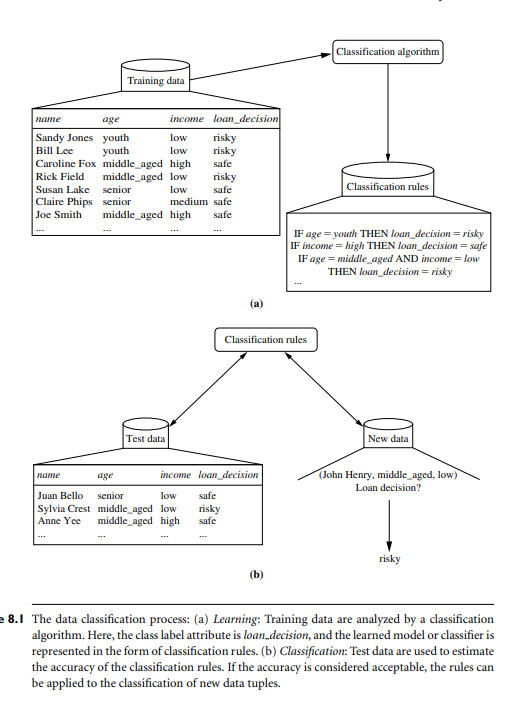
## **UNIT—IV:**

# Classification- Basic Concepts, Decision Tree Induction, Bayes Classification Methods: Baye’s Theorem, Naive Bayesian Classification, Rule-Based Classification: Using IF-THEN Rules for Classification, Rule Extraction from a Decision Tree, Rule Induction Using a Sequential Covering Algorithm, Model Evaluation and Selection, Techniques to Improve Classification Accuracy.

# Classification is a form of data analysis that extracts models describing important data classes. Such models, called classifiers, predict categorical (discrete, unordered) class labels.

“How does classification work?” Data classification is a two-step process, consisting of a learning step (where a classification model is constructed) and a classification step (where the model is used to predict class labels for given data).

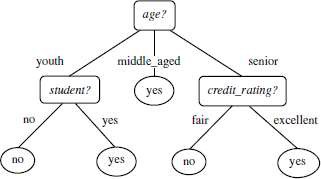


Classification by Decision Tree Induction:

Decision tree induction is the learning of decision trees from class-labeled training tuples.

A decision tree is a flowchart-like tree structure, where

* + Each internal node denotes a test on an attribute.
  + Each branch represents an outcome of the test.
  + Each leaf node holds a class label.
  + The topmost node in a tree is the root node.



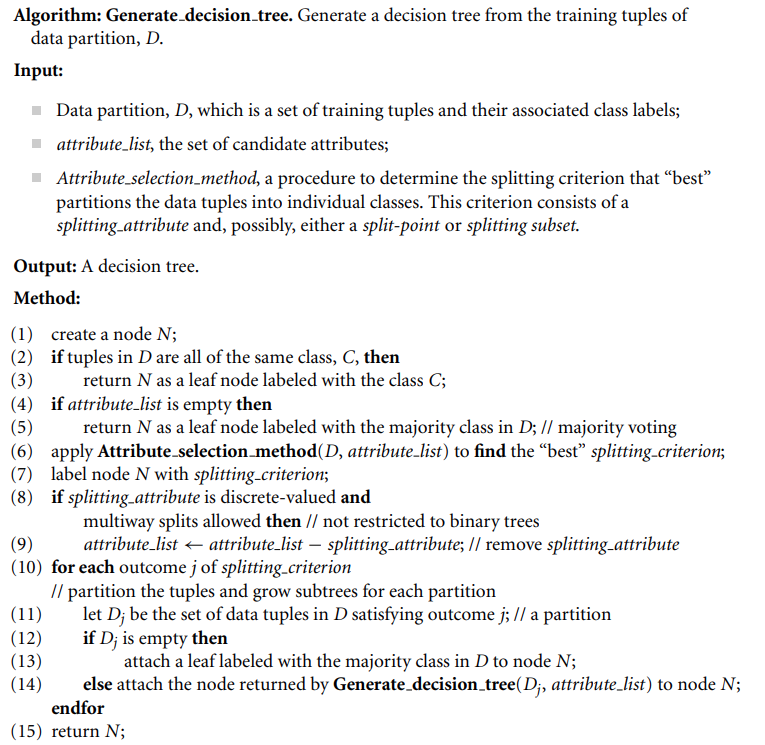
The construction of decision tree classifiers does not require any domain knowledge or parameter setting, and therefore its appropriate for exploratory knowledge discovery.

