# **Bagging and Random Forest**

Ensemble models in machine learning combine the decisions from multiple models to improve the overall performance. The objective of this article is to introduce the concept of ensemble learning and understand algorithms like bagging and random forest which use a similar technique.

## What is Ensemble Learning?

Ensemble methods aim at improving the predictive performance of a given statistical learning or model ﬁtting technique. The general principle of ensemble methods is to construct a linear combination of some model ﬁtting method, instead of using a single ﬁt of the method.

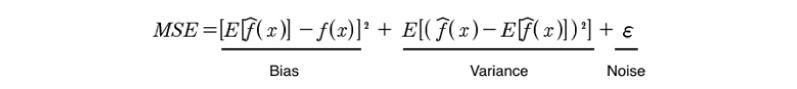
An ensemble is itself a supervised learning algorithm because it can be trained and then used to make predictions. Ensemble methods combine several decision trees classifiers to produce better predictive performance than a single decision tree classifier. The main principle behind the ensemble model is that a group of weak learners come together to form a strong learner, thus increasing the accuracy of the model. When we try to predict the target variable using any machine learning technique, the main causes of the difference in actual and predicted values are **noise, variance, and bias**. Ensemble helps to reduce these factors (except noise, which is irreducible error). The noise-related error is mainly due to noise in the training data and can't be removed. However, the errors due to bias and variance can be reduced.  
The total error can be expressed as follows:

**Total Error = Bias + Variance + Irreducible Error**

A measure such as **mean square error** (**MSE**) captures all of these errors for a continuous target variable and can be represented as follows:

Mean square error formula

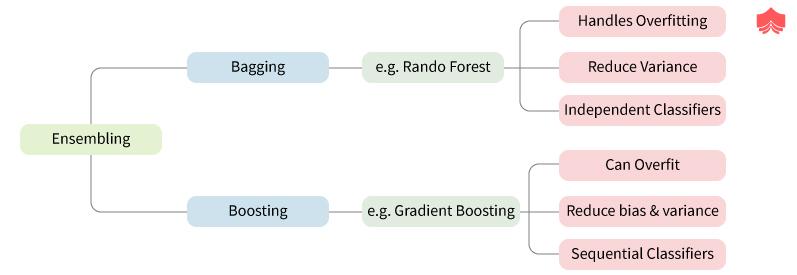
Where, E stands for the expected mean, Y represents the actual target values and fˆ(x) is the predicted values for the target variable. It can be broken down into its components such as bias, variance and noise as shown in the following formula:



Using techniques like Bagging and Boosting helps to decrease the variance and increase the robustness of the model. Combinations of multiple classifiers decrease variance, especially in the case of unstable classifiers, and may produce a more reliable classification than a single classifier.

### Ensemble Algorithm

The goal of ensemble algorithms is to combine the predictions of several base estimators built with a given learning algorithm in order to improve generalizability / robustness over a single estimator.



There are two families of ensemble methods which are usually distinguished:

1. **Averaging methods.**The driving principle is to build several estimators independently and then to average their predictions. On average, the combined estimator is usually better than any of the single base estimator because its variance is reduced.|  
   **Examples:** Bagging methods, Forests of randomized trees.
2. **Boosting methods.**Base estimators are built sequentially and one tries to reduce the bias of the combined estimator. The motivation is to combine several weak models to produce a powerful ensemble.  
   **Examples:** AdaBoost, Gradient Tree Boosting.

### Advantages of Ensemble Algorithm

* Ensemble is a proven method for improving the accuracy of the model and works in most of the cases.
* Ensemble makes the model more robust and stable thus ensuring decent performance on the test cases in most scenarios.
* You can use ensemble to capture linear and simple as well nonlinear complex relationships in the data. This can be done by using two different models and forming an ensemble of two.

### Disadvantages of Ensemble Algorithm

* Ensemble reduces the model interpret-ability and makes it very difficult to draw any crucial business insights at the end
* It is time-consuming and thus might not be the best idea for real-time applications
* The selection of models for creating an ensemble is an art which is really hard to master

## Basic Ensemble Techniques

* **Max Voting:** Max-voting is one of the simplest ways of combining predictions from multiple machine learning algorithms. Each base model makes a prediction and votes for each sample. The sample class with the highest votes is considered in the final predictive class. It is mainly used for classification problems.
* **Averaging:** Averaging can be used while estimating the probabilities in classification tasks. But it is usually used for regression problems. Predictions are extracted from multiple models and an average of the predictions are used to make the final prediction.
* **Weighted Average:** Like averaging, weighted averaging is also used for regression tasks. Alternatively, it can be used while estimating probabilities in classification problems. Base learners are assigned different weights, which represent the importance of each model in the prediction.

## Ensemble Methods

Ensemble methods became popular as a relatively simple device to improve the predictive performance of a base procedure. There are diﬀerent reasons for this: the bagging procedure turns out to be a variance reduction scheme, at least for some base procedures. On the other hand, boosting methods are primarily reducing the (model) bias of the base procedure. This already indicates that bagging and boosting are very diﬀerent ensemble methods. From the perspective of prediction, random forests is about as good as boosting, and often better than bagging.

**B**ootstrap **Agg**regation or **Bagging** tries to implement similar learners on small sample populations and then takes a mean of all the predictions.

