Machine Learning

(IV CSE – I SEM.)

A.Y.: 2022 – 2023

UNIT-V

Ensemble Learning

Ensemble Learning Model Combination Schemes, Voting, Error-Correcting Output Codes, Bagging: RandomForest Trees, Boosting: Adaboost, Stacking.

Ensemble Learning

Ensemble learning usually produces more accurate solutions than a single model would. Ensemble Learning is a technique that create multiple models and then combine them them to produce improved results. Ensemble learning usually produces more accurate solutions than a single model would.

- Ensemble learning methods is applied to regression as well as classification.
    - Ensemble learning for regression creates multiple repressors i.e. multiple regressionmodels such as linear, polynomial, etc.
    - Ensemble learning for classification creates multiple classifiers i.e. multiple classificationmodels such as logistic, decision tress, KNN, SVM, etc.

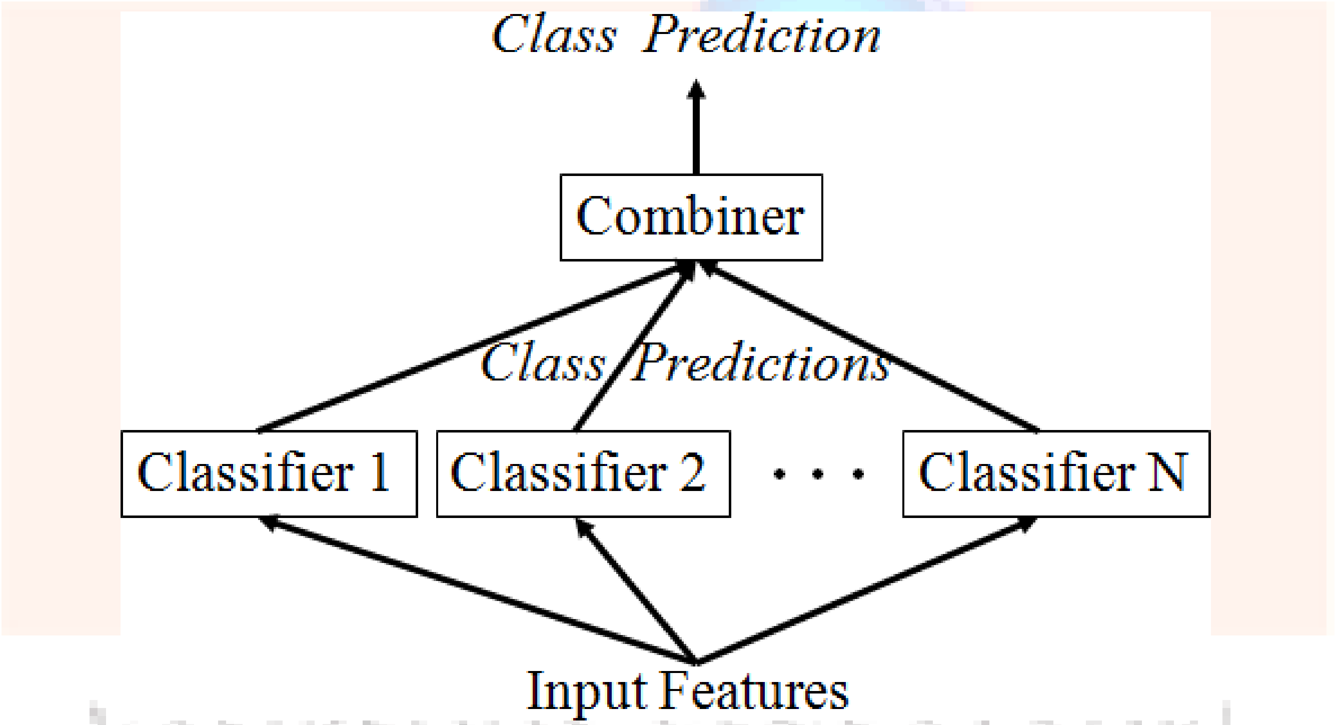


Figure 1: Ensemble learning view

Which components to combine?

- different learning algorithms
- same learning algorithm trained in different ways
- same learning algorithm trained the same way

There are two steps in ensemble learning:

Multiples machine learning models were generated using same or different machine learningalgorithm. These are called "base models". The prediction perform on the basis of base models.

Techniques/Methods in ensemble learning

Voting, Error-Correcting Output Codes, Bagging: Random Forest Trees, Boosting: Adaboost, Stacking.

Model Combination Schemes - Combining Multiple Learners

We discussed many different learning algorithms in the previous chapters. Though these are generally successful, no one single algorithm is always the most accurate. Now, we are going to discuss models composed of multiple learners that complement each other so that by combining them, we attain higher accuracy.

There are also different ways the multiple base-learners are combined to generate the final output:

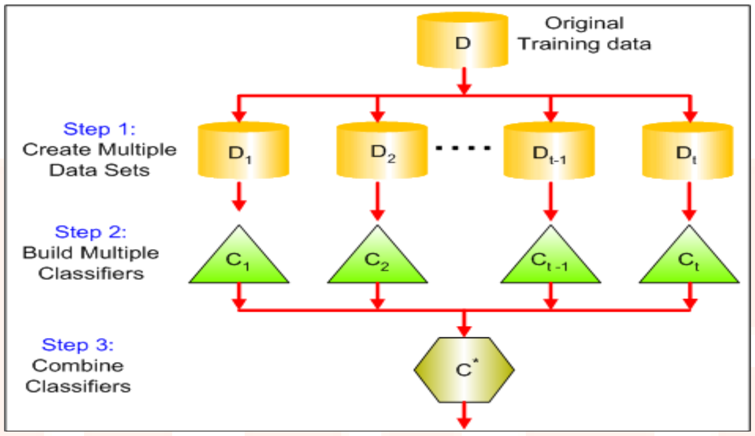

D
Original Training data
Step 1: Create Multiple Data Sets
D1
D2
Dt-1
Dt
Step 2: Build Multiple Classifiers
C1
C2
Ct-1
Ct
Step 3: Combine Classifiers
C*

