Module 20: Ensemble Techniques

Quick Reference Guide

Learning Outcomes:

1. Relate the concepts of bias and variance to real-world examples.
2. Employ the concept of aggregating predictors for bagging in ensemble model.
3. Build a metamodel for classification and regression problems.
4. Evaluate the performance of various metamodels.
5. Implement bootstrapping on a dataset.
6. Implement bagging on a metamodel.
7. Apply out-of-bag evaluation in scikit-learn.
8. Articulate the key concepts and parameters of the Random Forest algorithm in scikit-learn.
9. Design random forests for classification and regression problems.
10. Implement the AdaBoost algorithm.
11. Compare the AdaBoost algorithm with gradient boosted trees.
12. Perform a deep analysis of your identified problem in order to develop an executive brief detailing your technical findings.

Wisdom of the Crowd

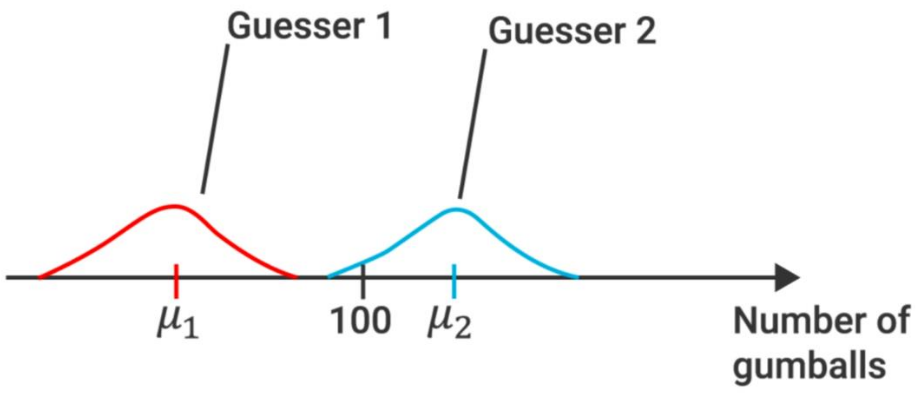
So far, you have learned how to train models for classification and regression tasks. These models include KNN, linear regression models, logistic regression models, decision trees, and support vector machines. In each case, the model is a **function** that takes a sample and returns a prediction for that sample. These types of models are sometimes called **predictors or classifiers**, depending on the context.

Up until now, you have used just one model to make a prediction. But in real life, you often make predictions based on a variety of opinions. In government, you have parliamentary debate. In democracies, you select leaders by popular vote. The idea of making decisions based on consensus is also used in machine learning, where it goes by the name of **ensemble learning**.

Now you will learn about the two most important types of ensemble learning, **bagging** and **boosting**. Both of these are general ideas with many different variations. The idea of bagging will lead you to the specific algorithm of random trees. And the two most important types of boosting are **AdaBoost** and **gradient-boosted trees**. The essence of ensemble methods is to take a large number of models, each of which is pretty bad on their own, and combine them in a way that results in a good model.

Say you want to estimate the total number of gumballs in a large jar. A mathematician might compute the volume occupied by a single gumball by, say, cubing its diameter, and then divide the volume of the jar by that number. A physicist might set up an experiment where they take some sample jar with a known volume and measure how many gumballs it fits and then extrapolate to the given jar. A statistician would take a different approach and instead ask many people to venture a guess and then take the average of those guesses.

Here is a graph that illustrates how a statistician would see it.

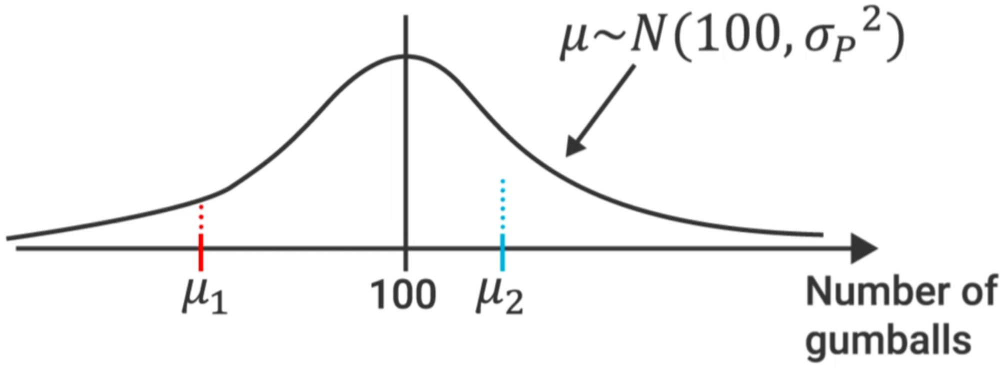


Say the jar contains exactly 100 gumballs. You have two guessers. Guesser 1 tends to guess low. Their expected value is less than 100. Guesser 2 tends to guess high. So they are both biased, but Guesser 1 is **negatively biased** and Guesser 2 is **positively biased**. Assume that their guesses follow a normal distribution with expected values μ_1 and variance σ^2. For simplicity, assume that they have the same variance.

$$G_1 \sim N(\mu_1, \sigma^2)$$

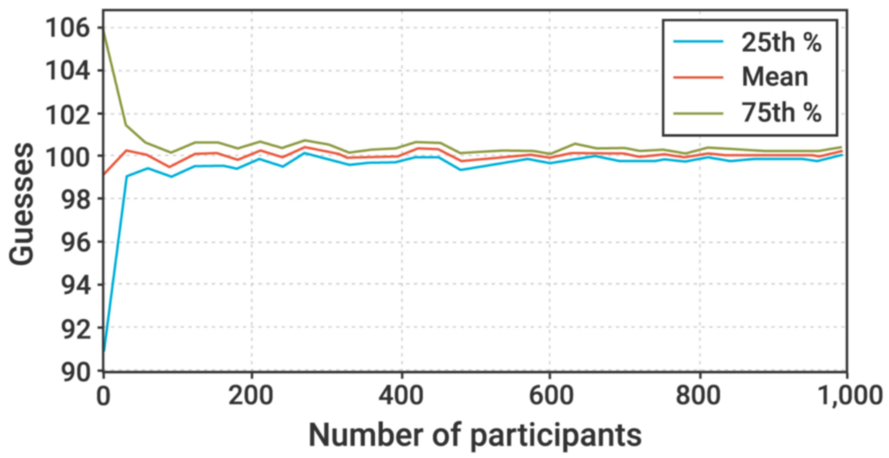
$$G_2 \sim N(\mu_2, \sigma^2)$$

Continuous normal distributions are perhaps not the best choice for guessing gumballs, but this will do for the time being. Also assume that the expected values, μ, are themselves chosen from a normal distribution whose mean is the correct value of 100 gumballs.



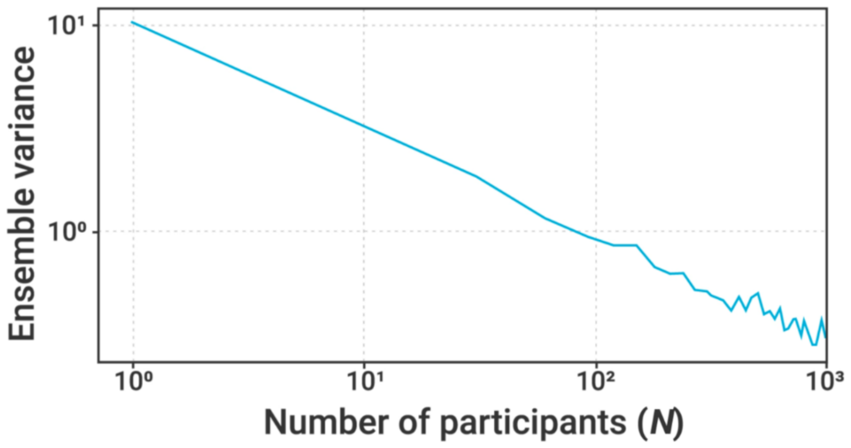
Even though each of the individuals may be biased, the population as a whole is not biased. This unbiasedness of the population is the **wisdom of the crowd**. This is what you try to access when you crowdsource your decisions.

Now, consider a simulation of this scenario with the number of guessers varying from a single person to 1,000 participants.



For each crowd size, you ran 50 rounds of ensemble voting. In each one of these rounds, every participant produced a single guess. Then you average all of the guesses to obtain a final estimate for that round. The lines in the plot show the mean of the 50 rounds, as well as the upper and lower quartiles. While the mean is close to 100 for all crowd sizes, the variance in the estimate decreases as the number of participants goes up.

This is evident in a plot of the variance, which shows a decreasing linear trend in a log-log plot.



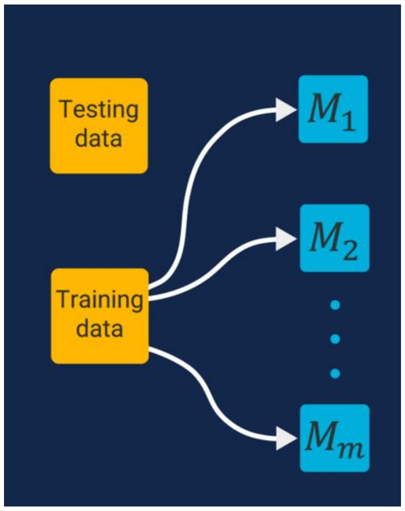
$$Variance[\text{ensemble}] = \frac{Variance[\text{individual}]}{N}$$

That means the variance is decreasing as 1 over the number of participants. This is the formula for the variance of the sample mean when the samples are independent. This is a very important condition for trusting the wisdom of the crowd. The aggregate of decisions produced by individuals will be trustworthy only insofar as those decisions are made independently of each other. This requirement is also true for machine learning models.

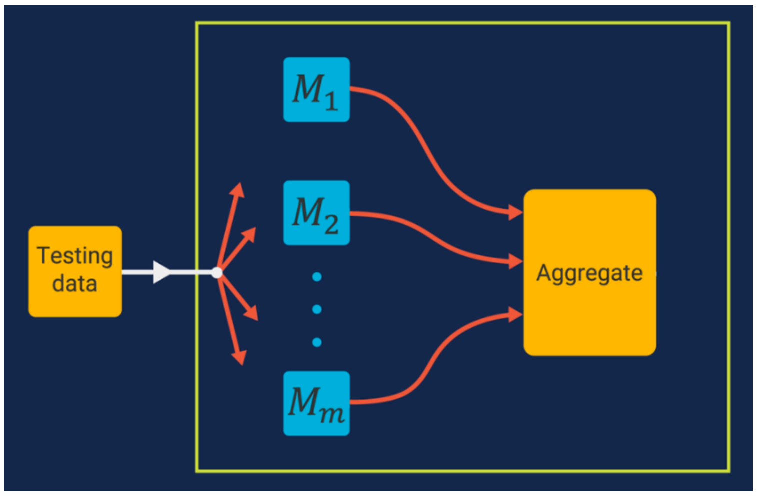
Imagine you live in a village in a time before modern meteorology and you need to know whether it will rain or not in the next few days. Your best strategy might be to ask everyone in the village what they think and then decide by majority vote. In machine learning, you would regard each of the villagers as a classifier who takes as input what they see, smell, and feel, and produces a binary output, rain or no rain. Each one has been trained through a lifetime of experience. Some of them will be highly biased. Their decision rules are very simple. Sun is out, no rain. Simple, but also often wrong. Others have high variance. They remember and apply many details of their past experience, but to a fault. They are also often wrong. The reasonable thing to do is to decide by majority vote. As long as the collective bias is small and the villagers are allowed to vote independently, then their individual biases and variances will cancel each other out.