* It combines Bootstrapping and Aggregation to form one ensemble model
* Reduces the variance error and helps to avoid overfitting

Bagging algorithms include:

* Bagging meta-estimator
* Random forest

**Boosting** refers to a family of algorithms which converts weak learner to strong learners. Boosting is a sequential process, where each subsequent model attempts to correct the errors of the previous model. Boosting is focused on reducing the bias. It makes the boosting algorithms prone to overfitting. To avoid overfitting, parameter tuning plays an important role in boosting algorithms. Some examples of boosting are mentioned below:

* AdaBoost
* GBM
* XGBM
* Light GBM
* CatBoost

## Why use ensemble models?

Ensemble models help in improving algorithm accuracy as well as the robustness of a model. Both Bagging and Boosting should be known by data scientists and machine learning engineers and especially people who are planning to attend data science/machine learning interviews.

Ensemble learning uses hundreds to thousands of models of the same algorithm and then work hand in hand to find the correct classification. You may also consider the [fable of the blind men and the elephant](https://en.wikipedia.org/wiki/Blind_men_and_an_elephant) to understand ensemble learning, where each blind man found a feature of the elephant and they all thought it was something different. However, if they would work together and discussed among themselves, they might have figured out what it is.

Using techniques like bagging and boosting leads to increased robustness of statistical models and decreased variance. Now the question becomes, between these different “B” words. Which is better?

## ****Which is better, Bagging or Boosting?****

There is no perfectly correct answer to that. It depends on the data, the simulation and the circumstances.

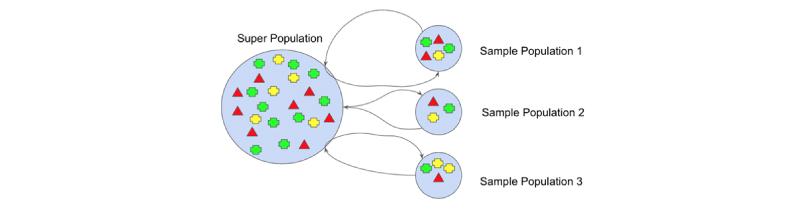
Bagging and boosting decrease the variance of your single estimate as they combine several estimates from different models. So the result may be a model with **higher stability**.

If the problem is that the single model gets a very low performance, Bagging will rarely get a **better bias**. However, Boosting could generate a combined model with lower errors as it optimizes the advantages and reduces pitfalls of the single model.

By contrast, if the difficulty of the single model is **overfitting**, then Bagging is the best option. Boosting for its part doesn’t help to avoid over-fitting; in fact, this technique is faced with this problem itself. For this reason, Bagging is effective more often than boosting. In this article we will discuss about Bagging, we will cover Boosting in the next post. But first, let us look into the very important concept of bootstrapping.

### Bootstrap Sampling

Sampling is the process of selecting a subset of observations from the population with the purpose of estimating some parameters about the whole population. Re[sampling methods](https://www.knowledgehut.com/blog/data-science/sampling-techniques), on the other hand, are used to improve the estimates of the population parameters.



In machine learning, the bootstrap method refers to random sampling with replacement. This sample is referred to as a resample. This allows the model or algorithm to get a better understanding of the various biases, variances and features that exist in the resample. Taking a sample of the data allows the resample to contain different characteristics then it might have contained as a whole. This is demonstrated in figure 1 where each sample population has different pieces, and none are identical. This would then affect the overall mean, [standard deviation](https://www.knowledgehut.com/blog/project-management/standard-deviation) and other descriptive metrics of a data set. In turn, it can develop more robust models.

Bootstrapping is also great for small size data sets that can have a tendency to overfit. In fact, we recommended this to one company that was concerned because their data sets were far from “Big Data”. Bootstrapping can be a solution in this case because algorithms that utilize bootstrapping can be more robust and handle new data sets depending on the methodology chosen (boosting or bagging).

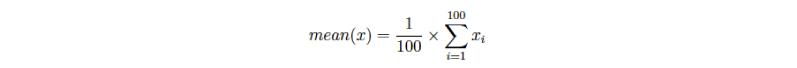
The reason behind using the bootstrap method is because it can test the stability of a solution. By using multiple sample data sets and then testing multiple models, it can increase robustness. Perhaps one sample data set has a larger mean than another, or a different standard deviation. This might break a model that was overfit, and not tested using data sets with different variations.

One of the many reasons bootstrapping has become very common is because of the increase in computing power. This allows for many times more permutations to be done with different resamples than previously. Bootstrapping is used in both Bagging and Boosting

Let us assume we have a sample of ‘n’ values (x) and we’d like to get an estimate of the mean of the sample.

mean(x) = 1/n \* sum(x)

Consider a sample of 100 values (x) and we’d like to get an estimate of the mean of the sample. We can calculate the mean directly from the sample as:



We know that our sample is small and that the mean has an error in it. We can improve the estimate of our mean using the bootstrap procedure:

1. Create many (e.g., 1000) random sub-samples of the data set with replacement (meaning we can select the same value multiple times).
2. Calculate the mean of each sub-*sample.*
3. Calculate the average of all of our collected means and use that as our estimated mean for the *data.*

E**xample:** Suppose we used 3 re-samples and got the mean values 2.3, 4.5 and 3.3. Taking the average of these we could take the estimated mean of the data to be 3.367. This process can be used to estimate other quantities like the [standard deviation](https://www.knowledgehut.com/blog/project-management/standard-deviation) and even quantities used in machine learning algorithms, like learned coefficients.

While using Python, we do not have to implement the bootstrap method manually. The scikit-learn library provides an implementation that creates a single bootstrap sample of a dataset.

The [resample () scikit-learn function](https://scikit-learn.org/stable/modules/generated/sklearn.utils.resample.html) can be used for sampling. It takes as arguments the data array, whether or not to sample with replacement, the size of the sample, and the seed for the pseudorandom number generator used prior to the sampling.