Figure2: General Idea - Combining Multiple Learners

Multiexpert combination

Multiexpert combination methods have base-learners that work in parallel. These methods can in turn be divided into two:

- In the global approach, also called learner fusion, given an input, all base-learners generate an output and all these outputs are used.
  Examples are voting and stacking.
- In the local approach, or learner selection, for example, in mixture of experts, there is a gating model, which looks at the input and chooses one (or very few) of the learners as responsible for generating the output.

Multistage combination

Multistage combination methods use a *serial* approach where the next base-learner is trained with or tested on only the instances where the previous base-learners are not accurate enough. Theidea is that the base-learners (or the different representations they use) are sorted in increasing complexity so that a complex base-learner is not used (or its complex representation is not extracted) unless the preceding simpler base-learners are not confident.
An example is *cascading*.

Let us say that we have *L* base-learners. We denote by *dj(x)* the prediction of base-learner M_j given the arbitrary dimensional input *x*. In the case of multiple representations, each M_j uses a different input representation x_j . The final prediction is calculated from the predictions of the base-learners:

$y = f(d_1, d_2, \ldots, d_L | \Phi)$

where $f(\cdot)$ is the combining function with Φ denoting its parameters.

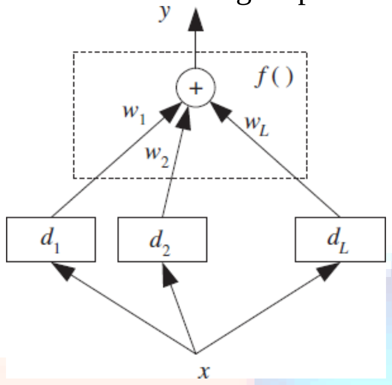


Figure 1: Base-learners are *dj* and their outputs are combined using *f (·)*.

This is for a single output; in the case of classification, each base-learner has *K* outputs that are separately used to calculate y_i, and then we choose the maximum. Note that here all learners observe the same input; it may be the case that different learners observe different representations of the same input object or event.

When there are *K* outputs, for each learner there are $d_{ji}(x)$, $i = 1, \ldots, K$,
$j = 1, \ldots, L$, and, combining them, we also generate *K* values, y_i, $i = 1, \ldots, K$ and then for example inclassification, we choose the class with
the maximum *yi* value:

$$\text{Choose } C_i \text{ if } y_i = \max_{k-1}^{K} y_k$$

Voting

The simplest way to combine multiple classifiers is by *voting*, which corresponds to taking alinear combination of the learn

ers, Refer figure 1.

$$y_i = \sum_j w_j d_{ji} \text{ where } w_j \geq 0, \sum_j w_j = 1$$

This is also known as *ensembles* and *linear opinion pools*. In the sim plest case, all learners are given equal weight and we have *simple voting* that corresponds to taking an average. Still, taking a (weighted) sum is only one of the possibilities and there are also other combination rules, as shown in table 1. If the outputs are not posterior probabilities, these rules require that outputs be normalized to the same scale

Table 1 - Classifier combination rules

Rule	Fusion function $f(\cdot)$
Sum	$y_i = \frac{1}{L}\sum_{j=1}^{L} d_{ji}$
Weighted sum	$y_i = \sum_j w_j d_{ji}, w_j \geq 0, \sum_j w_j = 1$
Median	$y_i = \text{median}_j d_{ji}$
Minimum	$y_i = \min_j d_{ji}$
Maximum	$y_i = \max_j d_{ji}$
Product	$y_i = \prod_j d_{ji}$

An example of the use of these rules is shown in table 2, which demonstrates the effects of different rules. Sum rule is the most intuitive and is the most widely used in practice. Median rule is more robust to outliers; minimum and maximum rules are pessimistic and optimistic, respectively. With the product rule, each learner has veto power; regardless of the other ones, if one learner has an output of 0, the overall output goes to 0. Note that after the combination rules, *yi* do not necessarily sum up to 1.

Table 2: Example of combination rules on three learners and three classes

	C_1	C_2	C_3
d_1	0.2	0.5	0.3
d_2	0.0	0.6	0.4
d_3	0.4	0.4	0.2
Sum	0.2	**0.5**	0.3
Median	0.2	**0.5**	0.4
Minimum	0.0	**0.4**	0.2
Maximum	0.4	**0.6**	0.4
Product	0.0	**0.12**	0.032

In weighted sum, d_{ji} is the vote of learner j for class C_i and w_j is the weight of its vote. Simple voting is a special case where all voters have equal weight, namely, $wj = 1/L$. In classification, this is called *plurality voting* where the class having the maximum number of votes is the winner.

When there are two classes, this is *majority voting* where the winning class gets more than half of the votes. If the voters can also supply the additional information of how much they vote for each class (e.g., by the posterior probability), then after normalization, these can be used as weights in a *weighted voting* scheme. Equivalently, if d_{ji} are the class posterior probabilities, $P(C_i \mid x, M_j)$, then we can just sum them up ($w_j = 1/L$) and choose the class with maximum y_i .

In the case of regression, simple or weighted averaging or median can be used to fuse the outputs of base-regressors. Median is more robust to noise than the average.