Aggregating Predictors (Part 1)

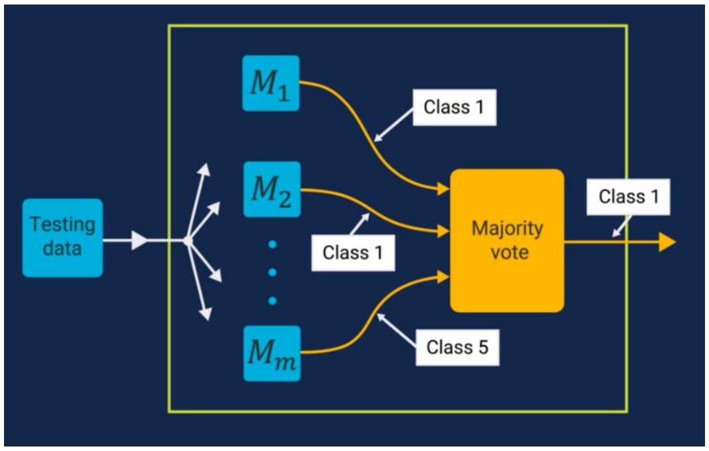
You can now apply the wisdom of the crowd to learning algorithms. As you have always done, you will split the dataset into training and testing data. Except now, instead of training just one model with the data, you will train many models.



These could be anything. You could have logistic regression with k-nearest neighbors and decision trees. Or you could also consider many slight variations on the same type of model. All of the models fit into a larger metamodel. The **metamodel**, or **ensemble model**, receives a test data point and is in charge of distributing that sample to the component models and then aggregating their predictions into a final decision.

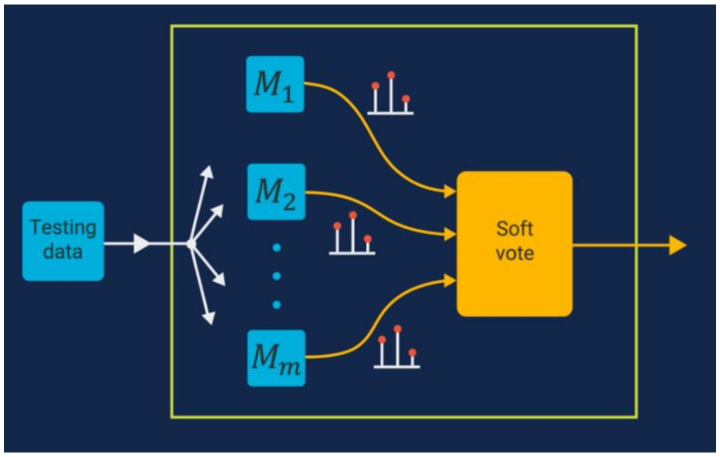


There are different ways of performing both of these tasks. The method for performing aggregation depends primarily on whether it is a **classification** or a **regression** task. If the task is **classification**, then the final decision can be reached by a **majority vote**, also known as **hard voting**.



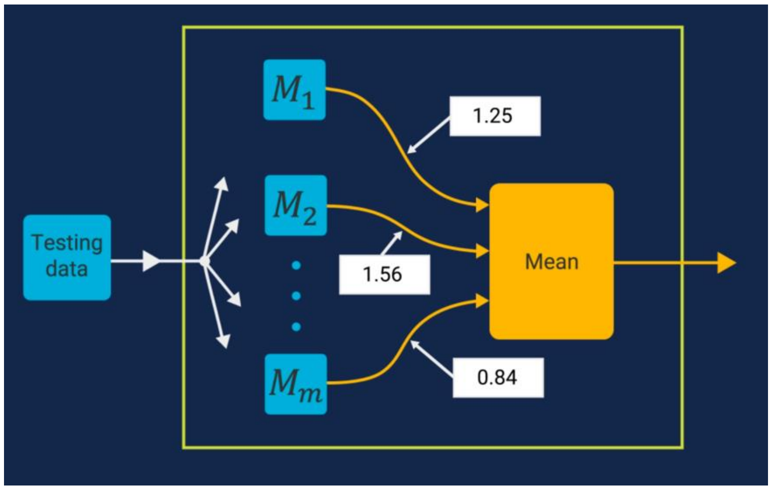
That means you choose the class that receives the most votes as the prediction for the given input. This can be done whether it is a binary two-class problem or a multiclass problem.

Some models produce not only a predicted class label, but also a probability of class membership for each class. So a data point may have 30% chance of belonging to class A, 20% chance for class B, et cetera. **Logistic regression** is an example of such a model.

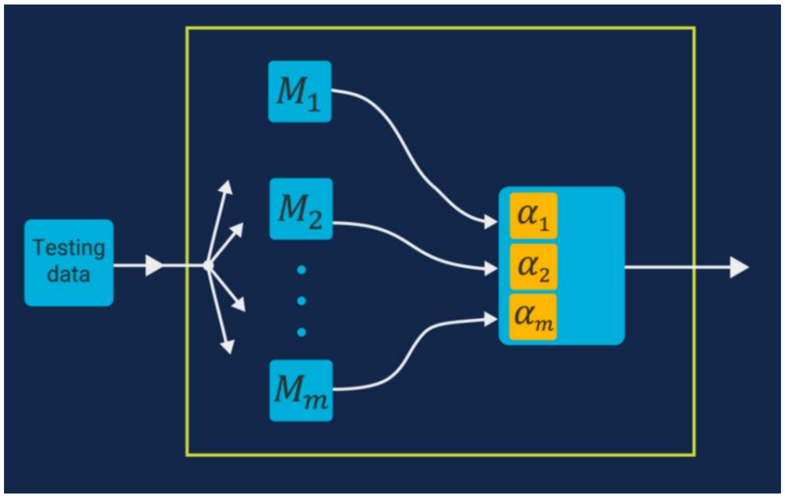


In this case, the aggregator can perform a **soft vote** in which you average all of the predicted class distributions and then choose the maximum.

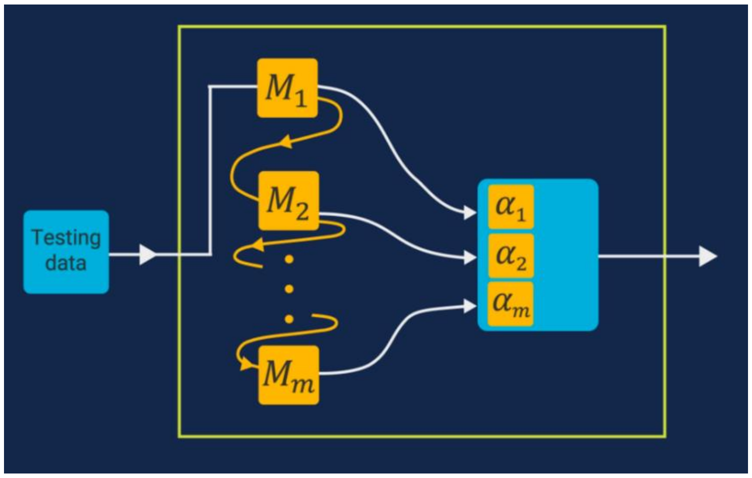
For **regression** problems, the outputs of the model are combined by simple averaging.



In all cases, the metamodel can also assign **weights**, α_1, through α_m, to the individual models, and thus give each one a greater or smaller influence in the final decision.



In **boosting**, the **predictors are trained sequentially**, with the output of one model influencing the inputs of the next model, and so on.



Aggregating Predictors (Part 2)

Next, you can look at a short program in which you measure the performance of four different classifiers on the Iris dataset and compare them to the performance of the ensemble. You have already loaded the dataset X and y.

The main loop in the code iterates the exercise 100 times, so that you get a more robust sense of the result. The code begins by allocating a couple of NumPy arrays that will hold the accuracy metrics for each of the four models and another for the ensemble.

```
numit = 100
model_acc = np.empty((numit,4))
ensemble_acc = np.empty(numit)
```

Within the loop, you begin by creating four classifiers and storing them in a list called **clfs**.

```
clfs = [
      Pipeline([ ('sc',StandardScaler()),
                  ('clf',LogisticRegression())
                 ]),
      Pipeline([ ('sc',StandardScaler()),
                  ('clf.,SVC())
                 ]),
      DecisionTreeClassifier(),

      Pipeline([ ('sc',StandardScaler()),
                  ('clf',KNeighborsClassifier())
                 ]),
]
```

The classifiers are: **LogisticRegression**, a support vector classifier, a decision tree, and k-nearest neighbors. You will couple each of these models, except for the decision tree, with a **StandardScaler**, and include them in a pipeline. It is not useful to do this for the decision tree because decision trees are **scale invariant**, so their output is the same whether or not you normalize the input. But it is good practice to normalize the input of the other models using a **StandardScaler**, in order to avoid numerical problems associated with having very different scales amongst features.

Next, you split the data into training and testing sets, reserving 20% of the data for testing.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
```

You then iterate through each of the four models.

```
y_pred = np.empty( (len(clfs),len(y_test)) , int)
for i, model in enumerate(clfs):
    model. fit(X_train,y_train)
    y_pred[i,:] = model.predict(X_test)
    model_acc(c,i] = accuracy_score(y_test,y_pred[i,:])
```

Within the iteration, you first train the model and then you test it by producing a prediction on the test data and comparing that prediction to the actual test output. To obtain an ensemble prediction, you aggregate the predictions of all of the models by taking a majority vote.

```
y_pred_ens = scipy.stats.mode(y_pred).mode[0]
ensemble_acc[c] = accuracy_score(y_test,y_pred_ens)
```

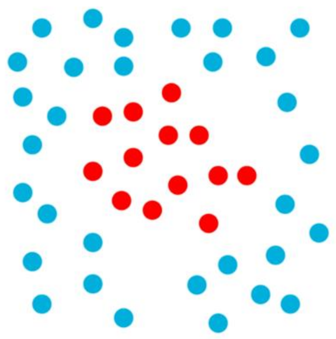
A simple way of implementing a majority vote is with the **mode** function of **scipy.stats**. This function takes in the predictions which, in this case, had been stored in **y_pred**, and it returns an object. To extract the actual mode, you must retrieve the 0th element in the mode attribute of the returned object. You can then compute the accuracy of the ensemble prediction and store it in its allocated array. After iterating this 100 times, you take the mean of the accuracies to obtain the final result for each model.