* Decision trees can handle high dimensional data.
* Their representation of acquired knowledge in tree form is intuitive and generally easy to assimilate by humans.
* The learning and classification steps of decision tree induction are simple and fast. In general, decision tree classifiers have good accuracy.
* Decision tree induction algorithms have been used for classification in many application areas, such as medicine, manufacturing and production, financial analysis, astronomy, and molecular biology.

## Algorithm For Decision Tree Induction:

The algorithm is called with three parameters:

1. Data partition
2. Attribute list
3. Attribute selection method



The parameter attribute list is a list of attributes describing the tuples.



Attribute selection method specifies a heuristic procedure for selecting the attribute that



- best discriminates the given tuples according to class.

The tree starts as a single node, *N,* representing the training tuples in *D.*



If the tuples in *D* are all of the same class, then node *N* becomes a leaf and is labelled with that class.



All of the terminating conditions are explained at the end of the algorithm. Otherwise, the algorithm calls Attribute selection method to determine the splitting criterion.





The splitting criterion tells us which attribute to test at node N by determining the-best way to separate or partition the tuples in D into individual classes.



There are three possible scenarios. Let *A* be the splitting attribute . *A* has *v* distinct values,

{a1, a2, ... ,av}, based on the training data.

### A is discrete-valued:

In this case, the outcomes of the test at node N correspond directly to the known values of A.



A branch is created for each known value, aj, of A and labeled with that value. Aneed not be considered in any future partitioning of the tuples.



### A is continuous-valued:

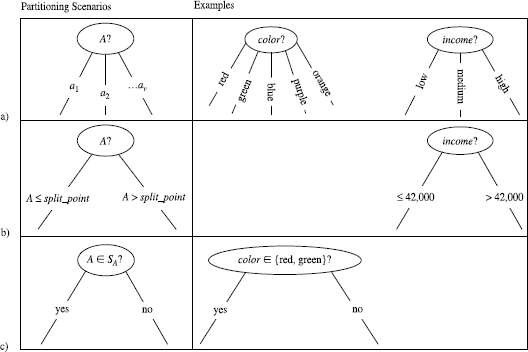
In this case, the test at node N has two possible outcomes, corresponding to the conditions A <=split point and A >split point, respectively

wheresplit point is the split-point returned by Attribute selection method as part of the splitting criterion.

### A is discrete-valued and a binary tree must be produced:

The test at node N is of the form-A€SA?II.

SA is the splitting subset for A, returned by Attribute selection methodas part of the splitting criterion. It is a subset of the known values of A.



If A is Discrete valued (b) If A is continuous valued (c) If A is discrete-valued and a binary tree must be produced.

# Bayes’ theorem is named after Thomas Bayes, a nonconformist English clergyman who did early work in probability and decision theory during the 18th century. Let X be a data tuple. In Bayesian terms, X is considered “evidence.” As usual, it is described by measurements made on a set of n attributes. Let H be some hypothesis such as that the data tuple X belongs to a specified class C. For classification problems, we want to determine P(H|X), the probability that the hypothesis H holds given the “evidence” or observed data tuple X. In other words, we are looking for the probability that tuple X belongs to class C, given that we know the attribute description of X.

# P(H|X) is the posterior probability, or a posteriori probability, of H conditioned on X. For example, suppose our world of data tuples is confined to customers described by the attributes age and income, respectively, and that X is a 35-year-old customer with an income of $40,000. Suppose that H is the hypothesis that our customer will buy a computer. Then P(H|X) reflects the probability that customer X will buy a computer given that we know the customer’s age and income.

# In contrast, P(H) is the prior probability, or a priori probability, of H. For our example, this is the probability that any given customer will buy a computer, regardless of age, income, or any other information, for that matter. The posterior probability, P(H|X), is based on more information (e.g., customer information) than the prior probability, P(H), which is independent of X.

# Similarly, P(X|H) is the posterior probability of X conditioned on H. That is, it is the probability that a customer, X, is 35 years old and earns $40,000, given that we know the customer will buy a computer.