For example, let us create a bootstrap that creates a sample with replacement with 4 observations and uses a value of 1 for the pseudorandom number generator.

boot = resample (data, replace=True, n\_samples=4, random\_state=1)

As the bootstrap API does not allow to easily gather the out-of-bag observations that could be used as a test set to evaluate a fit model, in the univariate case we can gather the out-of-bag observations using a simple Python list comprehension.

# out of bag observations

oob = [x for x in data if x not in boot]

Let us look at a small example and execute it.

# scikit-learn bootstrap

from sklearn.utils import resample

# data sample

data = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6]

# prepare bootstrap sample

boot = resample(data, replace=True, n\_samples=4, random\_state=1)

print('Bootstrap Sample: %s' % boot)

# out of bag observations

oob = [x for x in data if x not in boot]

print('OOB Sample: %s' % oob)

The output will include the observations in the bootstrap sample and those observations in the out-of-bag sample.

Bootstrap Sample: [0.6, 0.4, 0.5, 0.1]

OOB Sample: [0.2, 0.3]

### **Bagging**

**Bootstrap Aggregation**, also known as **Bagging**, is a powerful ensemble method that was proposed by Leo Breiman in 1994 to prevent overfitting. The concept behind bagging is to combine the predictions of several base learners to create a more accurate output. Bagging is the application of the Bootstrap procedure to a high-variance machine learning algorithm, typically decision trees.

1. Suppose there are N observations and M features. A sample from observation is selected randomly with replacement (Bootstrapping).
2. A subset of features are selected to create a model with sample of observations and subset of features.
3. Feature from the subset is selected which gives the best split on the training data.
4. This is repeated to create many models and every model is trained in parallel
5. Prediction is given based on the aggregation of predictions from all the models.

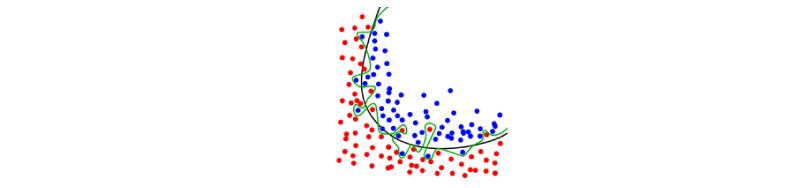
This approach can be used with machine learning algorithms that have a high variance, such as decision trees. A separate model is trained on each bootstrap sample of data and the average output of those models used to make predictions. This technique is called bootstrap aggregation or bagging for short.

Variance means that an algorithm’s performance is sensitive to the training data, with high variance suggesting that the more the training data is changed, the more the performance of the algorithm will vary.

The performance of high variance machine learning algorithms like unpruned decision trees can be improved by training many trees and taking the average of their predictions. Results are often better than a single decision tree.

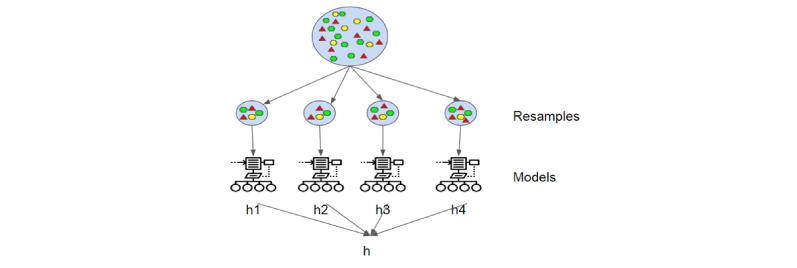
What Bagging does is help reduce variance from models that are might be very accurate, but only on the data they were trained on. This is also known as overfitting.

Overfitting is when a function fits the data too well. Typically, this is because the actual equation is much too complicated to take into account each data point and outlier.



Bagging gets around this by creating its own variance amongst the data by sampling and replacing data while it tests multiple hypothesis(models). In turn, this reduces the noise by utilizing multiple samples that would most likely be made up of data with various attributes(median, average, etc).

Once each model has developed a hypothesis. The models use voting for classification or averaging for regression. This is where the “Aggregating” in “Bootstrap Aggregating” comes into play. Each hypothesis has the same weight as all the others. When we later discuss boosting, this is one of the places the two methodologies differ.



Essentially, all these models run at the same time, and vote on which hypothesis is the most accurate.

This helps to decrease variance i.e. reduce the overfit.

### **Out-of-Bag Evaluation**

### With bagging, some instances may be sampled several times for any given predictor, while others may not be sampled at all. By default a BaggingClassifier samples m training instances with replacement (bootstrap=True), where m is the size of the training set. This means that only about 63% of the training instances are sampled on average for each predictor.6 The remaining 37% of the training instances that are not sampled are called out-of-bag (oob) instances. Note that they are not the same 37% for all predictors. Since a predictor never sees the oob instances during training, it can be evaluated on these instances, without the need for a separate validation set or cross-validation. You can evaluate the ensemble itself by averaging out the oob evaluations of each predic‐ tor. In Scikit-Learn, you can set oob\_score=True when creating a BaggingClassifier to request an automatic oob evaluation after training. The following code demonstrates this. The resulting evaluation score is available through the oob\_score\_ variable:

### >>> bag\_clf = BaggingClassifier(

### >>> DecisionTreeClassifier(), n\_estimators=500,

### >>> bootstrap=True, n\_jobs=-1, oob\_score=True)

>>> bag\_clf.fit(X\_train, y\_train)

>>> bag\_clf.oob\_score\_

0.93066666666666664

According to this oob evaluation, this BaggingClassifier is likely to achieve about 93.1% accuracy on the test set. Let’s verify this:

>>> from sklearn.metrics import accuracy\_score

>>> y\_pred = bag\_clf.predict(X\_test)

>>> accuracy\_score(y\_test, y\_pred)

0.93600000000000005

We get 93.6% accuracy on the test set—close enough! The oob decision function for each training instance is also available through the oob\_decision\_function\_ variable. In this case (since the base estimator has a pre dict\_proba() method) the decision function returns the class probabilities for each training instance. For example, the oob evaluation estimates that the second training instance has a 60.6% probability of belonging to the positive class (and 39.4% of belonging to the positive class):

>>> bag\_clf.oob\_decision\_function\_

array([[ 0. , 1. ],

[ 0.60588235, 0.39411765],

[ 1. , 0. ],

...