```
np.mean(model_acc,axis=0), np.mean(ensemble_acc)
(array([0.9335, 0.922 , 0.8835, 0.9195]), 0.9215000000000001)
```

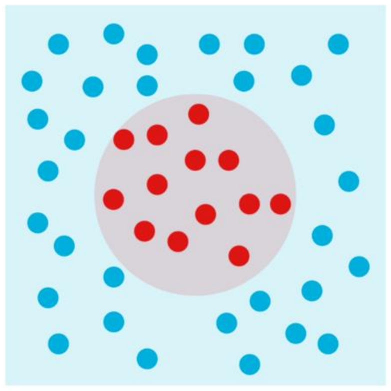
The logistic regression classifier on its own, achieved the highest accuracy of about 93%, followed by the support vector machine with 92%, and k-nearest neighbors with 91.95%. The decision tree came in last with 88%, but don't count them out. Decision trees are actually the stars of ensemble methods. The ensemble did pretty well with 92%, but it was certainly not the winner. Its performance was a little worse than logistic regression and about equal to the support vector classifier. The reason for this has to do with the main criterion for trusting the wisdom of the crowd: Their decisions should be independent. In this case, because all of the models were trained on exactly the same data, they turned out to be highly correlated.

Bootstrapping and Bagging (Part 1)

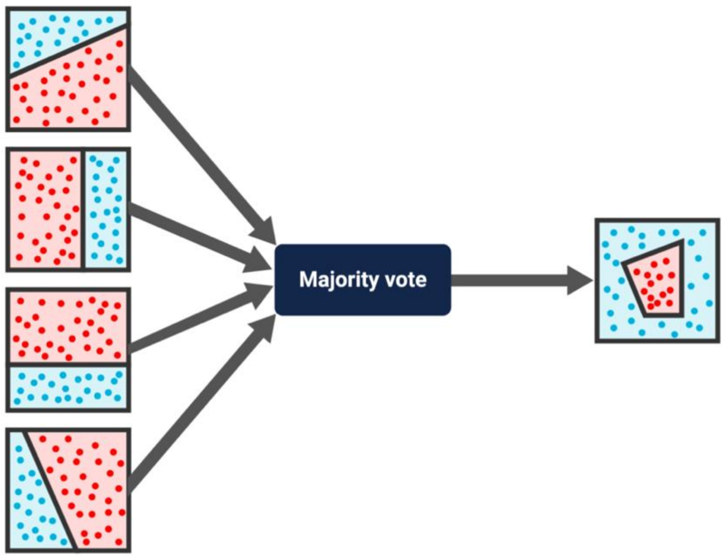
Calling on the wisdom of the crowd can help you in two ways. It can help to reduce bias and it can help to reduce variance. **Bias** is exhibited by models that are **too simple** for the data they are trying to match. For example, if you tried to capture the circular red class shown with a **linear decision boundary**, this will invariably produce large errors.



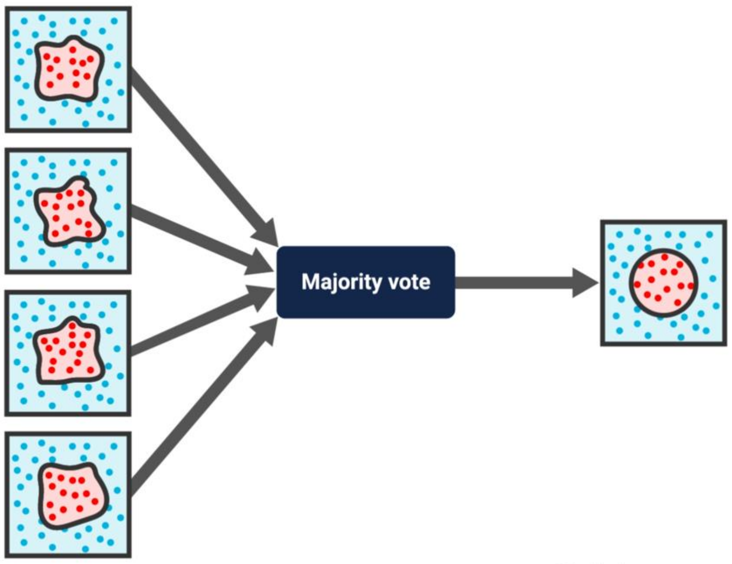
A good model for this type of data should have a **nonlinear decision boundary** that surrounds the red points.



However, by aggregating a large number of simple linear estimators by majority vote, you can obtain a model with a nonlinear decision boundary.



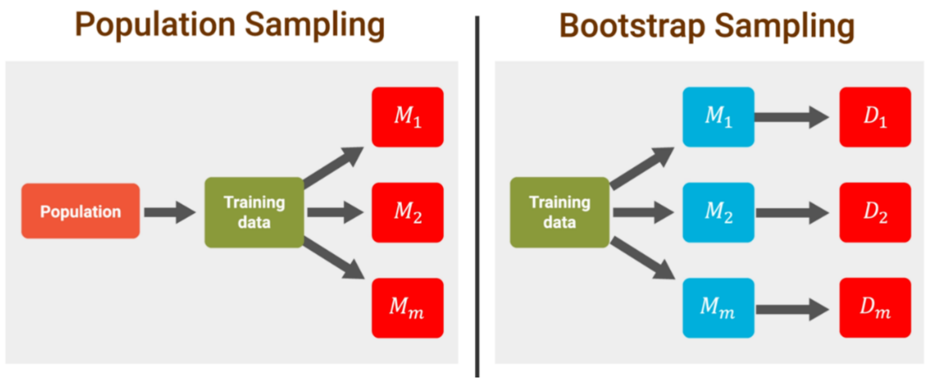
Aggregation can also help with models that have too much variance. In contrast to bias models, these models are too flexible and they try too hard to fit the training data. Researchers have found that it is best to build the ensemble with many variations of the same type of model, instead of using many different types.



When you do this, you put yourself into one of the two cases. Either the base models have a **bias problem** or they have a **variance problem**. The approach that you take to building the ensemble will be different in either of the two cases. If your base models have high variance, then bagging will be the solution. If they are highly biased, then boosting will help. You will begin with bagging.

The problem of high variance is that the models are too myopically focused on the training data. If there is too little data, then they have learned to behave strictly according to that little bit of experience. A simple solution would be to get more data and to train each of the component models with a different dataset. The problem, of course, is that you do not have more data. Data can be expensive. And if you had more data, you might instead decide to gather it all into one giant dataset and use that to train a single, more sophisticated model. So you will discard that possibility and instead assume that the data that you have is all that you are going to get. It is as if you are trapped in a hole. In this situation, you have to pull yourself out of the hole by your bootstraps, because there is no other way out. This is the name of the technique that you will use, the **bootstrap**.

The idea of the bootstrap is both simple and very consequential in statistics and machine learning.



The idea is this: The training dataset that you have was sampled from the population. You no longer have access to that population. So you do the next best thing, which is to sample new datasets from the dataset that you have. That is, you imagine that your training data is itself the population. It is very cheap to sample from a dataset that you already have. The key, however, is that this sampling has to be done **with replacement**. This means that having sampled a data point, this data point still remains available for future samples. Sampling with replacement makes the training data seem more like an infinite population and it increases the variation in the bootstrapped samples.

Now consider an example of how sampling with replacement works. You have a list, D, of 10 numbers from 0 to 9, and you wish to generate a new list of 10 numbers by sampling with replacement.

You first pick a number at random, and say you pick a 3. You add a 3 to your new list, D1, but you do not remove it from the original dataset. It is still there and available to be picked again. Then you pick a second value, and a third, and a fourth. And then perhaps again, you find a 3. So when you sample with replacement, you can get repetitions in the new dataset. This means that the list, D1, which also contains 10 numbers, may not contain all of the numbers from the original dataset.

How many of the items in the original dataset should you expect to find in the bootstrap sample and how many are not selected? The answer to this question can be found with a little bit of math.

First, in a dataset with N items, the chance of any one being chosen in a single draw can be calculated with the following formula:

$$\frac{1}{N}$$

The chance of not being chosen can be calculated with the following formula:

$$1-\frac{1}{N}$$

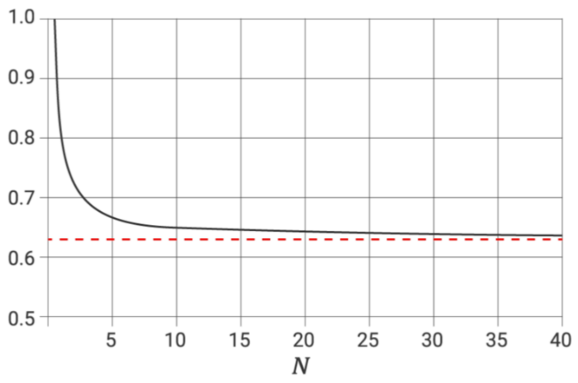
Because all of the draws are independent, the chance of not being chosen in any of the N draws can be calculated with the formula:

$$\left(1-\frac{1}{N}\right)^N$$

The chance of an item being included at least once in the bootstrap sample can be calculated with the formula:

$$1-\left(1-\frac{1}{N}\right)^N$$

This plot shows the evolution of this quantity, as the size of the training dataset gets larger.

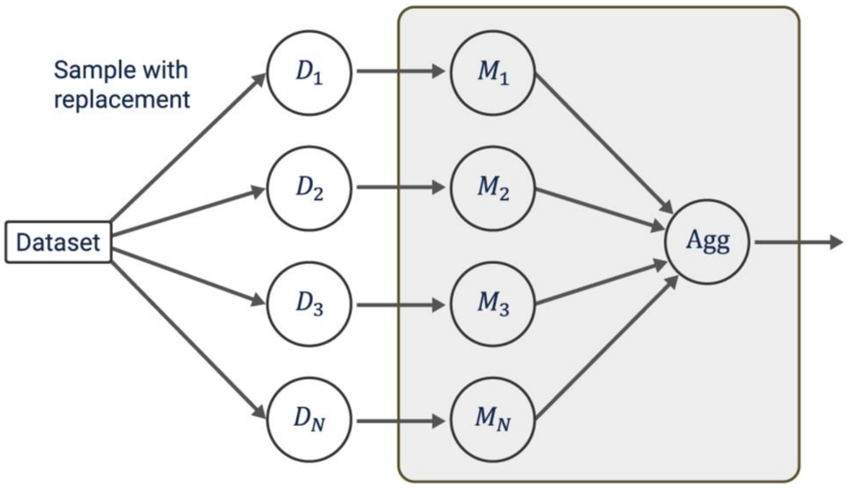


$$\lim_{N\to\infty}\left(1-\left(1-\frac{1}{N}\right)^{N}\right) = 1-\frac{1}{e} \cong 0.632$$

The dash line is the asymptotic tendency as N goes to infinity, and it can be shown that this equals 1 minus 1 over e, which is approximately 0.632. So you can conclude that in any bootstrap sample of moderate to large size, you should expect to see about 63% of the items from the training dataset. The remaining 37% will consist of repetitions.

Bootstrapping and Bagging (Part 2)

The technique of **bagging** is simply to use a bootstrap sampler to generate training data for an ensemble model.



Bagging stands for **bootstrap aggregation**. The bootstrap samples help to decorrelate the models and hence to increase the power of the ensemble.

Bagging can work with many models, both for classification and for regression. But it works particularly well with decision trees. Since the averaging step will help to reduce the variance, you are not so concerned with the component models having a high variance, but you do want to choose models with low bias. So if you are using decision trees, you would like those trees to be relatively deep, since deep trees have high variance and low bias.

Here is some sample code. You are going to create 1,000 decision trees for your ensemble.

num_trees = 1000

You are given the training dataset **X_train** and **y_train**, as well as the testing data **X_test** and **y_test**. You iterate through the trees and for each one you create a bootstrapped training dataset.

```
ind = np.random.choice(range(ntrain),ntrain)
X_bs = X_train[ind,:]
y_bs = y_train[ind]
```

This is accomplished with NumPy's choice function, which samples with replacement from the index set of the training data. The next step is to create and train a decision tree based on this bootstrap dataset and to add it to your list of trees.

```
tree = DecisionTreeClassifier().fit(X_bs,y_bs)
trees.append(tree)
```

You allow each of the trees to grow until all of their leaves are pure. This is the default behavior for the decision tree classifier in scikit-learn.

Finally, you compute the accuracy of the tree on the test set.

```
y_pred[I,;] = tree.predict(x_test)
```

```
all_acc[i] = accuracy_score(y_test,y_pred[I,;])
```

The average accuracy for the 1,000 individual classification trees is 90.22%.

```
all_acc.mean()
0.9021999999999999
```

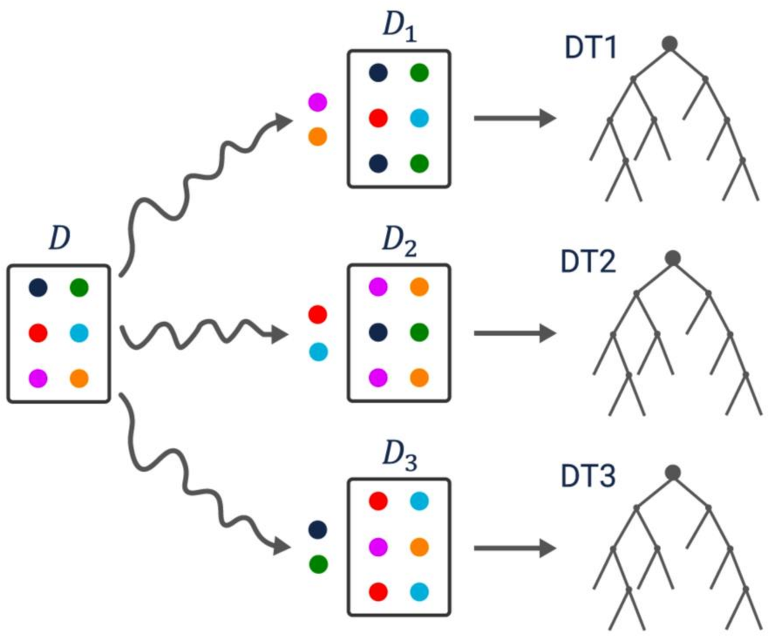
The prediction for the ensemble is found by majority vote among all of the 1,000 trees. And this is, again, done using the mode function from scipy.stats.

