=1.0, max\_depth=1, random\_state=0)

gb\_reg.fit(X\_train, y\_train)

y\_pred = gb\_reg.predict(X\_test)

4. Stacking

from sklearn.ensemble import StackingClassifier, StackingRegressor

from sklearn.linear\_model import LogisticRegression, RidgeCV

from sklearn.svm import SVC

from sklearn.tree import DecisionTreeClassifier, DecisionTreeRegressor

**# Stacking Classifier**

estimators = [

("svc", SVC()),

("dt", DecisionTreeClassifier()),

("lr", LogisticRegression())

]

clf = StackingClassifier(

estimators=estimators, final\_estimator=LogisticRegression()

)

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

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

**# Stacking Regressor**

estimators = [

("dt", DecisionTreeRegressor()),

("ridge", RidgeCV())

]

reg = StackingRegressor(

estimators=estimators, final\_estimator=RidgeCV()

)

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

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

5. Ensemble Pruning

from sklearn.ensemble import RandomForestClassifier

from sklearn.tree import DecisionTreeClassifier

from sklearn.datasets import make\_classification

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

from sklearn.metrics import accuracy\_score

from sklearn.model\_selection import cross\_val\_score

from sklearn.model\_selection import KFold

**# Generate a binary classification dataset**

X, y = make\_classification(n\_samples=1000, n\_features=10, n\_informative=5, random\_state=1)

**# Split the dataset into training and testing sets**

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

**# Train a Random Forest Classifier on the original dataset**

rf\_clf = RandomForestClassifier(n\_estimators=100, random\_state=1)

rf\_clf.fit(X\_train, y\_train)

**# Evaluate the Random Forest Classifier on the testing set**

y\_pred = rf\_clf.predict(X\_test)

print("Accuracy on original dataset:", accuracy\_score(y\_test, y\_pred))

**# Train a Decision Tree Classifier on the original dataset**

dt\_clf = DecisionTreeClassifier(random\_state=1)

dt\_clf.fit(X\_train, y\_train)

**# Evaluate the Decision Tree Classifier on the testing set**

y\_pred = dt\_clf.predict(X\_test)

print("Accuracy on original dataset:", accuracy\_score(y\_test, y\_pred))

**# Perform ensemble pruning by training a new Random Forest Classifier using the Decision Tree as a base estimator**

rf\_pruned = RandomForestClassifier(n\_estimators=100, base\_estimator=dt\_clf, random\_state=1)

rf\_pruned.fit(X\_train, y\_train)

**# Evaluate the pruned Random Forest Classifier on the testing set**

y\_pred = rf\_pruned.predict(X\_test)

print("Accuracy after ensemble pruning:", accuracy\_score(y\_test, y\_pred))

**# Compare the performance of the original Random Forest Classifier and the pruned one using cross-validation**

rf\_scores = cross\_val\_score(rf\_clf, X, y, cv=KFold(n\_splits=5, shuffle=True, random\_state=1))

print("Original Random Forest Classifier cross-validation scores:", rf\_scores.mean())

rf\_pruned\_scores = cross\_val\_score(rf\_pruned, X, y, cv=KFold(n\_splits=5, shuffle=True, random\_state=1))

print("Pruned Random Forest Classifier cross-validation scores:", rf\_pruned\_scores.mean())

FIXING AN UNBALANCED DATASET WITH SAMPLING TECHNIQUES

1. SMOTE (Synthetic Minority Over-Sampling Technique)

Imblearn.over\_sampling == SMOTE

1. OverSampling

Imblearn.over\_sampling == RandomOverSampler

1. UnderSampling

Imblearn.under\_sampling == RandomUnderSampler

**C O D E S**

**SMOTE**

from imblearn.over\_sampling import SMOTE

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

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import classification\_report

# assume X and y are the feature matrix and target variable, respectively

# split the dataset into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

# use SMOTE to balance the training data

sm = SMOTE(random\_state=42)

X\_train\_res, y\_train\_res = sm.fit\_resample(X\_train, y\_train)

# train a logistic regression model on the balanced training data

model = LogisticRegression()

model.fit(X\_train\_res, y\_train\_res)

# evaluate the model on the original testing data

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

print(classification\_report(y\_test, y\_pred))

**RANDOMOVERSAMPLER**

import pandas as pd

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

from imblearn.over\_sampling import RandomOverSampler

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import classification\_report

# Load the dataset

df = pd.read\_csv('my\_unbalanced\_data.csv')

# Split the dataset into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(df.drop('target\_variable', axis=1), df['target\_variable'], test\_size=0.3)

# Use the RandomOverSampler to oversample the minority class

oversampler = RandomOverSampler()

X\_train\_resampled, y\_train\_resampled = oversampler.fit\_resample(X\_train, y\_train)

# Train a logistic regression model on the resampled data

lr = LogisticRegression()

lr.fit(X\_train\_resampled, y\_train\_resampled)

# Test the model on the original test data

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

# Print a classification report

print(classification\_report(y\_test, y\_pred))

**RANDOMUNDERSAMPLER**

from imblearn.over\_sampling import RandomOverSampler

from imblearn.under\_sampling import RandomUnderSampler

# X\_train, y\_train are your training data and labels

# Instantiate the resampler

oversampler = RandomOverSampler(sampling\_strategy=0.5)

undersampler = RandomUnderSampler(sampling\_strategy=0.5)

# Resample the data

X\_resampled, y\_resampled = oversampler.fit\_resample(X\_train, y\_train)

X\_resampled, y\_resampled = undersampler.fit\_resample(X\_resampled, y\_resampled)