[ 1. , 0. ],

[ 0. , 1. ],

[ 0.48958333, 0.51041667]])

### Advantages

* Bagging takes advantage of ensemble learning wherein multiple weak learners outperform a single strong learner.
* It helps reduce variance and thus helps us avoid overfitting.

### Disadvantages

* There is a loss of interpretability of the model.
* There can possibly be a problem of high bias if not modeled properly.
* While bagging gives us more accuracy, it is computationally expensive and may not be desirable depending on the use case.

There are many bagging algorithms of which perhaps the most prominent would be Random Forest.

## Decision Trees

Decision trees are simple but intuitive models. Using a top-down approach, a root node creates binary splits unless a particular criterion is fulfilled. This binary splitting of nodes results in a predicted value on the basis of the interior nodes which lead to the terminal or the final nodes. For a classification problem, a decision tree will output a predicted target class for each terminal node produced.

### **Limitations to Decision Trees**

Decision trees tend to have a high variance when they utilize different training and test sets of the same data, since they tend to overfit on training data. This leads to poor performance when new and unseen data is added. This limits the usage of decision trees in predictive modeling. However, using ensemble methods, models that utilize decision trees can be created as a foundation for producing powerful results.

### Bootstrap Aggregating Trees

We have already discussed bootstrap aggregating (or bagging), we can create an ensemble (forest) of trees where multiple training sets are generated with replacement, meaning data instances. Once the training sets are created, a CART model can be trained on each subsample.

### **Features of Bagged Trees**

* Reduces variance by averaging the ensemble's results.
* The resulting model uses the entire feature space when considering node splits.
* Bagging trees allow the trees to grow without pruning, reducing the tree-depth sizes and resulting in high variance but lower bias, which can help improve predictive power.

### **Limitations to Bagging Trees**

The main limitation of bagging trees is that it uses the entire feature space when creating splits in the trees. Suppose some variables within the feature space are indicating certain predictions, there is a risk of having a forest of correlated trees, which actually increases bias and reduces variance.

### **Why a Forest is better than One Tree?**

The main objective of a machine learning model is to generalize properly to new and unseen data. When we have a flexible model, **overfitting** takes place. A flexible model is said to have high **variance**because the learned parameters (such as the structure of the decision tree) will vary with the training data.

On the other hand, an inflexible model is said to have high **bias** as it makes **assumptions** about the training data. An inflexible model may not have the capacity to fit even the training data and in both cases — high variance and high bias — the model is not able to generalize new and unseen data properly.

You can through the article on one of the foundational concepts in machine learning, [bias-variance tradeoff](https://www.knowledgehut.com/blog/data-science/bias-variance-tradeoff-in-machine-learning) which will help you understand that the balance between creating a model that is so flexible memorizes the training data and an inflexible model cannot learn the training data.

The main reason why the decision tree is prone to overfitting when we do not limit the maximum depth is because it has unlimited flexibility, which means it keeps growing unless there is one leaf node for every single observation.

Instead of limiting the depth of the tree which results in reduced variance and increase in bias, we can combine many decision trees into a single ensemble model known as the **random forest**.

## What is Random Forest algorithm?

Random forest is like bootstrapping algorithm with Decision tree (CART) model. Suppose we have 1000 observations in the complete population with 10 variables. Random forest will try to build multiple CART along with different samples and different initial variables. It will take a random sample of 100 observations and then chose 5 initial variables randomly to build a CART model. It will go on repeating the process say about 10 times and then make a final prediction on each of the observations. Final prediction is a function of each prediction. This final prediction can simply be the mean of each prediction.

The random forest is a model made up of many decision trees. Rather than just simply averaging the prediction of trees (which we could call a “forest”), this model uses two key concepts that gives it the name random:

1. Random sampling of training data points when building trees
2. Random subsets of features considered when splitting nodes

### How the Random Forest Algorithm Works

The basic steps involved in performing the random forest algorithm are mentioned below:

1. Pick N random records from the dataset.
2. Build a decision tree based on these N records.
3. Choose the number of trees you want in your algorithm and repeat steps 1 and 2.
4. In case of a regression problem, for a new record, each tree in the forest predicts a value for Y (output). The final value can be calculated by taking the average of all the values predicted by all the trees in the forest. Or, in the case of a classification problem, each tree in the forest predicts the category to which the new record belongs. Finally, the new record is assigned to the category that wins the majority vote.

### **Using Random Forest for Regression**

Here we have a problem where we have to predict the gas consumption (in millions of gallons) in 48 US states based on petrol tax (in cents), per capita income (dollars), paved highways (in miles) and the proportion of population with the driving license. We will use the random forest algorithm via the Scikit-Learn Python library to solve this regression problem.