```
y_ensemble = scipy.stats.mode(y_pred).mode[0]
accuracy_score( y_test, y_ensemble )
0.95
```

The accuracy for the ensemble is significantly higher than the average accuracy of the trees, which is impressive since the accuracy was already pretty high.

In the example you just did, you evaluated the accuracy of the individual trees and of the ensemble using a test dataset, which you had set apart before the training began. With bagging, however, there is another more attractive option called **out-of-bag** evaluation.

Consider an example where the dataset consists of six points, drawn here as blue, green, red, cyan, magenta, and orange dots.



You know that if you draw bootstrap samples from this dataset, you will get new datasets, D_1, D_2, and D_3, each of which will contain only about two-thirds of the original data. The remaining one-third are called out-of-bag samples. Of course, the out-of-bag samples will be different for each of the bootstrap datasets.

For D_1, for example, the out-of-bag samples are the magenta and orange dots. For D_2, they are the red and cyan, and for D_3, they are the blue and the green. This means that each data point will be in-bag for some trees and out-of-bag for others. The red dot is in-bag for the decision trees 1 and 3, and out-of-bag for decision tree number 2. You can then make a test prediction for a sample in D, if you only consider those trees for which that point is out-of-bag. A test prediction for the red dot can be made using a sub ensemble consisting only of tree number 2.

With out-of-bag testing, each data point is evaluated based on about one-third of the total ensemble. The technique has the advantage that it allows for the entirety of the dataset to be used for both training and testing. Scikit-learn includes a **BaggingClassifier** class that makes experimenting with bagging extremely easy.

```
from sklearn.ensemble import BaggingClassifier
model = BaggingClassifier( DecisionTreeClassifier(), n_estimators=500)
model.fit(X_train, y_train)
accuracy_score( y_test, model.predict(X_test) )
0.85
```

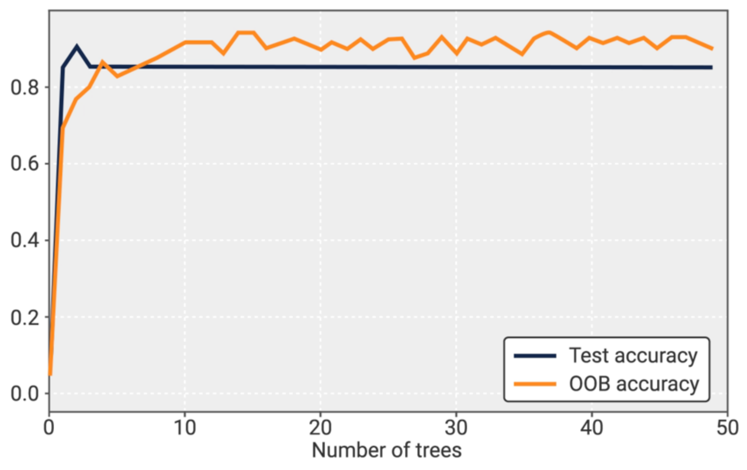
All one has to do is to pass a template model into the constructor, in this case a decision tree, as well as the number of models that one wishes to include in the ensemble. The rest is identical to the single model case.

Training is performed as usual with the **fit** method and prediction with the **predict** method.

To request an out-of-bag score for the **BaggingClassifier**, one must pass the parameter **oob_score=True** to the constructor. The out-of-bag score is then evaluated upon training of the models and made available in the **oob_score_ attribute** of the **BaggingClassifier** object.

```
model = BaggingClassifier( DecisionTreeClassifier(), n_estimators=500,
oob_score=True)
model.fit(X_train, y_train)
model.oob_score_
0.925
```

How many estimators should be included in the ensemble? The answer to this question depends, of course, on many factors; including the computational resources that are available. This plot shows that at some point it can become pointless to continue growing the model.



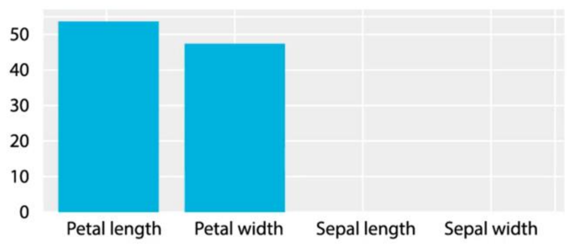
In this simple case, that point is reached quickly after about 20 trees.

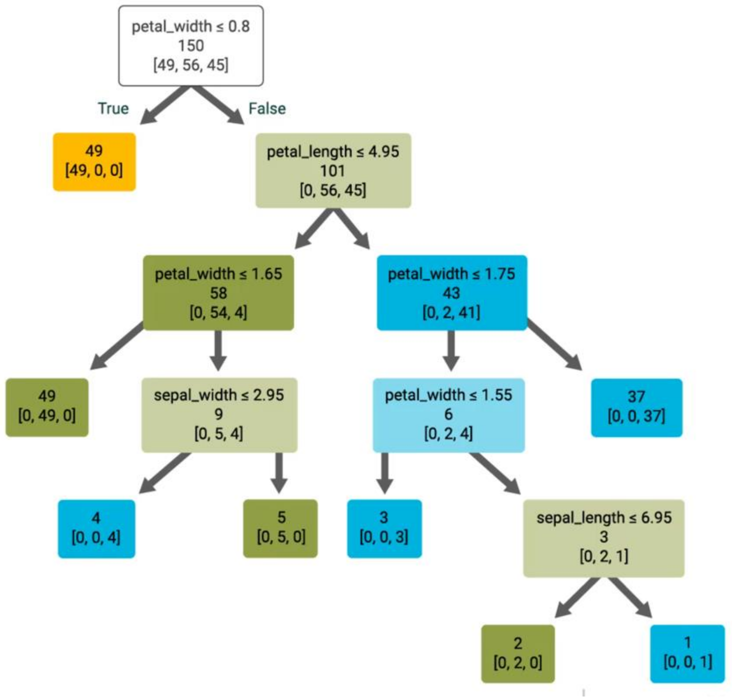
Random Forests (Part 1)

So aggregating models, and specifically decision trees, can improve performance. This can be explained by the wisdom of the crowd. But this wisdom depends on the capacity of the individuals to make their decisions independently from the rest. Are the trees that you created with bagging truly independent from one another? When you use the bootstrap, or sampling with replacement, they are indeed less correlated than they would have been otherwise. But ultimately the data is still coming from a single dataset. You saw that each bootstrap sample will contain about two-thirds of the original samples. There will be many samples in common in each of the bags and, hence, the models remain somewhat correlated.

Random forests is an algorithm that reduces the correlation amongst trees in a bagged ensemble by introducing randomness into the training process itself. Recall the traditional decision tree generation algorithm. At every iteration of the algorithm, you created two new branches by splitting a node along the feature and threshold that yield the greatest reduction in entropy.

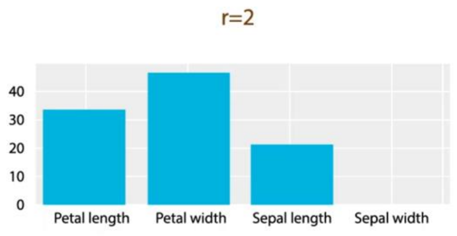
To see this, consider what happens if you build 100 separate decision trees based on bootstrap samples from the Iris dataset.

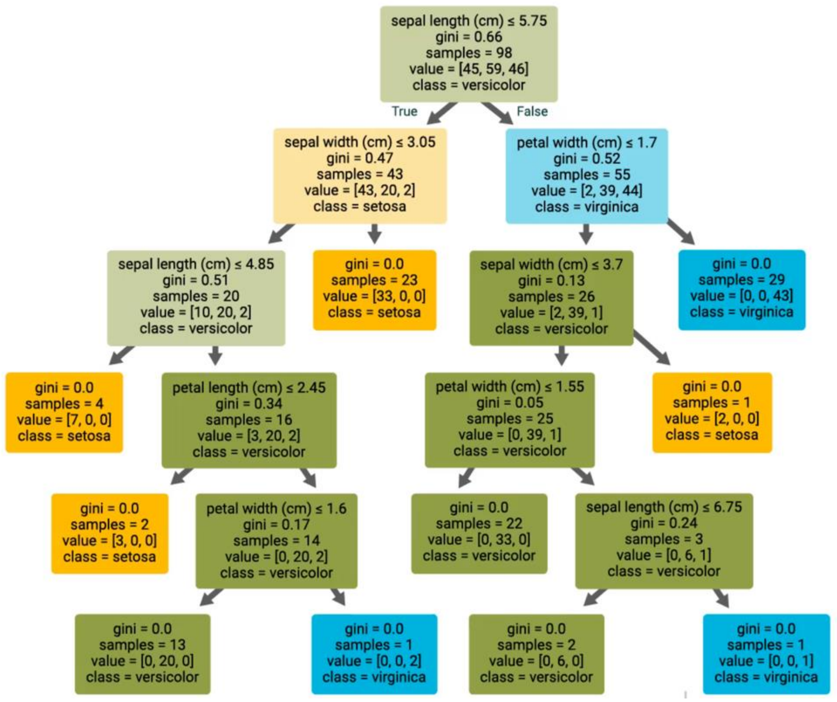