# P(X) is the prior probability of X. Using our example, it is the probability that a person from our set of customers is 35 years old and earns $40,000.

# “How are these probabilities estimated?” P(H), P(X|H), and P(X) may be estimated from the given data, as we shall see next. Bayes’ theorem is useful in that it provides a way of calculating the posterior probability, P(H|X), from P(H), P(X|H), and P(X). Bayes’ theorem is



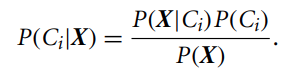
The naive Bayesian classifier, or simple Bayesian classifier, works as follows:

1. Let D be a training set of tuples and their associated class labels. As usual, each tuple is represented by an n-dimensional attribute vector, X = (x1, x2,..., xn), depicting n measurements made on the tuple from n attributes, respectively, A1, A2,..., An.

2. Suppose that there are m classes, C1, C2,..., Cm. Given a tuple, X, the classifier will predict that X belongs to the class having the highest posterior probability, condi-tioned on X. That is, the na¨ıve Bayesian classifier predicts that tuple X belongs to the class Ci if and only if

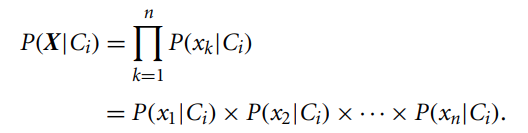


Thus, we maximize P(Ci|X). The class Ci for which P(Ci|X) is maximized is called the maximum posteriori hypothesis. By Bayes’ theorem



3. As P(X) is constant for all classes, only P(X|Ci)P(Ci) needs to be maximized. If the class prior probabilities are not known, then it is commonly assumed that the classes are equally likely, that is, P(C1) = P(C2) = ··· = P(Cm), and we would therefore maximize P(X|Ci). Otherwise, we maximize P(X|Ci)P(Ci). Note that the class prior probabilities may be estimated by P(Ci) = |Ci,D|/|D|, where |Ci,D| is the number of training tuples of class Ci in D.

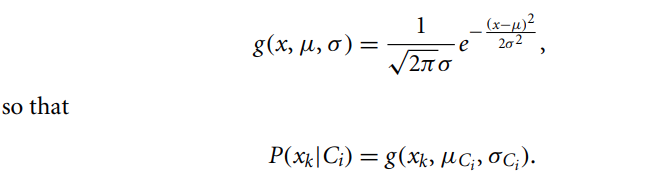
4. Given data sets with many attributes, it would be extremely computationally expensive to compute P(X|Ci). To reduce computation in evaluating P(X|Ci), the na¨ıve assumption of class-conditional independence is made. This presumes that the attributes’ values are conditionally independent of one another, given the class label of the tuple (i.e., that there are no dependence relationships among the attributes). Thus,



We can easily estimate the probabilities P(x1|Ci), P(x2|Ci),..., P(xn|Ci) from the training tuples. Recall that here xk refers to the value of attribute Ak for tuple X. For each attribute, we look at whether the attribute is categorical or continuous-valued. For instance, to compute P(X|Ci), we consider the following:

(a) If Ak is categorical, then P(xk|Ci) is the number of tuples of class Ci in D having the value xk for Ak, divided by |Ci,D|, the number of tuples of class Ci in D.

(b) If Ak is continuous-valued, then we need to do a bit more work, but the cal-culation is pretty straightforward. A continuous-valued attribute is typically assumed to have a Gaussian distribution with a mean µ and standard deviation σ , defined by



These equations may appear daunting, We need to compute µCi and σCi , which are the mean (i.e., average) and standard deviation, respectively, of the values of attribute Ak for training tuples of class Ci. We then plug these two quantities into Eq. (8.13), together with xk, to estimate P(xk|Ci).

5.To predict the class label of X, P(X|Ci)P(Ci) is evaluated for each class Ci. The classifier predicts that the class label of tuple X is the class Ci if and only if



In other words, the predicted class label is the class Ci for which P(X|Ci)P(Ci) is the maximum.

Rule-based classifiers, where the learned model is represented as a set of IF-THEN rules. We first examine how such rules are used for classification. We then study ways in which they can be generated, either from a decision tree or directly from the training data using a sequential covering algorithm.