Another possible way to find w_j is to assess the accuracies of the learners (regressor or classifier) on a separate validation set and use that information to compute the weights, so that we give more weights to more accurate learners.

Voting schemes can be seen as approximations under a Bayesian framework with weights approximating prior model probabilities, and model decisions approximating model-conditional likelihoods.

$$P(C_i|x) = \sum_{\text{all models } \mathcal{M}_j} P(C_i|x, \mathcal{M}_j)P(\mathcal{M}_j)$$

Simple voting corresponds to a uniform prior. If we have a prior distribution preferring simpler models, this would give larger weights to them. We cannot integrate over all models; we only choose a subset for which we believe *P(*M_j *)* is high, or we can have another Bayesian step and calculate P(C_i | x,M_j), the probability of a model given the sample, and sample high probable models from this density.

Let us assume that d_j are iid with expected value *E[*d_j*]* and variance Var*(*d_j *)*, then when we take a simple average with w_j = 1/*L*, the expected value and variance of the output are

$$E[y] = E\left[\sum_j \frac{1}{L}d_j\right] = \frac{1}{L}LE[d_j] = E[d_j]$$

$$\text{Var}(y) = \text{Var}\left(\sum_j \frac{1}{L}d_j\right) = \frac{1}{L^2}\text{Var}\left(\sum_j d_j\right) = \frac{1}{L^2}L\text{Var}(d_j) = \frac{1}{L}\text{Var}(d_j)$$

We see that the expected value does not change, so the bias does not change. But variance, and therefore mean square error, decreases as the number of independent voters, *L*, increases. In the general case,

$$\text{Var}(y) = \frac{1}{L^2}\text{Var}\left(\sum_j d_j\right) = \frac{1}{L^2}\left[\sum_j \text{Var}(d_j) + 2\sum_j\sum_{i<j}\text{Cov}(d_j, d_i)\right]$$

which implies that if learners are positively correlated, variance (and error) increase. We can thus view using different algorithms and input features as efforts to decrease, if not completely eliminate, the positive correlation.

Error-Correcting Output Codes

The **Error-Correcting Output Codes** method is a technique that allows a multi-class classificationproblem to be reframed as multiple binary classification problems, allowing the use of native binary classification models to be used directly.

Unlike one-vs-rest and one-vs-one methods that offer a similar solution by dividing a multi-class classification problem into a fixed number of binary classification problems, the error-correcting output codes technique allows each class to be encoded as an arbitrary number of binary classification problems. When an overdetermined representation is used, it allows the extra models to act as "error- correction" predictions that can result in better predictive performance.

In *error-correcting output codes* (ECOC), the main classification task is defined in terms of a number of subtasks that are implemented by the base-learners. The idea is that the original task of separating one class from all other classes may be a difficult problem. Instead, we want to define a setof simpler classification problems, each specializing in one aspect of the task, and combining these simpler classifiers, we get the final classifier.

Base-learners are binary classifiers having output $-1/+1$, and there is a *code matrix* **W** of $K \times L$ whose K rows are the binary codes of classes in terms of the L base-learners d_j. For example, if the second row of **W** is $[-1,+1,+1,-1]$, this means that for us to say an instance belongs to C_2, the instance should be on the negative side of d_1 and d_4, and on the positive side of d_2 and d_3. Similarly, the columns of the code matrix defines the task of the base-learners. For example, if the third column is $[-1,+1,+1]T$, we understand that the task of the third base-learner, d_3, is to separate the instances of C_1 from the instances of C_2 and C_3 combined. This is how we form the training set of the base-learners.
For example,

in this case, all instances labeled with C2 and C3 form $X_3^{\pm}$ and instances labeled with C_1 form X_3^{-}, and d_3 is
trained so that $x^t \in X_3^{\pm}$ give output +1 and $x^t \in X_3^{-}$ give output −1.

The code matrix thus allows us to define a polychotomy ($K > 2$ classification problem) in terms of dichotomies ($K = 2$ classification problem), and it is a method that is applicable using any learning algorithm to implement the dichotomizer base-learners—for example, linear or multilayer perceptrons (with a single output), decision trees, or SVMs whose original definition is for two-class problems.
The typical one discriminant per class setting corresponds to the diagonal code matrix where $L = K$. For example, for $K = 4$,

we have

$$\mathbf{W} = \begin{bmatrix} +1 & -1 & -1 & -1 \\ -1 & +1 & -1 & -1 \\ -1 & -1 & +1 & -1 \\ -1 & -1 & -1 & +1 \end{bmatrix}$$

The problem here is that if there is an error with one of the baselearners, there may be a misclassification because the class code words are so similar. So the approach in error-correcting codes is to have $L > K$ and increase the Hamming distance between the code words. One possibility is *pairwise separation* of classes where there is a separate baselearner to separate Ci from Cj, for $i < j$. In this case, L
$= K(K-1)/2$ and with $K = 4$, the code matrix is

$$\mathbf{W} = \begin{bmatrix} +1 & +1 & +1 & 0 & 0 & 0 \\ -1 & 0 & 0 & +1 & +1 & 0 \\ 0 & -1 & 0 & -1 & 0 & +1 \\ 0 & 0 & -1 & 0 & -1 & -1 \end{bmatrix}$$

where a 0 entry denotes “don’t care.” That is, d_1 is trained to separate C_1 from C_2 and does not use the training instances belonging to the other classes. Similarly, we say that an instance

belongs to C_2 if d_1 =
-1 and $d_4 = d_5 = +1$, and we do not consider the values of d_2, d_3, and d_6. The problem here is that L is $O(K^2)$, and for large K pairwise separation may not be feasible.