This bar plot shows the number of times that each feature was chosen as the topmost split of the tree. In 53% of the trees, the first split occurred along the petal length, and in 47%, it occurred along the petal width. But in none of the bootstrapped trees was either the sepal length or the sepal width chosen as the root-splitting feature. This is reasonable because entropy can be most effectively reduced by splitting along petal length and petal width. So sepal length and width are poor choices. However, from the ensemble's perspective, it is missing out on an entire class of decision trees, where sepal length and sepal width play a more important role.

The random forests algorithm seeks to remedy the situation by searching from a randomly-chosen subset of the features at every split in the tree. In scikit-learn's implementation, this parameter is called **max_features**.

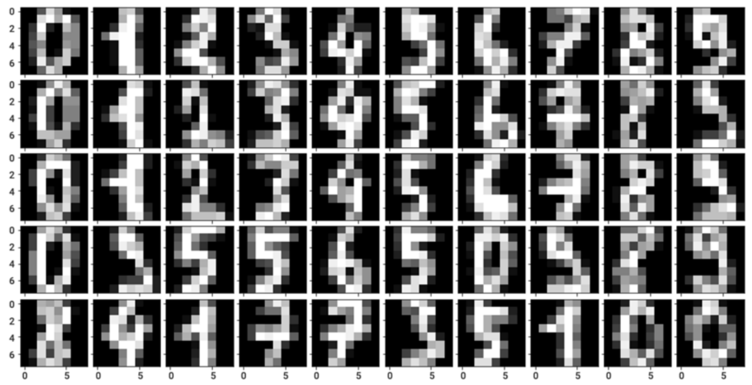



sepal length (cm) ≤ 5.75 gini = 0.66 samples = 98 value = [45, 59, 46] class = versicolor
True
False
sepal width (cm) ≤ 3.05 gini = 0.47 samples = 43 value = [43, 20, 2] class = setosa
petal width (cm) ≤ 1.7 gini = 0.52 samples = 55 value = [2, 39, 44] class = virginica
sepal length (cm) ≤ 4.85 gini = 0.51 samples = 20 value = [10, 20, 2] class = versicolor
gini = 0.0 samples = 23 value = [33, 0, 0] class = setosa
sepal width (cm) ≤ 3.7 gini = 0.13 samples = 26 value = [2, 39, 1] class = versicolor
gini = 0.0 samples = 29 value = [0, 0, 43] class = virginica
gini = 0.0 samples = 4 value = [7, 0, 0] class = setosa
petal length (cm) ≤ 2.45 gini = 0.34 samples = 16 value = [3, 20, 2] class = versicolor
petal width (cm) ≤ 1.55 gini = 0.05 samples = 25 value = [0, 39, 1] class = versicolor
gini = 0.0 samples = 1 value = [2, 0, 0] class = setosa
gini = 0.0 samples = 2 value = [3, 0, 0] class = setosa
petal width (cm) ≤ 1.6 gini = 0.17 samples = 14 value = [0, 20, 2] class = versicolor
gini = 0.0 samples = 22 value = [0, 33, 0] class = versicolor
sepal length (cm) ≤ 6.75 gini = 0.24 samples = 3 value = [0, 6, 1] class = versicolor
gini = 0.0 samples = 13 value = [0, 20, 0] class = versicolor
gini = 0.0 samples = 1 value = [0, 0, 2] class = virginica
gini = 0.0 samples = 2 value = [0, 6, 0] class = versicolor
gini = 0.0 samples = 1 value = [0, 0, 1] class = virginica

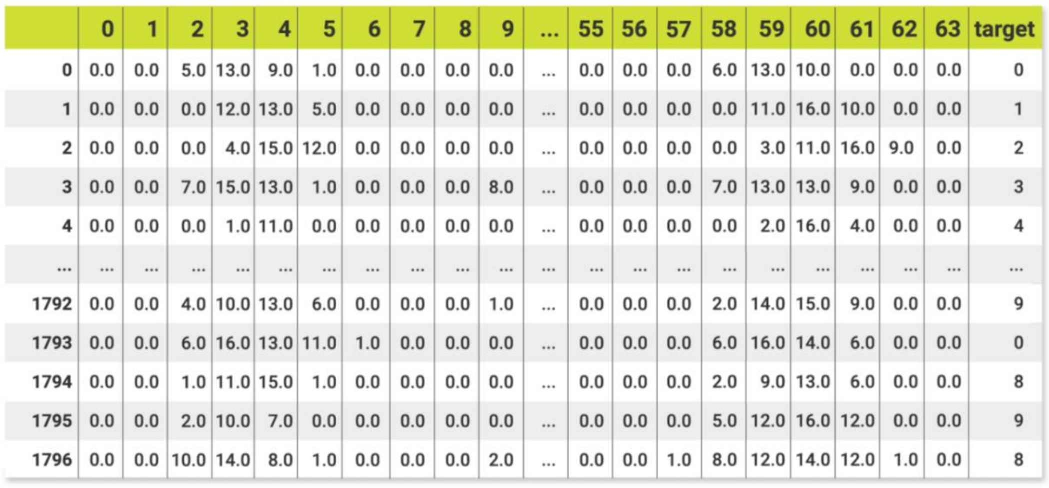
By setting **max_features** to 2, you are telling the algorithm to design each split based on a randomly chosen set of two of the four features. This means that in about one-sixth of the cases, this subset will consist only of sepal length and sepal width. And hence, one of those must be chosen in about one-sixth of the cases. This may result in trees that are individually not as strong as they might have been. However, it increases the diversity of the ensemble, and this turns out to be very beneficial.

Random Forests (Part 2)

You can test random forests on the task of recognizing images of handwritten digits.



The dataset comes from NIST, the National Institute of Standards and Technology. It consists of 1,797 low resolution grayscale images of handwritten digits from 0 to 9. Each image has only 64 pixels, and each of these pixels is taken as an input feature. Here is what it looks like as a pandas DataFrame.



	0	1	2	3	4	5	6	7	8	9	...	55	56	57	58	59	60	61	62	63	target
0	0.0	0.0	5.0	13.0	9.0	1.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	6.0	13.0	10.0	0.0	0.0	0.0	0
1	0.0	0.0	0.0	12.0	13.0	5.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	11.0	16.0	10.0	0.0	0.0	1
2	0.0	0.0	0.0	4.0	15.0	12.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	3.0	11.0	16.0	9.0	0.0	2
3	0.0	0.0	7.0	15.0	13.0	1.0	0.0	0.0	0.0	8.0	...	0.0	0.0	0.0	7.0	13.0	13.0	9.0	0.0	0.0	3
4	0.0	0.0	0.0	1.0	11.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	2.0	16.0	4.0	0.0	0.0	4
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
1792	0.0	0.0	4.0	10.0	13.0	6.0	0.0	0.0	0.0	1.0	...	0.0	0.0	0.0	2.0	14.0	15.0	9.0	0.0	0.0	9
1793	0.0	0.0	6.0	16.0	13.0	11.0	1.0	0.0	0.0	0.0	...	0.0	0.0	0.0	6.0	16.0	14.0	6.0	0.0	0.0	0
1794	0.0	0.0	1.0	11.0	15.0	1.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	2.0	9.0	13.0	6.0	0.0	0.0	8
1795	0.0	0.0	2.0	10.0	7.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	5.0	12.0	16.0	12.0	0.0	0.0	9
1796	0.0	0.0	10.0	14.0	8.0	1.0	0.0	0.0	0.0	2.0	...	0.0	0.0	1.0	8.0	12.0	14.0	12.0	1.0	0.0	8

Each row contains a single image. The 64 columns are the shades of the 64 pixels, which take values from 0 to 16. The last column is the digit represented in the image.

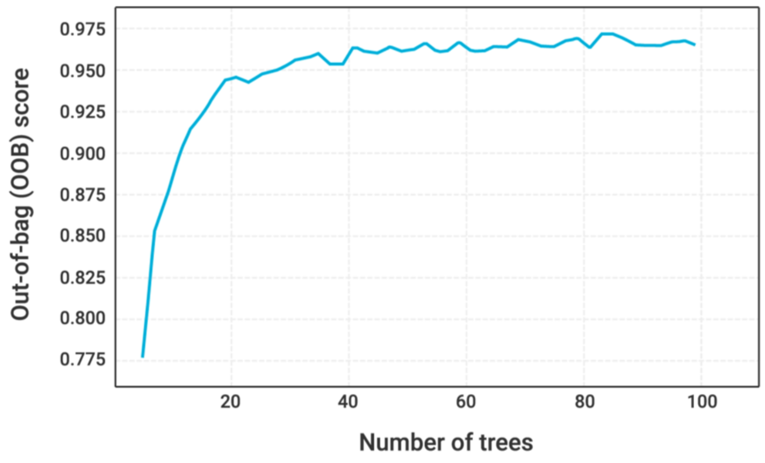
Scikit-learn has a **RandomForestClassifier** class that makes working with this relatively complicated model very easy to do.