Using IF-THEN Rules for Classification

Rules are a good way of representing information or bits of knowledge. A rule-based classifier uses a set of IF-THEN rules for classification. An IF-THEN rule is an expres-sion of the form

IF condition THEN conclusion

An example is rule R1,

R1: IF age = youth AND student = yes THEN buys computer = yes.

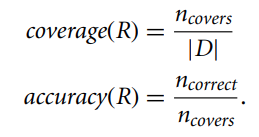
The “IF” part (or left side) of a rule is known as the rule antecedent or precondition. The “THEN” part (or right side) is the rule consequent. In the rule antecedent, the condition consists of one or more attribute tests (e.g., age = youth and student = yes)

that are logically ANDed. The rule’s consequent contains a class prediction (in this case, we are predicting whether a customer will buy a computer). R1 can also be written as

R1: (age = youth) ∧ (student = yes) ⇒ (buys computer = yes).

If the condition (i.e., all the attribute tests) in a rule antecedent holds true for a given tuple, we say that the rule antecedent is satisfied (or simply, that the rule is satisfied) and that the rule covers the tuple.

A rule R can be assessed by its coverage and accuracy. Given a tuple, X, from a class-labeled data set, D, let ncovers be the number of tuples covered by R; ncorrect be the number of tuples correctly classified by R; and |D| be the number of tuples in D. We can define the coverage and accuracy of R as

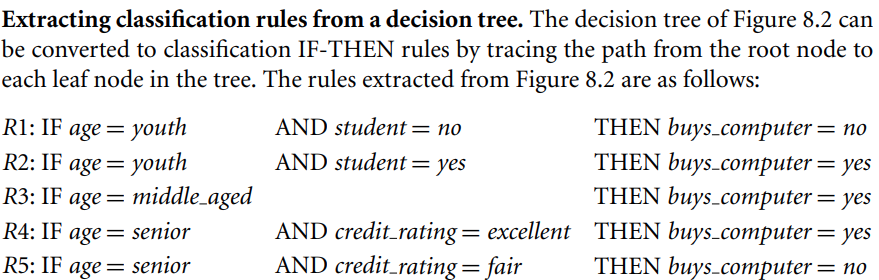


That is, a rule’s coverage is the percentage of tuples that are covered by the rule (i.e., their attribute values hold true for the rule’s antecedent). For a rule’s accuracy, we look at the tuples that it covers and see what percentage of them the rule can correctly classify.

Rule Extraction from a Decision Tree

Decision tree classifiers are a popular method of classification—it is easy to under-stand how decision trees work and they are known for their accuracy. Decision trees can become large and difficult to interpret. In this subsection, we look at how to build a rule-based classifier by extracting IF-THEN rules from a decision tree. In comparison with a decision tree, the IF-THEN rules may be easier for humans to understand, particularly if the decision tree is very large.

To extract rules from a decision tree, one rule is created for each path from the root to a leaf node. Each splitting criterion along a given path is logically ANDed to form the rule antecedent (“IF” part). The leaf node holds the class prediction, forming the rule consequent (“THEN” part).



A disjunction (logical OR) is implied between each of the extracted rules. Because the rules are extracted directly from the tree, they are mutually exclusive and exhaustive. Mutually exclusive means that we cannot have rule conflicts here because no two rules will be triggered for the same tuple. (We have one rule per leaf, and any tuple can map to only one leaf.) Exhaustive means there is one rule for each possible attribute–value combination, so that this set of rules does not require a default rule. Therefore, the order of the rules does not matter—they are unordered.

Since we end up with one rule per leaf, the set of extracted rules is not much simpler than the corresponding decision tree! The extracted rules may be even more difficult to interpret than the original trees in some cases. The resulting set of rules extracted can be large and difficult to follow, because some of the attribute tests may be irrelevant or redundant. So, the plot thickens. Although it is easy to extract rules from a decision tree, we may need to do some more work by pruning the resulting rule set.

“How can we prune the rule set?” For a given rule antecedent, any condition that does not improve the estimated accuracy of the rule can be pruned (i.e., removed), thereby generalizing the rule. C4.5 extracts rules from an unpruned tree, and then prunes the rules using a pessimistic approach similar to its tree pruning method. The training tuples and their associated class labels are used to estimate rule accuracy. However, because this would result in an optimistic estimate, alternatively, the estimate is adjusted to compen-sate for the bias, resulting in a pessimistic estimate. In addition, any rule that does not contribute to the overall accuracy of the entire rule set can also be pruned.