If we can have L high, we can just randomly generate the code matrix with $-1/+1$ and this will work fine, but if we want to keep L low, we need to optimize **W**. The approach is to set L beforehand and then find **W** such that the distances between rows, and at the same time the distances between columns, are as large as possible, in terms of Hamming distance. With K classes, there are $2^{(K-1)} - 1$ possible columns, namely, two-class problems. This is because K bits can be written in $2K$ different ways and complements (e.g., “0101” and “1010,” from our point of view, define the same discriminant) dividing the possible combinations by 2 and then subtracting 1 because a column of all 0s (or 1s) is useless. For example, when $K = 4$, we have

$$\mathbf{W} = \begin{bmatrix} -1 & -1 & -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & +1 & +1 & +1 & +1 \\ -1 & +1 & +1 & -1 & -1 & +1 & +1 \\ +1 & -1 & +1 & -1 & +1 & -1 & +1 \end{bmatrix}$$

When K is large, for a given value of L, we look for L columns out of the $2^{(K-1)}-1$. We would like these columns of **W** to be as different as possible so that the tasks to be learned by the base-learners are as different from each other as possible. At the same time, we would like the rows of **W** to be as different as possible so that we can have maximum error correction in case one or more base-learners fail.

ECOC can be written as a voting scheme where the entries of **W**, w_{ij} , are considered as vote weights:

$$y_i = \sum_{j=1}^{L} w_{ij} d_j$$

and then we choose the class with the highest y_i . Taking a weighted sum and then choosing the maximum instead of checking for an exact match allows d_j to no longer need to be binary but to take a value between -1 and $+1$, carrying soft certainties instead of hard decisions. Note that a value p_j between 0 and 1, for example, a posterior probability, can be converted to a value d_j between -1 and $+1$ simply as

$$d_j = 2p_j - 1$$

One problem with ECOC is that because the code matrix **W** is set a priori, there is no guarantee that the subtasks as defined by the columns of **W** will be simple.

Bagging

Bootstrap aggregating, often abbreviated as bagging, involves having each model in the ensemble vote with equal weight. In order to promote model variance, bagging trains each model in the ensemble using a randomly drawn subset of the training set. As an example, the random forest algorithm combines random decision trees with bagging to achieve very high classification accuracy.

The simplest method of combining classifiers is known as bagging, which stands for bootstrap aggregating, the statistical description of the method. This is fine if you know what a bootstrap is, but fairly useless if you don't. A bootstrap sample is a sample taken from the original dataset with replacement, so that we may get some data several times and others not at all. The bootstrap sample is the same size as the original, and lots and lots of these samples are taken: *B* of them, where *B* is at least 50, and could even be in the thousands. The name bootstrap is more popular in computer science than anywhere else, since there is also a bootstrap loader, which is the first program to run when a computer is turned on. It comes from the nonsensical idea of 'picking yourself up by your bootstraps,' which means lifting yourself up by your shoelaces, and is meant to imply starting from nothing.

Bootstrap sampling seems like a very strange thing to do. We've taken a perfectly good dataset, mucked it up by sampling from it, which might be good if we had made a smaller dataset (since it would be faster), but we still ended up with a dataset the same size. Worse, we've done it lots of times. Surely this is just a way to burn up computer time without gaining anything. The benefit of it is that we will get lots of learners that perform slightly differently, which is exactly what we want for an ensemble method. Another benefit is that estimates of the accuracy of the classification function can be made without complicated analytic work, by throwing computer resources at the problem (technically, bagging is a variance reducing algorithm; the meaning of this will become clearer when we talk about bias and variance). Having taken a set of bootstrap samples, the bagging method simply requires that we fit a model to each dataset, and then combine them by taking the output to be the majority vote of all the classifiers. A NumPy implementation is shown next, and then we will look at a simple example.

```
# Compute bootstrap samples
samplePoints =
np.random.randint(0,nPoints,(nPoints,nSamples))classifiers
= []

for i in range(nSamples):
        sample = []
        sampleTarget =
[]for j in
range(nPoints):
        sample.append(data[samplePoints[j,i]])
        sampleTarget.append(targets[samplePoints[j,i]])
# Train classifiers
classifiers.append(self.tree.make_tree(sample,sampleTarget,features))
```

The example consists of taking the party data that was used to demonstrate the decision tree, and restricting the trees to stumps, so that they can make a classification based on just one variable

When we want to construct the decision tree to decide what to do in the evening, we start by listing everything that we've done for the past few days to get a suitable dataset (here, the last ten days):

Deadline?	Is there a party?	Lazy?	Activity
Urgent	Yes	Yes	Party
Urgent	No	Yes	Study
Near	Yes	Yes	Party
None	Yes	No	Party
None	No	Yes	Pub
None	Yes	No	Party
Near	No	No	Study
Near	No	Yes	TV
Near	Yes	Yes	Party
Urgent	No	No	Study

The output of a decision tree that uses the whole dataset for this is not surprising: it takes the two largest classes, and separates them. However, using just stumps of trees and 20 samples, bagging can separate the data perfectly, as this output shows:

```
Tree Stump Prediction
['Party', 'Party', 'Party', 'Party', 'Pub', 'Party', 'Study', 'Study', ↩
'Party', 'Study']
Correct Classes
['Party', 'Study', 'Party', 'Party', 'Pub', 'Party', 'Study', 'TV', 'Party', ↩
'Study']
Bagged Results
['Party', 'Study', 'Party', 'Party', 'Pub', 'Party', 'Study', 'TV', 'Party', ↩
'Study']
```

RANDOM FORESTS

A random forest is an ensemble learning method where multiple decision trees are constructed and then they are merged to get a more accurate prediction.