```
from sklearn.ensemble import RandomForestClassifier
model = RandomForestClassifier(n estimators=30, max_features=10,
oob_score=True)
model.fit(X, y)
model.oob_score_
0.9526989426822482
```

The constructor takes in all of the parameters related to its decision trees. Parameters, such as the maximum depth of the tree, the splitting criterion, whether gini or entropy, et cetera. It also takes in the higher-level parameters that apply to the forest itself. The most important of these is **n_estimators**, the number of trees in the forest.

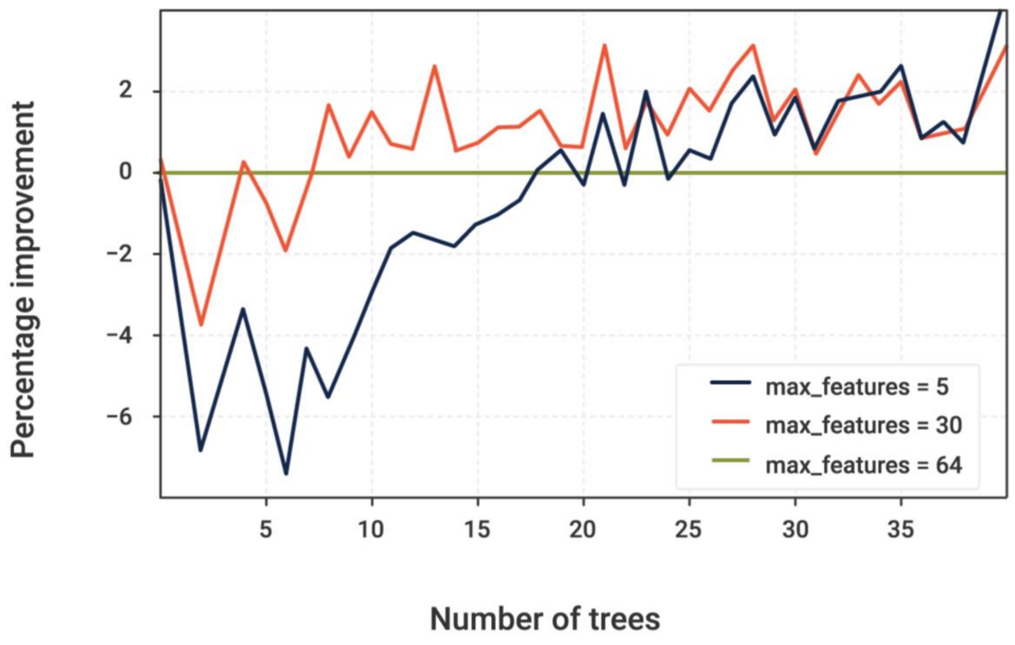
As stated, **max_features** is the number of features to be used when splitting nodes. And in this case, you are requesting that the out-of-bag score also be computed. After training the model with the **fit** function, you can retrieve the out-of-bag score. This random forest, with 30 trees and **max_features** set to 10, achieves a classification score of about 95%.

Here, you can appreciate the dependence of the out-of-bag score on the size of the random forest.



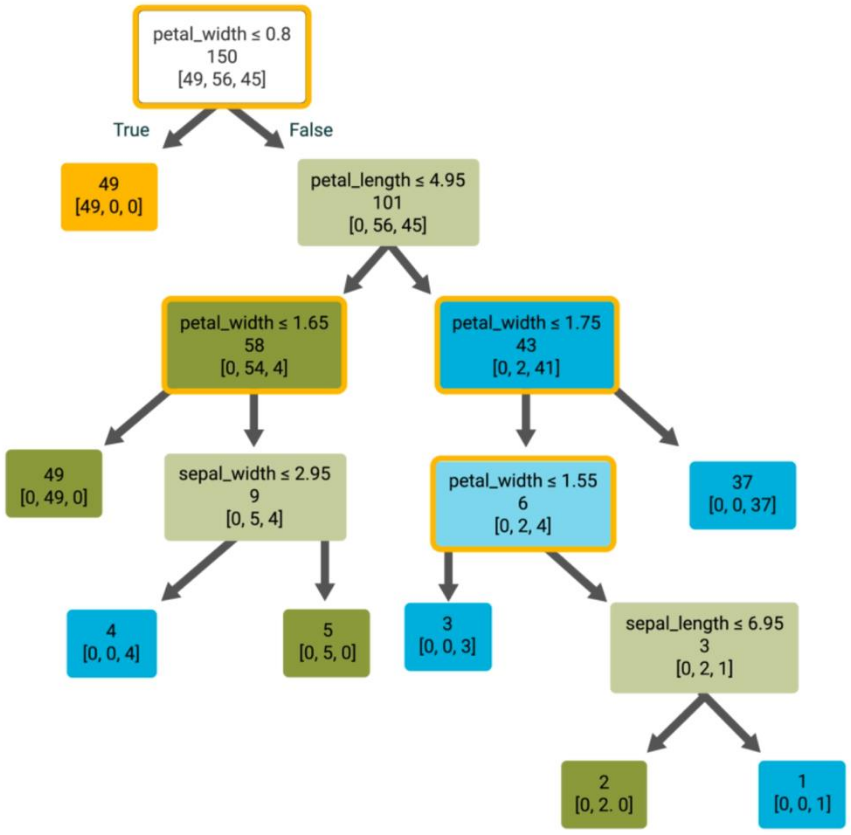
As you increase the number of trees, the score increases rapidly at first, but then settles at around 97. A plot like this can be helpful for selecting the ideal size of the forest. In this case, I would say that about 80 trees works well. However, this may also depend on factors such as computational resources and how quickly you need the predictions to be computed.

This plot demonstrates the benefits of increasing the diversity of the forest, by restricting the set of features used to split the nodes.



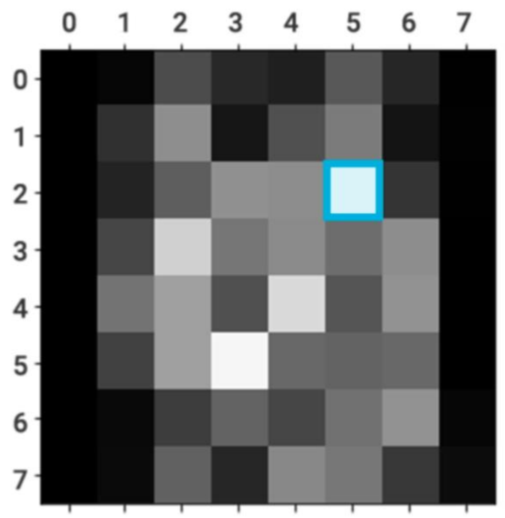
The blue and orange lines show the difference in accuracy obtained with **max_features** set to either 5 or 30, relative to using all of the 64 features. The baseline of using all 64 features is depicted with a green line at 0. Using all of the features is a better choice when you have very few trees. But for a large forest, you get about a 2% boost in performance by restricting the number of features to as little as 5.

Another very nice thing about a random forest is that it can be used to compute the relative importance of the features.



The importance of a given feature is measured by finding all of the nodes in the forest that split along that feature, and then adding up the reductions in entropy that they produced, weighted by the number of data points in each node. A relatively important feature will be responsible for a larger share of the total reduction in entropy. In this way, the random forest can produce a score that ranks all of the features in terms of their importance.

Here you see the feature importance score for each of the pixels in the digit recognition task.

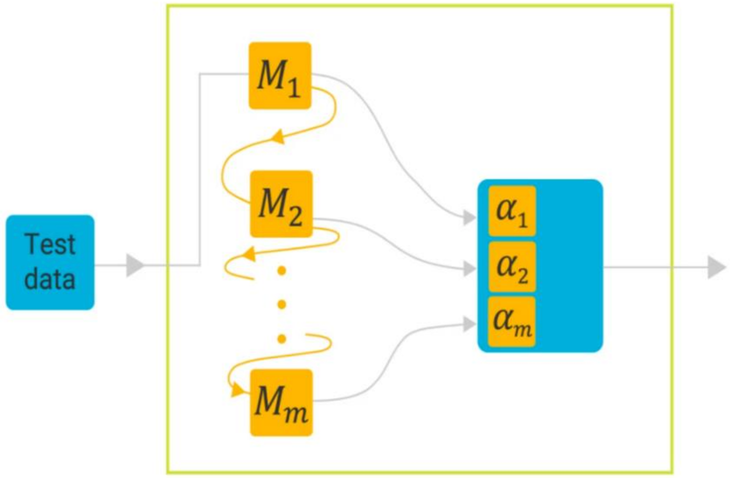


The lighter colored pixels are ones whose values are the most important for identifying the number in an image. It looks like the pixel in the third row and sixth column is the most valuable one. A camera that lost this pixel would be more severely hobbled in its ability to recognize digits than one that loses any, or maybe even all, of the pixels in the left and right columns.

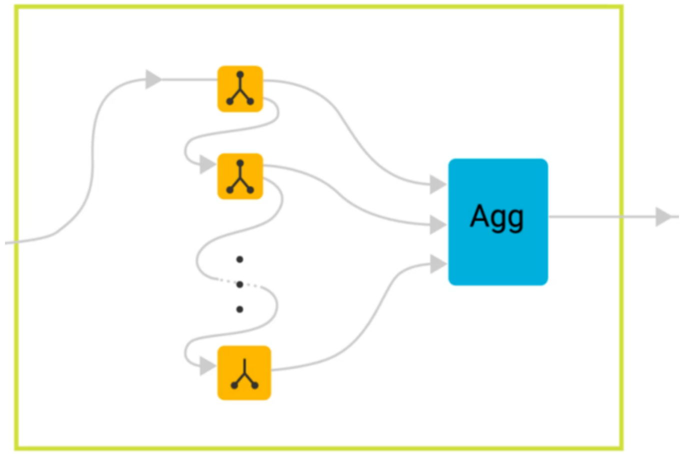
AdaBoost

Next, you will explore a different class of ensemble methods, called **boosting** methods. In contrast to bagging, which is useful for models with too much variance, boosting works well for models with too much bias. You will learn about the two most popular boosting algorithms, **AdaBoost** and **gradient-boosted trees**.

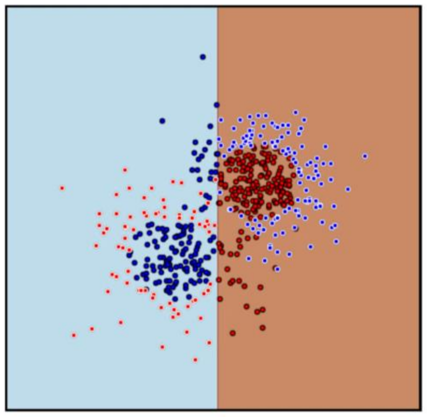
The bagging ensembles you studied previously had a **parallel architecture**. That is, the training of the constituent models could be performed independently. Boosting ensembles, on the other hand, are built by stringing together a number of so-called weak models.



A **weak model** is a very simple model, that due to its simplicity, cannot perform very well on the given training data. In the context of classification, a weak classifier is one that performs only slightly better than random guessing. The idea of boosting is to combine a large number of weak models in a clever way, such that together they make a good or strong model.



AdaBoost was the first hugely successful boosting algorithm. AdaBoost is primarily a classification algorithm. The weak classifiers it employs can in principle be anything. However, they are usually very shallow decision trees called decision stumps. A **decision stump** is a tree with only one node. It takes the dataset and cuts it with a single slice that is aligned with one of the features.



Obviously, these trees are extremely fast to train and even faster to evaluate. To make a prediction with a decision stump, all you have to do is check whether the value of a single feature is above or below the threshold.

It is very easy to construct a decision stump that qualifies as a weak classifier, meaning that it achieves greater than 50% accuracy on the training data. Why is this true? Well, if you choose any decision stump at random, meaning you pick a random feature and a random threshold and you get less than 50% accuracy, then all you have to do is invert the inequality, and now you have a stump with greater than 50% accuracy. So decision stumps are good candidates for building boosting algorithms.