First we import the necessary libraries and our dataset.

import pandas as pd

import numpy as np

dataset = pd.read\_csv('/content/petrol\_consumption.csv')

dataset.head()

|  | **Petrol\_tax** | **Average\_income** | **paved\_Highways** | **Population\_Driver\_licence(%)** | **Petrol\_Consumption** |
| --- | --- | --- | --- | --- | --- |
| **0** | **9.0** | **3571** | **1976** | **0.525** | **541** |
| **1** | **9.0** | **4092** | **1250** | **0.572** | **524** |
| **2** | **9.0** | **3865** | **1586** | **0.580** | **561** |
| **3** | **7.5** | **4870** | **2351** | **0.529** | **414** |
| **4** | **8.0** | **4399** | **431** | **0.544** | **410** |

You will notice that the values in our dataset are not very well scaled. Let us scale them down before training the algorithm.

### **Preparing Data For Training**

We will perform two tasks in order to prepare the data. Firstly, we will divide the data into ‘attributes’ and ‘label’ sets. The resultant will then be divided into training and test sets.

X = dataset.iloc[:, 0:4].values

y = dataset.iloc[:, 4].values

Now let us divide the data into training and testing sets:

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\_state=0)

### **Feature Scaling**

The dataset is not yet a scaled value as you will see that the Average\_Income field has values in the range of thousands while Petrol\_tax has values in the range of tens. It will be better if we scale our data. We will use Scikit-Learn's StandardScaler class to do the same.

# Feature Scaling

from sklearn.preprocessing import StandardScaler

sc = StandardScaler()

X\_train = sc.fit\_transform(X\_train)

X\_test = sc.transform(X\_test)

### **Training the Algorithm**

Now that we have scaled our dataset, let us train the random forest algorithm to solve this regression problem.

from sklearn.ensemble import Random Forest Regressor

regressor = Random Forest Regressor(n\_estimators=20,random\_state=0)

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

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

The RandomForestRegressor is used to solve regression problems via random forest. The most important parameter of the RandomForestRegressor class is the n\_estimators parameter. This parameter defines the number of trees in the random forest. Here we started with n\_estimator=20 and check the performance of the algorithm. You can find details for all of the parameters of RandomForestRegressor [here](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html).

### **Evaluating the Algorithm**

Let us evaluate the performance of the algorithm. For regression problems the metrics used to evaluate an algorithm are mean absolute error, mean squared error, and root mean squared error.

from sklearn import metrics

print('Mean Absolute Error:', metrics.mean\_absolute\_error(y\_test, y\_pred))

print('Mean Squared Error:', metrics.mean\_squared\_error(y\_test, y\_pred))

print('Root Mean Squared Error:',

np.sqrt(metrics.mean\_squared\_error(y\_test, y\_pred)))

Mean Absolute Error: 51.76500000000001

Mean Squared Error: 4216.166749999999

Root Mean Squared Error: 64.93201637097064

With 20 trees, the root mean squared error is 64.93 which is greater than 10 percent of the average petrol consumption i.e. 576.77. This may indicate, among other things, that we have not used enough estimators (trees).

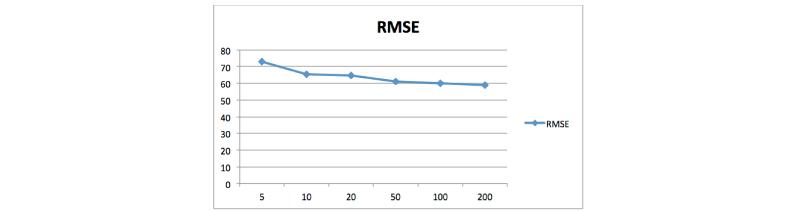
Let us now change the number of estimators to 200, the results are as follows:

Mean Absolute Error: 48.33899999999999

Mean Squared Error: 3494.2330150000003

Root Mean Squared Error: 59.112037818028234

The graph below shows the decrease in the value of the [root mean squared error](https://en.wikipedia.org/wiki/Root-mean-square_deviation) (RMSE) with respect to number of estimators.



You will notice that the error values decrease with the increase in the number of estimators. You may consider 200 a good number for n\_estimators as the rate of decrease in error diminishes. You may try playing around with other parameters to figure out a better result.

### **Using Random Forest for Classification**

Now let us consider a classification problem to predict whether a bank currency note is authentic or not based on four attributes i.e. variance of the image wavelet transformed image, skewness, entropy, andkurtosis of the image. We will use Random Forest Classifier to solve this binary classification problem. Let’s get started.

import pandas as pd

import numpy as np

dataset = pd.read\_csv('/content/bill\_authentication.csv')

dataset.head()

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Variance** | **Skewness** | **Kurtosis** | **Entropy** | **Class** |
| **0** | **3.62160** | **8.6661** | **-2.8073** | **-0.44699** | **0** |
| **1** | **4.54590** | **8.1674** | **-2.4586** | **-1.46210** | **0** |
| **2** | **3.86600** | **-2.6383** | **1.9242** | **0.10645** | **0** |
| **3** | **3.45660** | **9.5228** | **-4.0112** | **-3.59440** | **0** |
| **4** | **0.32924** | **-4.4552** | **4.5718** | **-0.98880** | **0** |

Similar to the data we used previously for the regression problem, this data is not scaled. Let us prepare the data for training.

**Preparing Data For Training**

The following code divides data into attributes and labels:

X = dataset.iloc[:, 0:4].values

y = dataset.iloc[:, 4].values

The following code divides data into training and testing sets:

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\_state=0)

**Feature Scaling**

We will do the same thing as we did for the previous problem.

# Feature Scaling

from sklearn.preprocessing import StandardScaler

sc = StandardScaler()

X\_train = sc.fit\_transform(X\_train)

X\_test = sc.transform(X\_test)

**Training the Algorithm**

Now that we have scaled our dataset, let us train the random forest algorithm to solve this classification problem.

from sklearn.ensemble import Random Forest Classifier

classifier = RandomForestClassifier(n\_estimators=20, random\_state=0)

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

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

For classification, we have used RandomForestClassifier class of the sklearn.ensemble library. It takes n\_estimators as a parameter. This parameter defines the number of trees in out random forest. Similar to the regression problem, we have started with 20 trees here. You can find details for all of the parameters of Random Forest Classifier here.