If there is one method in machine learning that has grown in popularity over the last few years, then it is the idea of random forests. The concept has been around for longer than that, with several different people inventing variations, but the name that is most strongly attached to it is that of Breiman, who also described the CART algorithm in unit 2.

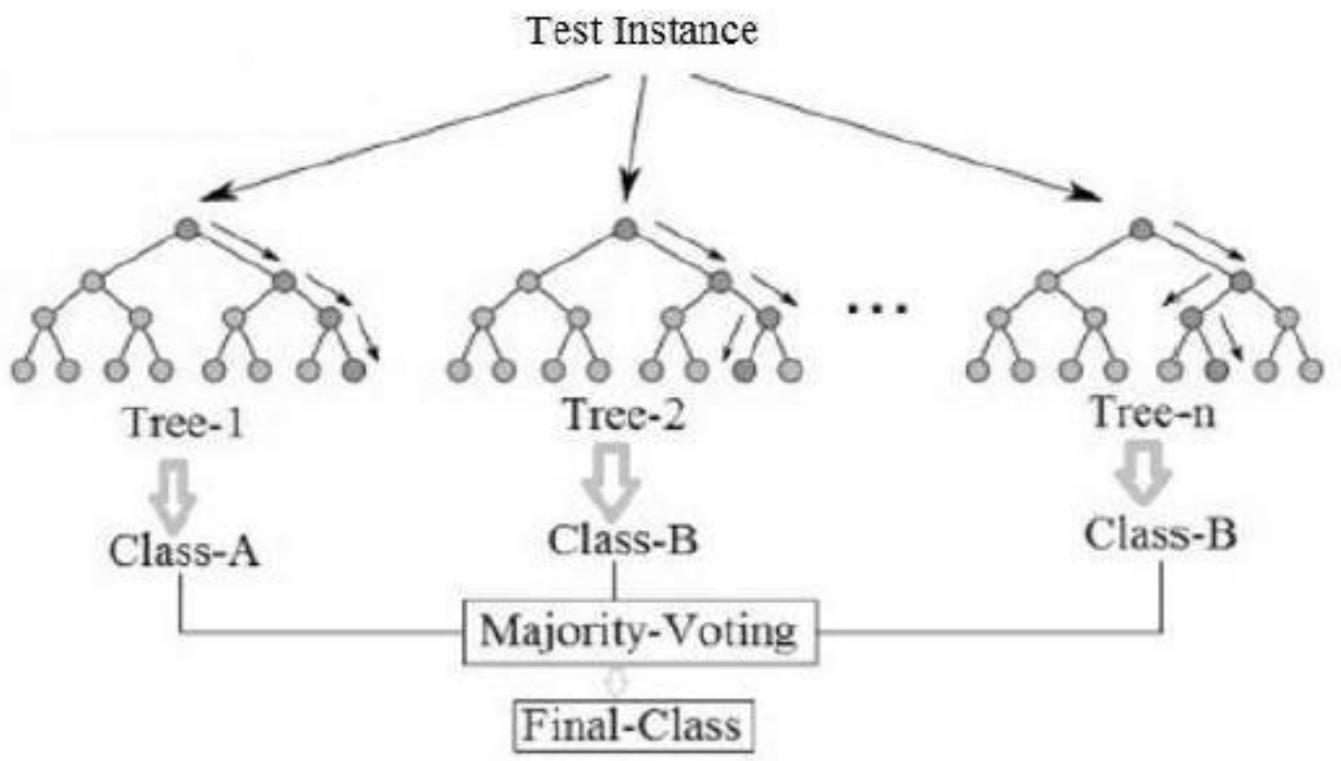


Figure 3: Example of random forest with majority voting

The idea is largely that if one tree is good, then many trees (a forest) should be better, provided that there is enough variety between them. The most interesting thing about a random forest is the ways that it creates randomness from a standard dataset. The first of the methods that it uses is the one that we have just seen: bagging. If we wish to create a forest then we can make the trees different by training them on slightly different data, so we take bootstrap samples from the dataset for each tree. However, this isn't enough randomness yet. The other obvious place where it is possible to add randomness is to limit the choices that the decision tree can make. At each node, a random subset of the features is given to the tree, and it can only pick from that subset rather than from the whole set.

As well as increasing the randomness in the training of each tree, it also speeds up the training, since there are fewer features to search over at each stage. Of course, it does introduce a new parameter (how many features to consider), but the random forest does not seem to be very sensitive to this parameter; in practice, a subset size that is the square root of the number of features seems to be common. The effect of these two forms of randomness is to reduce the variance without effecting the bias. Another benefit of this is that there is no need to prune the trees. There is another parameter that we don't know how to choose yet, which is the number of trees to put into the forest. However, this is fairly easy to pick if we want optimal results: we can keep on building trees until the error stops decreasing.

Once the set of trees are trained, the output of the forest is the majority vote for classification, as with the other committee methods that we have seen, or the mean response for regression. And those are pretty much the main features needed for creating a random forest. The algorithm is given next before we see some results of using the random forest.

Algorithm

Here is an outline of the random forest algorithm.