The algorithm for AdaBoost begins by setting the iteration counter, s, to 0 and assigning an equal weight, w_s^i, to every sample i in the dataset. This weight represents how much effort the stump, s, should put into correctly classifying data point i. And to begin with, all samples are weighted equally.

$$s = 0$$

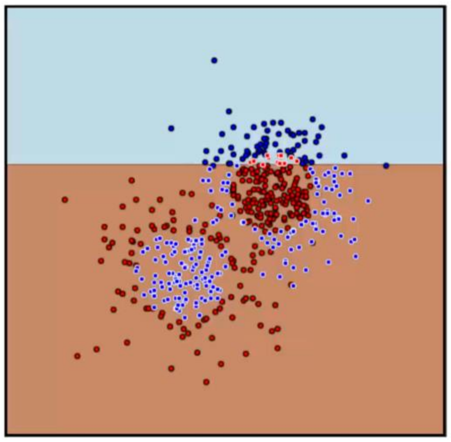
$$w_s^i = \frac{1}{N} \text{ for all samples}$$

$$i = 1 \dots N$$

Next, you create a stump that can correctly classify samples, accounting for at least half of the total weight.

$$S_s.fit(x_1 y_1, W_s)$$

Again, this is not difficult to do. If you do not succeed, you can simply flip the inequality.



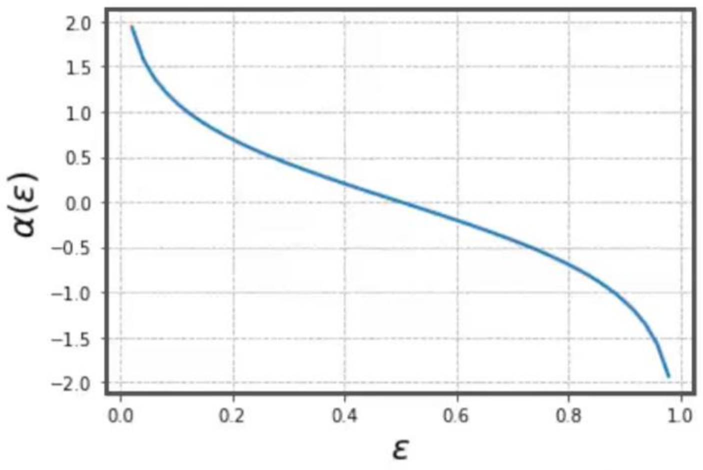
In this picture, you can see that this stump correctly classifies most of the red points, but incorrectly classifies a large clump of blue points in the middle of the lower cluster. You then compute the total weight of the samples that were misclassified by the stump. This is the stump misclassification score.

$$\in_s = \sum_{misclass} W_s^i$$

This number is expected to be less than 0.5, since the stump is capable of better than random classification. The misclassification score is now used to determine the influence of this particular stump in the overall ensemble. The larger the stump's misclassification score, the smaller its influence. You denote the influence coefficient with an α_s.

$$\alpha_s = \frac{1}{2} \log \text{ of } \frac{1-\in_s}{\in_s}$$

The plot for this formula drops from plus infinity to minus infinity, as ∈ goes from 0 to 1. And it crosses 0 at ∈ = 0.5.



Since all of your stumps are required to have misclassification rates less than 0.5, their α-values will always be positive.

The influence parameter α is then used to update the weights for each of the samples. If a sample was misclassified, then its weight is multiplied by e to the α_s, which is larger than 1, since α_s is positive. If the sample was correctly classified, then its weight is divided by e to the α_s, which causes it to decrease.

$$w_{s+1}^i = \begin{cases} w_s^i e^{\alpha_s} \\ w_s^i e^{-\alpha_s} \end{cases}$$

The weights are then normalized by dividing them by their sum.

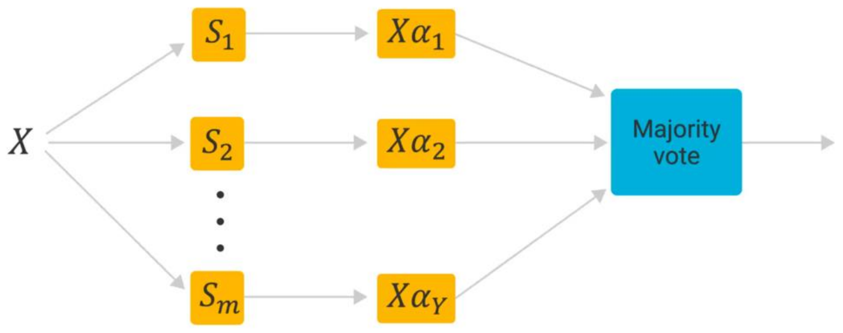
$$w_{s+1}^i = \frac{w_{s+1}^i}{\sum_i w_{s+1}^i}$$

This ensures that they continue to add up to 1.

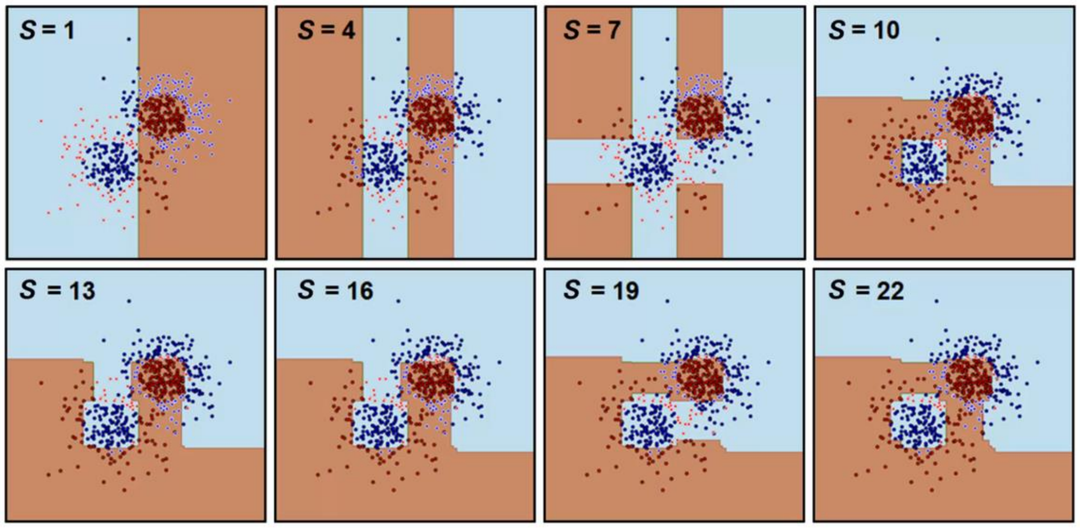
$$S \leftarrow s + 1$$

The counter is then incremented and you repeat the loop as many times as you wish.

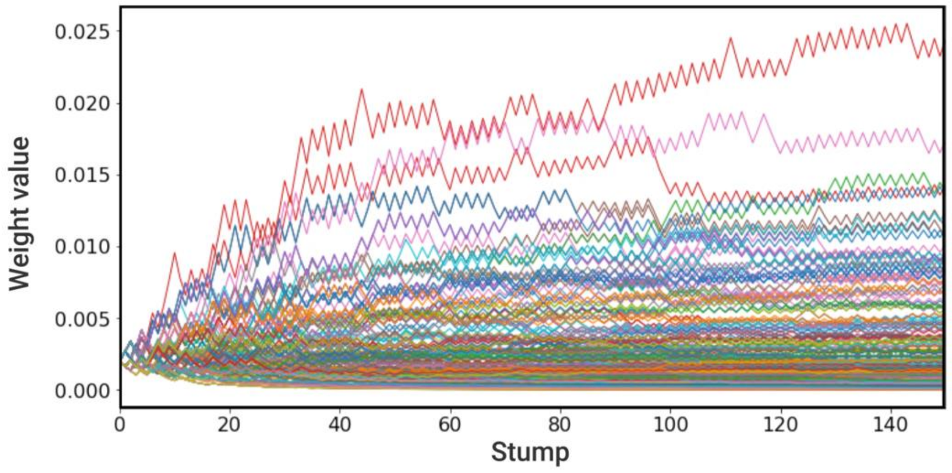
To make a prediction with AdaBoost, you simply collect all of the predictions from the component classifiers, weigh them by their influence parameters, α_s, and then select the predicted class by a weighted majority vote.



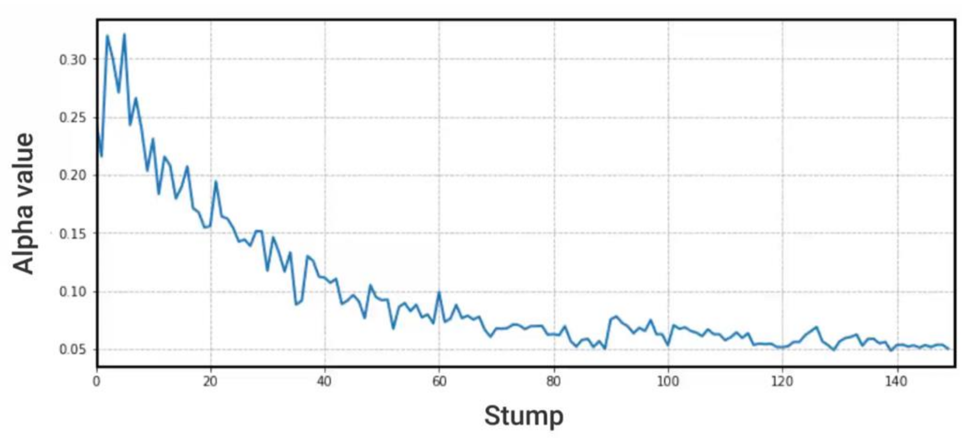
Here you see the evolution of AdaBoost through the first 22 iterations.



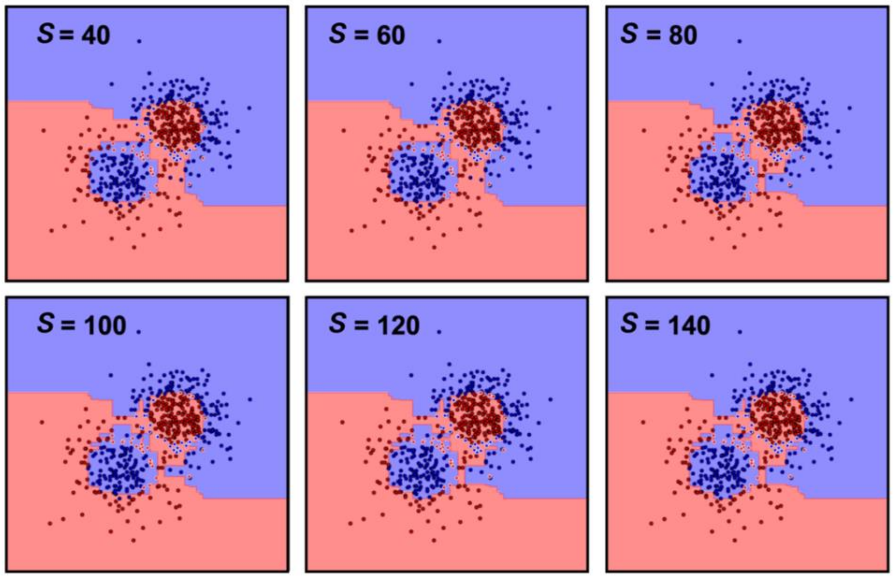
With S = 1, you have only a single stump, which splits the data into two approximately equal parts. With four stumps, the algorithm has created four vertical stripes that correctly classify the centers of the two clusters. After seven iterations, you begin to see horizontal stripes. And after 10, the algorithm begins to isolate the centers. It continues to work at picking off more and more samples. But you can see that it is having a hard time with the region of mixing between the two clusters.



Here you see the evolution of the weights for all of the individual data points over 150 iterations. Whenever a point is correctly classified, its weight decreases and attention has shifted to other points, but then it gets misclassified and the weight jumps back up. The result is a jagged switching evolution of the weights, which makes it a bit challenging to decide when to stop the algorithm. The predictions of the algorithm do settle down, however, because they depend only on the influence parameter, α_s, and not on the sample weights.



Here you see that α_s becomes small after about 80 iterations. This is a good time to stop the training process.



And here is confirmation that the decision boundaries do indeed settle down. You can see that the red and blue regions stop changing after about 40 iterations. This points to an important property of AdaBoost and of boosting algorithms in general. They are slow learners, meaning that they are not easily overfitted. If you continued to boost this model, you would eventually overfit it. But in contrast with other models, such as decision trees, AdaBoost is very forgiving and it gives you plenty of time to stop the training process before reaching that point.

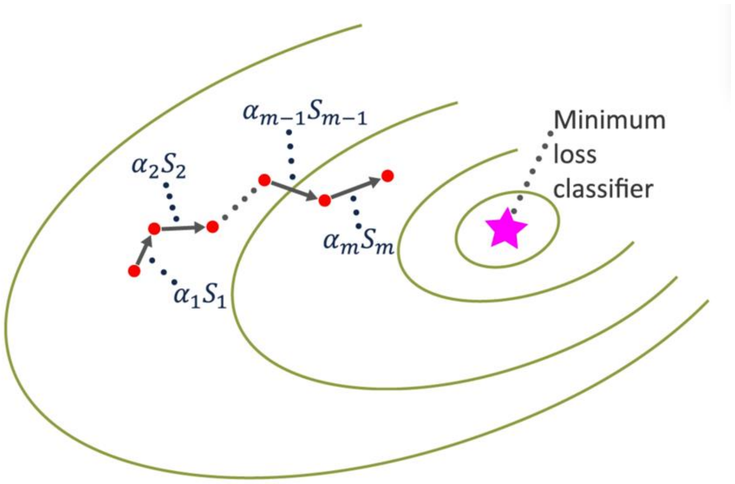
Gradient Boosting Trees

Scikit-learn provides an implementation that makes AdaBoost as easy to use as any other classification algorithm. You simply call the **AdaBoostClassifier** constructor and pass in a base model, which is usually a decision tree with a maximum depth of 1.