**Evaluating the Algorithm**

For evaluating classification problems,  the metrics used are accuracy, confusion matrix, precision recall, and F1 values

from sklearn.metrics import classification\_report, confusion\_matrix, accuracy\_score

print(confusion\_matrix(y\_test,y\_pred))

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

print(accuracy\_score(y\_test, y\_pred))

The output will look something like this:

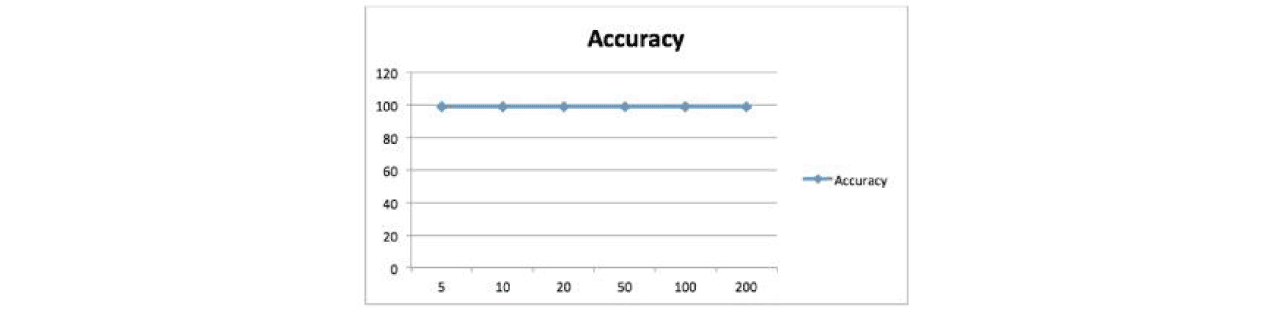
**Output:**

[ [ 155   2]

[     1  117] ]

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Precision** | **recall** | **f1-score** | **support** |
| **0** | **0.99** | **0.99** | **0.99** | **157** |
| **1** | **0.98** | **0.99** | **0.99** | **118** |
| **accuracy** |  |  | **0.99** | **275** |
| **macro avg** | **0.99** | **0.99** | **0.99** | **275** |
| **0.9890909090909091** | **0.99** | **0.99** | **0.99** | **275** |

Unlike the regression problem, changing the number of estimators for this problem did not make any difference in the results.



An accuracy of 98.9% is pretty good. In this case, we have seen that there is not much improvement if the number of trees are increased. You may try playing around with other parameters of the RandomForestClassifier class and see if you can improve on our results.

## Advantages and Disadvantages of using Random Forest

As with any algorithm, there are advantages and disadvantages to using it. Let us look into the pros and cons of using Random Forest for classification and regression.

**Advantages**

* Random forest algorithm is unbiased as there are multiple trees and each tree is trained on a subset of data.
* Random Forest algorithm is very stable. Introducing a new data in the dataset does not affect much as the new data impacts one tree and is pretty hard to impact all the trees.
* The random forest algorithm works well when you have both categorical and numerical features.
* With missing values in the dataset, the random forest algorithm performs very well.

**Disadvantages**

* A major disadvantage of random forests lies in their complexity. More computational resources are required and also results in the large number of decision trees joined together.
* Due to their complexity, training time is more compared to other algorithms.

## Summary

In this article, we have covered what ensemble learning is and discussed basic ensemble techniques. We also looked into bootstrap sampling involves iteratively resampling of a dataset with a replacement which allows the model or algorithm to get a better understanding of various features. Then we moved on to bagging followed by random forest. We also implemented random forest in Python for both regression and classification and came to the conclusion that increasing the number of trees or estimators does not always make a difference in a classification problem. However, in regression, there is an impact.

**Gift:** <https://machinelearningmastery.com/bagging-and-random-forest-for-imbalanced-classification/>

**Random Forest application to large dataset vs small dataset**

The Random Forest algorithm can be applied to both large and small datasets, but there are considerations and differences in how it might perform and be used in each scenario.

**Random Forest on Large Datasets:**

1. **Scalability:** Random Forest can handle large datasets reasonably well due to its ensemble nature. It builds multiple decision trees and combines their outputs, which can help in managing large amounts of data.
2. **Computation Time:** While Random Forest can be parallelized and distributed to some extent, training on very large datasets might still take a significant amount of time, especially if you're using a single machine.
3. **Sampling:** In the case of large datasets, Random Forest's ability to randomly sample data points (with replacement) for each tree can still lead to diverse trees and good generalization.
4. **Feature Importance:** Random Forest can handle a large number of features without much issue. However, interpreting feature importance might become more complex with a very high-dimensional dataset.
5. **Memory:** Training Random Forest on large datasets might require substantial memory, especially if the algorithm isn't optimized for memory efficiency.
6. **Overfitting:** Random Forests are less likely to overfit on larger datasets due to the averaging effect of multiple trees. However, if the dataset is extremely noisy, overfitting can still occur.

**Random Forest on Small Datasets:**

1. **Overfitting:** One of the advantages of Random Forest is its ability to mitigate overfitting, even on smaller datasets, by constructing multiple trees and combining their predictions.
2. **Computation Time:** Training a Random Forest on a small dataset is generally faster compared to a large dataset. However, for very small datasets, other algorithms like decision trees might also be effective and faster.
3. **Sampling:** When the dataset is small, the random sampling with replacement might lead to trees that are very similar to each other. This could potentially reduce the diversity and effectiveness of the ensemble.
4. **Feature Importance:** Random Forest can still provide insights into feature importance on small datasets, and these insights might be easier to interpret.
5. **Noise Impact:** With a small dataset, noisy data can have a larger impact on the model's performance and predictions.