1. The random forests algorithm generates many classification trees. Each tree is generated as follows:
   a) If the number of examples in the training set is N, take a sample of N examples at random - but with replacement, from the original data. This sample will be the training set for generating the tree.
   b) If there are M input variables, a number m is specified such that at each node, m variables are selected at random out of the M and the best split on these m is used to split the node. The value of m is held constant during the generation of the various trees in the forest.
   c) Each tree is grown to the largest extent possible.
2. To classify a new object from an input vector, put the input vector down each of the trees in the forest. Each tree gives a classification, and we say the tree "votes" for that class. The forest chooses the classification

The implementation of this is very easy: we modify the decision to take an extra parameter, which is *m*, the number of features that should be used in the selection set at each stage. We will look at an example of using it shortly as a comparison to boosting.

Looking at the algorithm you might be able to see that it is a very unusual machine learning method because it is embarrassingly parallel: since the trees do not depend upon each other, you can both create and get decisions from different trees on different individual processors if you have them. This means that the random forest can run on as many processors as you have available with nearly linear speedup.

There is one more nice thing to mention about random forests, which is that with a little bit of programming effort they come with built-in test data: the bootstrap sample will miss out about 35% of the data on average, the so-called out-of-bootstrap examples. If we keep track of these datapoints then they can be used as novel samples for that particular tree, giving an estimated test error that we get without having to use any extra datapoints.

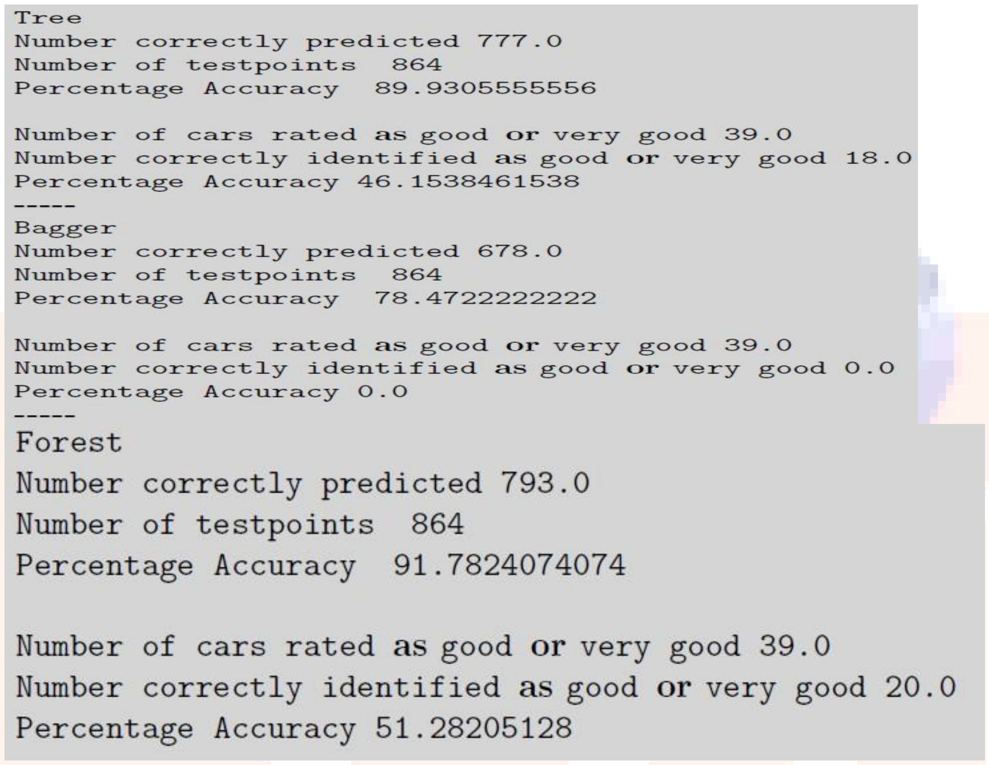
This avoids the need for cross-validation.

As a brief example of using the random forest, we start by demonstrating that the random forest gets the correct results on the Party example that has been used in both this and the previous chapters, based on 10 trees, each trained on 7 samples, and with just two levels allowed in each tree:

```
RF prediction
['Party', 'Study', 'Party', 'Party', 'Pub', 'Party', 'Study', 'TV', 'Party', ⤸
'Study']
```

As a rather more involved example, the car evaluation dataset in the UCI Repository contains

1,728 examples aiming to classify whether or not a car is a good purchase based on six attributes. The following results compare a single decision tree, bagging, and a random forest with 50 trees, each based on 100 samples, and with a maximum depth of five for each tree. It can be seen that the random forest is the most accurate of the three methods.



```
Tree
Number correctly predicted 777.0
Number of testpoints  864
Percentage Accuracy  89.9305555556

Number of cars rated as good or very good 39.0
Number correctly identified as good or very good 18.0
Percentage Accuracy 46.1538461538
-----
Bagger
Number correctly predicted 678.0
Number of testpoints  864
Percentage Accuracy  78.4722222222

Number of cars rated as good or very good 39.0
Number correctly identified as good or very good 0.0
Percentage Accuracy 0.0
-----
Forest
Number correctly predicted 793.0
Number of testpoints  864
Percentage Accuracy  91.7824074074

Number of cars rated as good or very good 39.0
Number correctly identified as good or very good 20.0
Percentage Accuracy 51.28205128
```

Strengths and weaknesses

Strengths

The following are some of the important strengths of random forests.

- It runs efficiently on large data bases.
- It can handle thousands of input variables without variable deletion.
- It gives estimates of what variables are important in the classification.
- It has an effective method for estimating missing data and maintains accuracy when a large proportion of the data are missing.
- Generated forests can be saved for future use on other data.
- Prototypes are computed that give information about the relation between the variables and the classification.
- The capabilities of the above can be extended to unlabeled data, leading to unsupervised clustering, data views and outlier detection.
- It offers an experimental method for detecting variable interactions.
- Random forest run times are quite fast, and they are able to deal with unbalanced and missing data.
- They can handle binary features, categorical features, numerical features without any

need forscaling.