```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
model = AdaBoostClassifier(DecisionTreeClassifier(max_depth=1))
model.fit(X, y)
```

When you described the AdaBoost algorithm, you were probably asking yourself where these particular formulas came from. Why is the influence parameter equal to 1/2 the logarithm of 1 minus ∈ over ∈? Why are the weights multiplied by e to the α? As it turns out, AdaBoost is a particular example of a larger class of boosting algorithms called **gradient boosting**.

The gradient boosting setup defines a loss function at the bottom of which is an ideal classifier.



The algorithm then applies gradient descent to build up a model that with every step gets closer and closer to this ideal classifier.

Gradient descent:

$$X_n = \sum_{i=1}^{n-1} \gamma_i(-\nabla f(x_i))$$

At every step, the gradient boosting algorithm creates a weak model that points in the approximate direction of the ideal classifier. Because the weak model is better than random guessing, you are assured that it will point you in the general direction of the target. It will never point away from the target.

Gradient boosting (AdaBoost):

$$\text{Ensemble model} = \sum_{i=1}^{s} \alpha_s \begin{pmatrix} weak \\ learner \end{pmatrix}$$

The algorithm then takes a step in the direction indicated by the weak model, and then repeats the process. It creates another weak model, takes another step, and so forth. Provided the loss function is convex, this procedure guarantees that you will eventually get very close to the target.

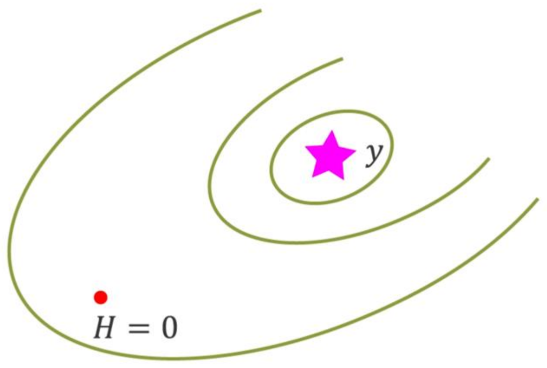
AdaBoost is an implementation of this idea, with a particular loss function, and an adaptive step size. The loss function is the exponential risk. And you should know that this particular cost function is what leads to the weight updating rule. In the gradient descent view of AdaBoost, the influence parameters, α, are actually the step sizes. The formula for α was optimally chosen to find the minimum of this particular loss function as quickly as possible.

Gradient boosting refers to the general idea of applying gradient descent to the problem of boosting. **Gradient boosting trees** is another particular type of gradient boosting.

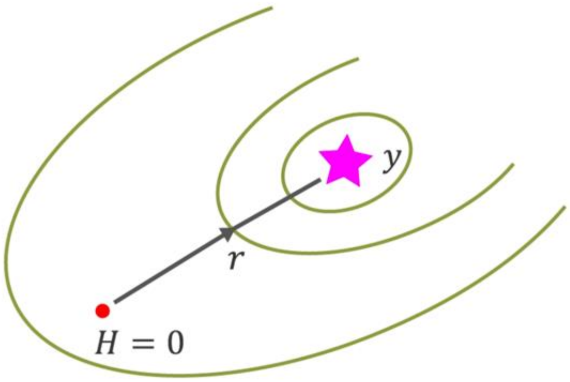
Gradient-boosted trees:

- Uses trees for its base model – either decision trees or regression trees
- Uses the squared loss as its cost function and cannot be implemented as a simple weight updating scheme

The algorithm begins by initializing the boosting model to 0. This means that it predicts 0 for all of the samples in the dataset.



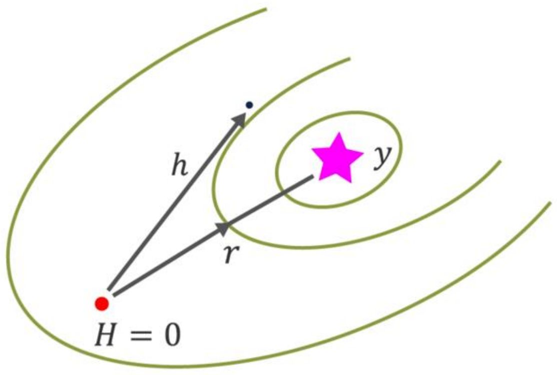
Your goal is to build up H so that it predicts the correct labels, y, for all of the data. If you achieve this, then the red dot, H, will coincide with the magenta star, y. You then compute the difference between the desired outputs and the predicted outputs.



In the picture, this is an arrow that points from the model, H, toward the target, y. It is the direction in which you would like to advance. You can call this direction, r.

$$r_i = y_i - H(x_i)$$

Next you train a weak model, h, on labels, r. In other words, you train a model that takes the feature data as input and is trained to produce the residuals, r.

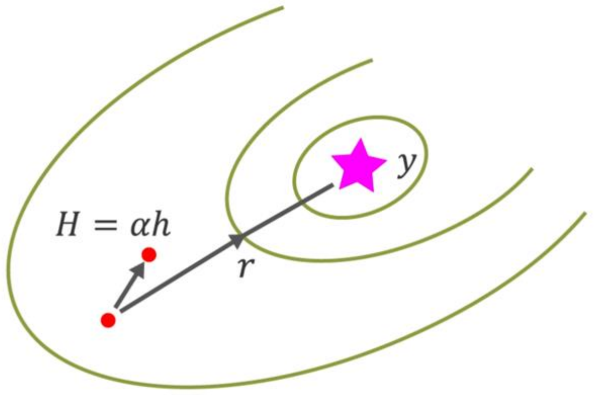


Here h is a shallow regression tree, typically no deeper than four nodes.

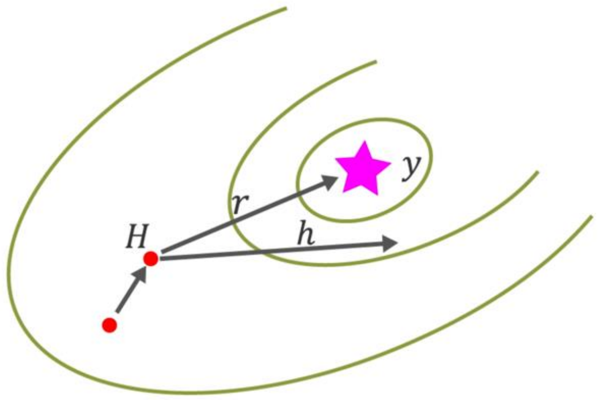
$$h(x_i) \cong r(x_i)$$

It will not be perfect. So it will not reproduce r exactly. But you trust it enough to take a small step in that direction. You do this by adding to H a small coefficient, α, times the output of the weak learner.

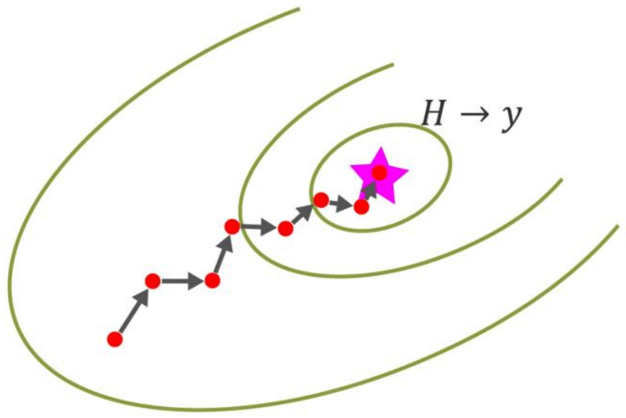
$$H \leftarrow H + \alpha h$$



Then you repeat, and again compute the desired direction, r, using your updated model, train a new regression tree, take a new step, and so on.



Until eventually you get very close to the target.



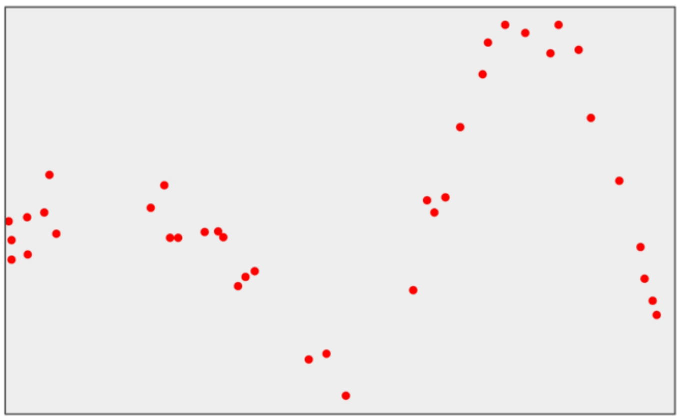
When this happens, you will have a strong learner, H, composed of a weighted sum of a bunch of weak learners, h, with weights, α.

$$H(x) = \sum_i \alpha_i h_i(x)$$

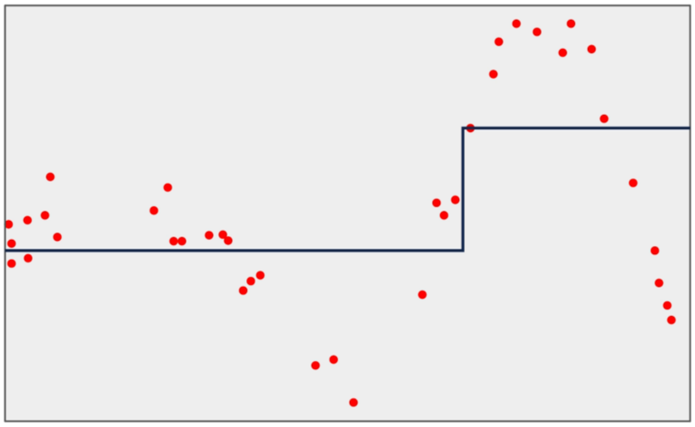
In scikit-learn, gradient-boosted regression trees is implemented in the GradientBoostingRegressor class.

from sklearn.ensemble import GradientBoostingRegressor
GradientBoostingRegressor(n_estimators=1000, max_depth=1)

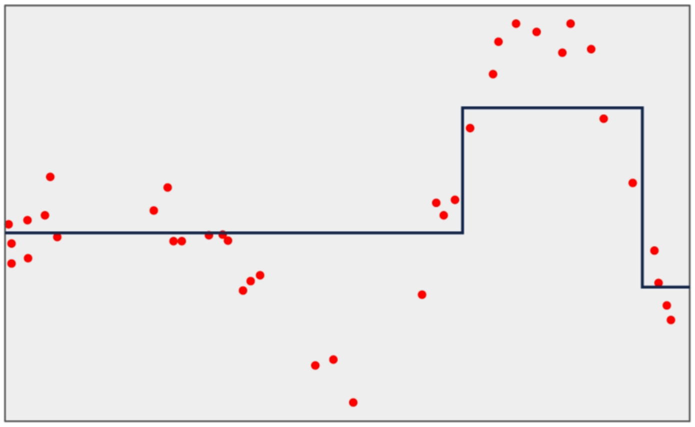
You are going to create a one-dimensional regression model to approximate this series of red points.



In the first iteration, the algorithm creates a regression stump that approximates the points with a step function.

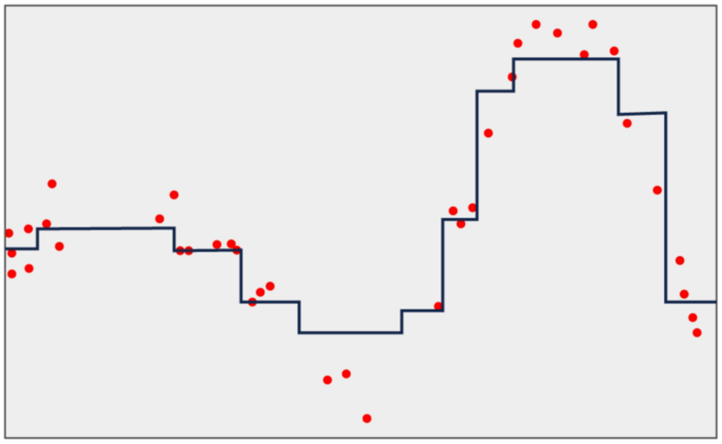


The residuals of this model are then used to fit a second regression stump, which is added to the first.

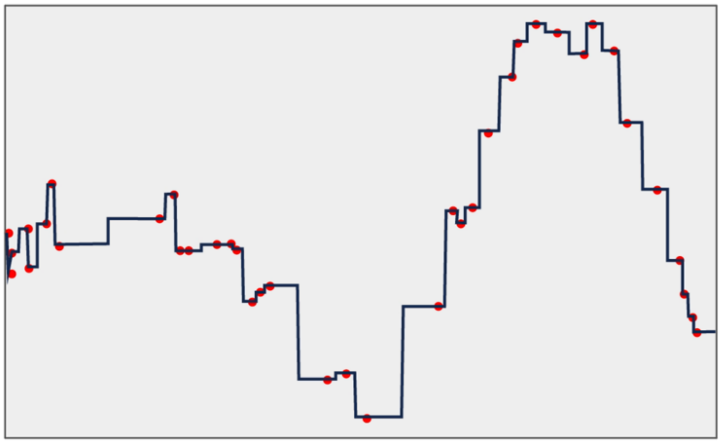


It seems that the second tree chose to correct large errors near the right edge of the plot. Then you repeat to find a third tree, a fourth, and so on.

And this is what the model looks like after 10 iterations.



This may be good enough for your purposes. The complete model is still pretty simple, since it consists only of ten stumps. But you can also go on to 50 stumps, or to 100. And after 500 iterations, you have pretty much converged to a model that fits the data exactly.



H has reached the magenta star. But what if you kept on going? Well, after 1,000 iterations, the model looks pretty much the same. So the algorithm is pretty robust in the number of iterations. These extra 500 regression stumps are probably not worth the added computational burden.

As you can tell, boosting algorithms have some very nice properties.

The boosting algorithm:

- Reduces the bias of weak learners
- Does not increase the variance as much as other algorithms

Ensemble methods, in general, have an excellent track record in the world of machine learning. You should always consider an ensemble method for your most challenging machine learning tasks.