In summary, Random Forest can be applied to both large and small datasets, but the considerations mentioned above should be kept in mind. For very large datasets, techniques like distributed computing and optimizing memory usage might be necessary. For very small datasets, other algorithms like decision trees, which are simpler and faster, might also be viable options.

**Is random forest algorithm sensitive to outlier?**

Yes, the Random Forest algorithm is to some extent sensitive to outliers, but its sensitivity is generally lower compared to some other algorithms like linear regression or K-means clustering. The ensemble nature of Random Forest, which combines the predictions of multiple decision trees, provides it with some inherent resistance to the effects of outliers.

Here's how Random Forest's sensitivity to outliers works:

1. **Robustness through Averaging:** The random sampling of data points with replacement during the construction of individual decision trees within the Random Forest helps reduce the impact of individual outliers. Outliers might affect the predictions of a single decision tree, but when their effects are averaged across multiple trees, their influence is diminished.
2. **Tree Depth:** The depth of the decision trees in a Random Forest can impact its sensitivity to outliers. Shallower trees are less likely to be affected by individual outliers as they won't be able to overfit to those outliers.
3. **Splitting Criterion:** Random Forest typically uses metrics like Gini impurity or entropy to decide how to split nodes in decision trees. These metrics are less sensitive to individual data points compared to, for example, mean squared error in linear regression.
4. **Ensemble Effect:** The final prediction of the Random Forest is the average (regression) or mode (classification) of predictions from all the individual trees. Outliers in some trees might not be present in others, so their effect on the final prediction is reduced.

However, it's important to note that while Random Forests are relatively robust to outliers, extreme outliers or a significant number of outliers could still impact the performance of the algorithm. It's always a good practice to preprocess and understand your data, and if outliers are a concern, you might consider techniques like:

* **Data Transformation:** Transforming the data using techniques like logarithmic or square root transformations can help reduce the impact of extreme values.
* **Outlier Detection:** Identify and handle outliers using techniques like z-score, IQR (interquartile range), or specialized outlier detection algorithms.
* **Data Trimming:** Truncate or remove extreme values before training the model, if appropriate and justifiable based on domain knowledge.

In summary, while Random Forests are generally less sensitive to outliers than some other algorithms, it's still important to be aware of the potential impact of outliers and to apply appropriate preprocessing techniques if necessary.

**Effect of missing values on random forest algorithm**

Missing values can have an impact on the Random Forest algorithm, just as they do on other machine learning algorithms. However, Random Forests have some built-in mechanisms that make them relatively robust to missing data compared to certain other algorithms. Here's how missing values can affect Random Forests:

**Advantages of Random Forests with Missing Values:**

1. **No Imputation Needed:** Random Forests can handle missing values without requiring imputation (filling in missing values with estimated values). The algorithm can use the available data for splitting nodes in the decision trees, and it doesn't rely on a global mean or imputed values.
2. **Nonparametric Nature:** Random Forests make decisions based on binary splits in the data, not assuming a specific distribution or functional form. This can help in cases where certain data points have missing values, as long as there's enough information in other features to guide the splits.
3. **Averaging Effect:** The ensemble nature of Random Forests, where multiple trees are combined, helps mitigate the impact of missing values. If one tree is influenced by missing values, the effect is likely to be balanced out by the other trees.

**Considerations and Mitigation Strategies:**

1. **Bias in Feature Importance:** If a feature has a large number of missing values, it might be downplayed in terms of feature importance. This could potentially lead to underestimating the true importance of the feature if it's important in predicting the target variable.
2. **Impact on Similarity Measures:** In distance-based similarity measures (if used for instance in proximity-based variants of Random Forests), missing values might affect the calculation of distances between data points.
3. **Imputing Missingness Information:** While Random Forests don't require imputation, it might still be beneficial to create a binary flag indicating whether a value is missing. This information could become relevant if the missingness itself has predictive power.
4. **Evaluation of Performance:** It's important to assess the performance of the Random Forest model with missing values. In some cases, missing values might introduce bias or reduce predictive accuracy.
5. **Randomness in Missingness:** If the missingness in your data is not random but related to the target variable, the Random Forest model might not completely mitigate the effects of this bias.

In summary, Random Forests offer some advantages in handling missing values compared to certain other algorithms, but it's still important to be cautious. The impact of missing values can vary depending on the nature of the data and the problem you're trying to solve. It's recommended to experiment with both including and excluding missing values, and assessing the impact on model performance, to make informed decisions about how to handle them.

**Effect of correlation on random forest algorithm**

Correlation among features (variables) can have both positive and negative effects on the performance of the Random Forest algorithm, just as it does for other machine learning algorithms. The effects depend on the nature and degree of correlation, as well as the specific dataset and problem you're dealing with. Here's how correlation can impact Random Forests:

**Positive Effects:**

1. **Reduction of Overfitting:** Correlated features can provide similar information to the model. Random Forests are less prone to overfitting when there's redundancy in the data, as the averaging across multiple trees helps mitigate the impact of noise introduced by correlated features.
2. **Stability of Importance Measures:** Correlated features might have similar feature importance scores. In Random Forests, if a feature is important, it's likely that multiple correlated features will contribute to the importance measure. This provides stability and can help in interpreting feature importance.