Weaknesses

- A weakness of random forest algorithms is that when used for regression they cannot predict beyond the range in the training data, and that they may over-fit data sets that are particularly noisy.
- The sizes of the models created by random forests may be very large. It may take hundreds ofmegabytes of memory and may be slow to evaluate.
- Random forest models are black boxes that are very hard to interpret.

Boosting

- **Boosting: train next learner on mistakes made by previous learner(s)**

In bagging, generating complementary base-learners is left to chance and to the unstability of the learning method. In boosting, we actively try to generate complementary base-learners by training the next learner on the mistakes of the previous learners. The original *boosting* algorithm combines three weak learners to generate a strong learner. A *weak learner* has error probability less than 1/2, which makes it better than random guessing on a two-class problem, and a *strong learner* has arbitrarily small error probability.

Original Boosting Concept

Given a large training set, we randomly divide it into three. We use X1 and train *d*1. We then take X2 and feed it to *d*1. We take all instances misclassified by *d*1 and also as many instances on which *d*1 is correct
from X2, and these together form the training set of *d*2. We then take X3 and feed it to *d*1 and *d*2. The instances on which *d*1 and *d*2 disagree form the training set of *d*3. During testing, given an instance, we give it to *d*1 and *d*2; if they agree, that is the response, otherwise the response of *d*3 is taken as the output.

1. Split data X into {X_1, X_2, X_3}
2. Train d1 on X_1
   - Test d_1 on X_2
3. Train d_2 on d_1's mistakes on X_2 (plus some right)
   - Test d_1 and d_2 on X_3
4. Train d_3 on disagreements between d_1 and d_2

- Testing: apply d_1 and d_2; if disagree, use d_3
- Drawback: need large X

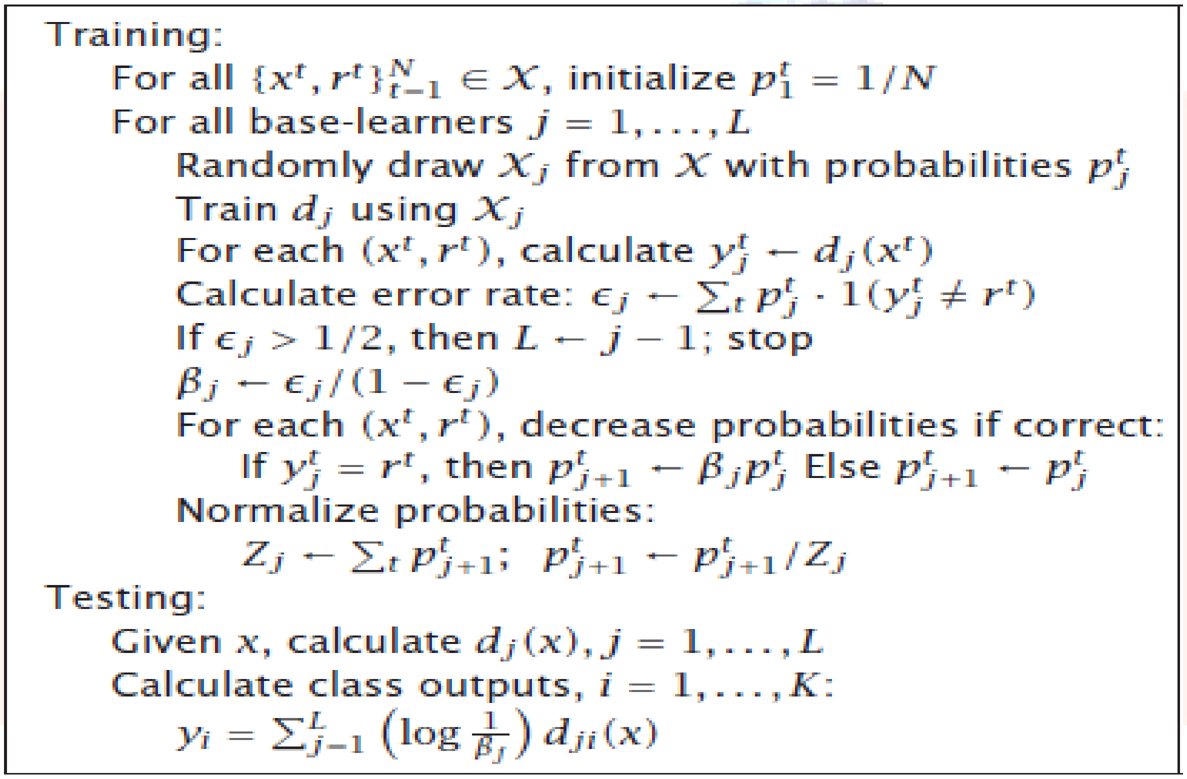
overall system has reduced error rate, and the error rate can arbitrarily be reduced by using such systems recursively, that is, a boosting system of three models used as *dj* in a higher system.

Though it is quite successful, the disadvantage of the original boosting method is that it requires a very large training sample. The sample should be divided into three and furthermore, the second and third classifiers are only trained on a subset on which the previous ones err. So unless one has a quite large training set, d_2 and d_3 will not have training
sets of reasonable size.

AdaBoost

Freund and Schapire (1996) proposed a variant, named *AdaBoost*, short for adaptive boosting, that uses the same training set over and over and thus need not be large, but the classifiers should be simple so that they do not overfit. AdaBoost can also combine an arbitrary number of baselearners, not three.