Other problems arise during rule pruning, however, as the rules will no longer be mutually exclusive and exhaustive. For conflict resolution, C4.5 adopts a class-based ordering scheme. It groups together all rules for a single class, and then determines a ranking of these class rule sets. Within a rule set, the rules are not ordered. C4.5 orders the class rule sets so as to minimize the number of false-positive errors (i.e., where a rule predicts a class, C, but the actual class is not C). The class rule set with the least number of false positives is examined first. Once pruning is complete, a final check is done to remove any duplicates. When choosing a default class, C4.5 does not choose the majority class, because this class will likely have many rules for its tuples. Instead, it selects the class that contains the most training tuples that were not covered by any rule.

Rule Induction Using a Sequential Covering Algorithm

IF-THEN rules can be extracted directly from the training data (i.e., without having to generate a decision tree first) using a sequential covering algorithm. The name comes from the notion that the rules are learned sequentially (one at a time), where each rule for a given class will ideally cover many of the class’s tuples (and hopefully none of the tuples of other classes). Sequential covering algorithms are the most widely used approach to mining disjunctive sets of classification rules, and form the topic of this subsection.

There are many sequential covering algorithms. Popular variations include AQ, CN2, and the more recent RIPPER. The general strategy is as follows. Rules are learned one at a time. Each time a rule is learned, the tuples covered by the rule are removed, and the process repeats on the remaining tuples. This sequential learning of rules is in contrast to decision tree induction. Because the path to each leaf in a decision tree corresponds to a rule, we can consider decision tree induction as learning a set of rules simultaneously.

A basic sequential covering algorithm is shown below. Here, rules are learned for one class at a time. Ideally, when learning a rule for a class, C, we would like the rule to cover all (or many) of the training tuples of class C and none (or few) of the tuples

Algorithm: Sequential covering. Learn a set of IF-THEN rules for classification.

Input:

D, a data set of class-labeled tuples;

Att vals, the set of all attributes and their possible values.

Output: A set of IF-THEN rules.

Method:

(1) Rule set = {}; // initial set of rules learned is empty

(2) for each class c do

(3) repeat

(4) Rule = Learn One Rule(D, Att vals, c);

(5) remove tuples covered by Rule from D;

(6) Rule set = Rule set + Rule; // add new rule to rule set

(7) until terminating condition;

(8) endfor

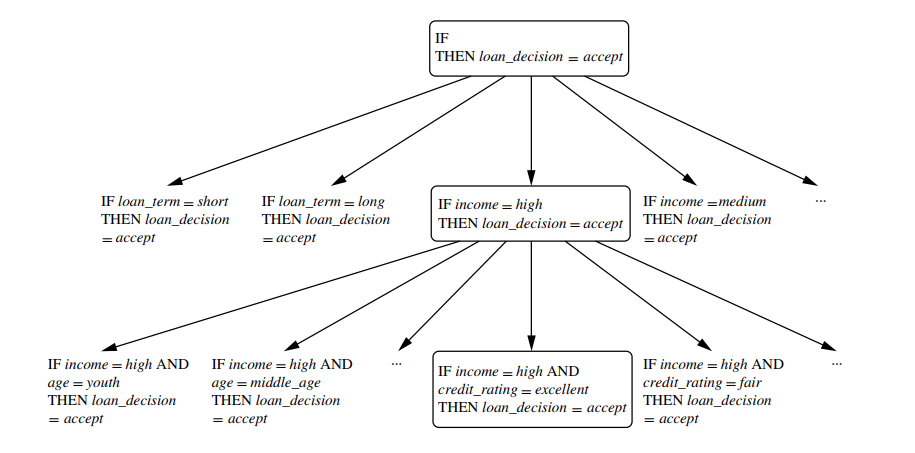
(9) return Rule Set;

from other classes. In this way, the rules learned should be of high accuracy. The rules need not necessarily be of high coverage. This is because we can have more than one rule for a class, so that different rules may cover different tuples within the same class. The process continues until the terminating condition is met, such as when there are no more training tuples or the quality of a rule returned is below a user-specified threshold. The Learn One Rule procedure finds the “best” rule for the current class, given the current set of training tuples.