**Negative Effects:**

1. **Decreased Model Interpretability:** Highly correlated features can make it harder to interpret the model, as it might be challenging to determine which of the correlated features is truly contributing to the predictions.
2. **Redundancy:** If two or more features are highly correlated, the model might not gain much additional information from all of them. In such cases, removing one of the correlated features might not significantly impact model performance.
3. **Increased Complexity:** While Random Forests are generally robust to correlated features, highly correlated features might introduce additional complexity without necessarily improving model performance.

**Mitigation Strategies:**

1. **Feature Selection:** If you have highly correlated features, consider performing feature selection to choose the most relevant features and reduce redundancy.
2. **Feature Engineering:** Instead of using raw correlated features, you can create new features that capture the essence of the correlation, potentially improving model performance and interpretability.
3. **Partial Dependence Plots:** Use partial dependence plots to analyze the relationship between individual features and the target variable, which can help you understand the impact of correlated features.
4. **Experimentation:** Experiment with both including and excluding correlated features to see how they affect model performance. Cross-validation can help you assess which approach works better.

In summary, the effect of correlation on Random Forests can be both positive and negative. While Random Forests are generally robust to correlated features and can handle them reasonably well, it's still important to consider the context of your specific dataset and problem, and to apply appropriate strategies to mitigate any negative effects that correlated features might introduce.

**Feature Engineering, Feature Selection and Feature Importance in random forest algorithm**

Certainly! Let's delve into Feature Engineering, Feature Selection, and Feature Importance in the context of the Random Forest algorithm:

**Feature Engineering:** Feature engineering involves creating new features or transforming existing ones to improve the performance of a machine learning model. Here's how it applies to Random Forest:

1. **Creating New Features:**
   * Combine existing features to capture interactions or patterns that might not be apparent in individual features.
   * Generate features that represent domain-specific insights.
2. **Transformations:**
   * Apply mathematical transformations to features (logarithm, square root, etc.) to make their distribution more suitable for modeling.
3. **Encoding Categorical Variables:**
   * Convert categorical variables into numerical format using techniques like one-hot encoding or label encoding.
4. **Time-Series Features:**
   * Create lag features for time-series data to capture temporal dependencies.

**Feature Selection:** Feature selection involves choosing a subset of relevant features to include in the model. Random Forests have inherent mechanisms that can help with feature selection:

1. **Feature Importance:**
   * Random Forests provide feature importance scores, which indicate how much each feature contributes to the model's predictive performance.
2. **Recursive Feature Elimination (RFE):**
   * Train a Random Forest and iteratively remove the least important features until the model's performance starts to degrade.
3. **Threshold-Based Selection:**
   * Set a threshold for feature importance scores and include only features that exceed this threshold.

**Feature Importance:** Random Forests provide a measure of how important each feature is in making predictions. The importance is calculated based on the decrease in impurity (such as Gini impurity) caused by using a particular feature for splitting nodes in the trees. Key points:

1. **Gini Importance:**
   * The decrease in impurity (Gini impurity) for each feature is averaged across all trees in the forest.
2. **Permutation Importance:**
   * Instead of using the decrease in impurity, the model measures the decrease in predictive accuracy when the values of a feature are randomly permuted.
3. **Visualization:**
   * Feature importance can be visualized in bar charts, allowing you to quickly identify the most influential features.
4. **Interpretation:**
   * Feature importance helps understand which features are driving the model's predictions, aiding in model explanation and validation.

In summary, feature engineering enhances the quality of input features, feature selection helps choose the most relevant features, and feature importance quantifies the contribution of each feature to model performance. Utilizing these techniques in combination can lead to improved Random Forest models with better predictive power and interpretability.

**How to handle overfitting issue in Random Forests**

Overfitting can also be a concern in the Random Forest algorithm, but Random Forests are generally more resistant to overfitting compared to individual Decision Trees due to their ensemble nature. However, overfitting can still occur, especially in certain scenarios. Here's how to handle overfitting in the Random Forests algorithm:

1. **Number of Trees:**
   * Increase the number of trees in the forest. More trees tend to improve the model's generalization as the average prediction becomes more stable.
2. **Maximum Features per Split:**
   * Limit the number of features considered for splitting at each node. This can help diversify the trees and reduce their tendency to fit noise.
3. **Minimum Samples per Leaf or Split:**
   * Set thresholds for the minimum number of samples required in a leaf node or for splitting a node. Similar to Decision Trees, this prevents small subsets from being used to make decisions.
4. **Minimum Impurity Decrease:**
   * Set a threshold for the minimum impurity decrease required for a split. This can prevent the creation of noisy splits.
5. **Pruning:**
   * Prune individual trees in the forest by removing branches that do not contribute significantly to the ensemble's predictive accuracy. This can be done post-training.
6. **Feature Selection:**
   * Use techniques like Recursive Feature Elimination (RFE) to select a subset of relevant features for training the Random Forest.
7. **Validation and Cross-Validation:**
   * Validate your Random Forest model using a separate validation set or through cross-validation to ensure it generalizes well to unseen data.
8. **Hyperparameter Tuning:**
   * Experiment with hyperparameters like the number of trees, maximum features per split, and thresholds for minimum samples or impurity decrease to find the optimal settings.
9. **Data Augmentation:**
   * Apply data augmentation techniques to increase the effective size of your dataset and expose the model to more variations.
10. **Ensemble Methods:**
    * You can combine different ensemble methods or techniques like Bagging, Boosting, or Stacking with Random Forests to further enhance their performance and robustness.

Remember that while Random Forests are less prone to overfitting than individual Decision Trees, careful tuning of hyperparameters and regularization techniques can still significantly improve their generalization capabilities. Monitoring and evaluating the model's performance on validation data is crucial to ensure you strike the right balance between complexity and generalization.