AdaBoost algorithm



```
Training:
    For all $\{x^t, r^t\}_{t=1}^N \in \mathcal{X}$, initialize $p_1^t = 1/N$
    For all base-learners $j = 1, \ldots, L$
        Randomly draw $\mathcal{X}_j$ from $\mathcal{X}$ with probabilities $p_j^t$
        Train $d_j$ using $\mathcal{X}_j$
        For each $(x^t, r^t)$, calculate $y_j^t \leftarrow d_j(x^t)$
        Calculate error rate: $\epsilon_j \leftarrow \sum_t p_j^t \cdot 1(y_j^t \neq r^t)$
        If $\epsilon_j > 1/2$, then $L \leftarrow j - 1$; stop
        $\beta_j \leftarrow \epsilon_j / (1 - \epsilon_j)$
        For each $(x^t, r^t)$, decrease probabilities if correct:
            If $y_j^t = r^t$, then $p_{j+1}^t \leftarrow \beta_j p_j^t$ Else $p_{j+1}^t \leftarrow p_j^t$
        Normalize probabilities:
            $Z_j \leftarrow \sum_t p_{j+1}^t$;  $p_{j+1}^t \leftarrow p_{j+1}^t / Z_j$
Testing:
    Given $x$, calculate $d_j(x), j = 1, \ldots, L$
    Calculate class outputs, $i = 1, \ldots, K$:
        $y_i = \sum_{j=1}^L \left(\log \frac{1}{\beta_j}\right) d_{ji}(x)$
```

The idea is to modify the probabilities of drawing the instances as a function of the error. Let us say p_j^t denotes the probability that the instance pair (x^t, r^t) is drawn to train the *j*th base-learner.
Initially, all $p_1^t = 1/N$. Then we add new base-learners as follows, starting from $j = 1$: $Є_j$ denotes the error rate of d_j .
AdaBoost requires that learners are weak, that is, $Є_j < 1/2, \forall j$; if not, we stop adding new base-learners. Note that this error rate is not on the original problem but on the dataset used at step *j*.
We define $\beta_j = Є_j /(1 - Є_j) < 1$, and we set $p_{j+1}^t = \beta_j p_j^t$ if d_j correctly classifies x^t ; otherwise, $p_{j+1}^t = p_j^t$.
Because p_{j+1}^t should be probabilities, there is a normalization where we divide p_{j+1}^t by $\sum_t p_{j+1}^t$, so that

j+1 j+1

they sum up to 1. This has the effect that the probability of a correctly classified instance is decreased, and the probability of a misclassified instance increases. Then a new sample of the same size is drawn from the original sample according to these modified probabilities, p^t_{j+1} with replacement, and is used to train d_{j+1}.

This has the effect that d_{j+1} focuses more on instances misclassified by d_j ; that is why the base-learners are chosen to be simple and not accurate, since otherwise the next training sample would contain only a few outlier and noisy instances repeated many times over. For example, with decision trees, *decision stumps*, which are trees grown only one or two levels, are used. So it is clear that these would have bias but the decrease in variance is larger and the overall error decreases. An algorithm like the linear discriminant has low variance, and we cannot gain by AdaBoosting linear discriminants.

.

Stacking - Stacked Generalization

Stacked generalization is a technique proposed by Wolpert (1992) that extends voting in that the way the output of the base-learners is combined need not be linear but is learned through a combiner system, $f(\cdot|\Phi)$, which is another learner, whose parameters Φ are also trained. (see the below given figure)

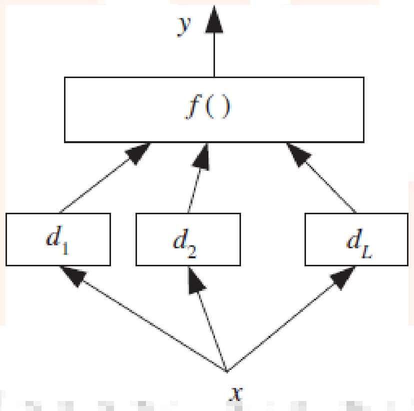


Figure: In stacked generalization, the combiner is another learner and is not restricted to being a linear combination as in voting.

$$y = f(d_1, d_2, \ldots, d_L|\Phi)$$

The combiner learns what the correct output is when the base-learners give a certain output combination. We cannot train the combiner function on the training data because the base-learners may be memorizing the training set; the combiner system should actually learn how the baselearners make errors. Stacking is a means of estimating and correcting for the biases of the base-learners. Therefore, the combiner should be trained on data unused in training the base-learners.

If $f(\cdot|w1, \ldots, wL)$ is a linear model with constraints, $wi \geq 0$, $\sum jW_j = 1$, the optimal weights can be found by constrained regression, but of course we do not need to enforce this; in stacking, there is no restriction on the combiner function and unlike voting, $f(\cdot)$ can be nonlinear. For example, it may be implemented as a multilayer perceptron with Φ its connection weights.

The outputs of the base-learners d_j define a new L-dimensional space in which the output discriminant/regression function is learned by the combiner function.

In stacked generalization, we would like the base-learners to be as different as possible so that they will complement each other, and, for this, it is best if they are based on different learning algorithms. If we are combining classifiers that can generate continuous outputs, for example, posterior probabilities, it isbetter that they be the combined rather than hard decisions.

When we compare a trained combiner as we have in stacking, with a fixed rule such as in voting, we see that both have their advantages: A trained rule is more flexible and may have less bias, but adds extra parameters, risks introducing variance, and needs extra time and data for training. Note also that there is no need to normalize classifier outputs before stacking.