“How are rules learned?” Typically, rules are grown in a general-to-specific manner. We can think of this as a beam search, where we start off with an empty rule and then gradually keep appending attribute tests to it. We append by adding the attribute test as a logical conjunct to the existing condition of the rule antecedent. Sup-pose our training set, D, consists of loan application data. Attributes regarding each applicant include their age, income, education level, residence, credit rating, and the term of the loan. The classifying attribute is loan decision, which indicates whether a loan is accepted (considered safe) or rejected (considered risky). To learn a rule for the class “accept,” we start off with the most general rule possible, that is, the condition of the rule antecedent is empty. The rule is

IF THEN loan decision = accept.

We then consider each possible attribute test that may be added to the rule. These can be derived from the parameter Att vals, which contains a list of attributes with their associated values. For example, for an attribute–value pair (att, val), we can consider attribute tests such as att = val, att ≤ val, att > val, and so on. Typically, the training data will contain many attributes, each of which may have several possible values. Find-ing an optimal rule set becomes computationally explosive. Instead, Learn One Rule



adopts a greedy depth-first strategy. Each time it is faced with adding a new attribute test (conjunct) to the current rule, it picks the one that most improves the rule qual-ity, based on the training samples. We will say more about rule quality measures in a minute. For the moment, let’s say we use rule accuracy as our quality measure. suppose Learn One Rule finds that the attribute test income = high best improves the accuracy of our current (empty) rule. We append it to the condition, so that the current rule becomes

IF income = high THEN loan decision = accept.

Each time we add an attribute test to a rule, the resulting rule should cover relatively more of the “accept” tuples. During the next iteration, we again consider the possible attribute tests and end up selecting credit rating = excellent. Our current rule grows to become

IF income = high AND credit rating = excellent THEN loan decision = accept.

The process repeats, where at each step we continue to greedily grow rules until the resulting rule meets an acceptable quality level.

Greedy search does not allow for backtracking. At each step, we heuristically add what appears to be the best choice at the moment. What if we unknowingly made a poor choice along the way? To lessen the chance of this happening, instead of selecting the best attribute test to append to the current rule, we can select the best k attribute tests. In this way, we perform a beam search of width k, wherein we maintain the k best candidates overall at each step, rather than a single best candidate.

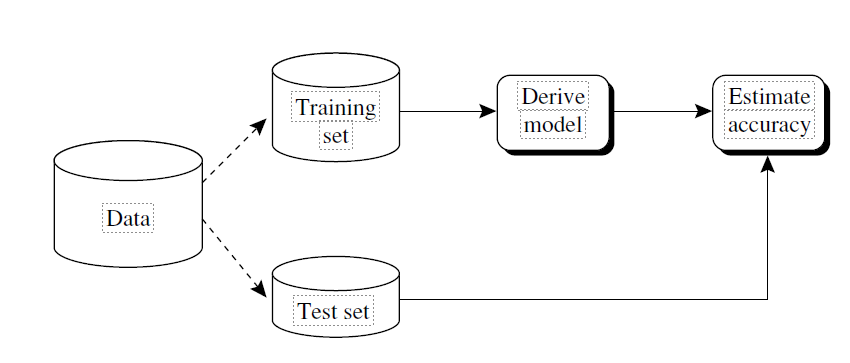
Model Evaluation and Selection

various evaluation metrics for the predictive accuracy of a classifier. Holdout and random subsampling, cross-validation, and bootstrap methods are common techniques for assessing accuracy, based on randomly sampled partitions of the given data.

Holdout Method and Random Subsampling

The holdout method is what we have alluded to so far in our discussions about accuracy. In this method, the given data are randomly partitioned into two independent sets, a training set and a test set. Typically, two-thirds of the data are allocated to the training set, and the remaining one-third is allocated to the test set. The training set is used to derive the model. The model’s accuracy is then estimated with the test set. The estimate is pessimistic because only a portion of the initial data is used to derive the model.

Random subsampling is a variation of the holdout method in which the holdout method is repeated k times. The overall accuracy estimate is taken as the average of the accuracies obtained from each iteration.



Cross-Validation

In k-fold cross-validation, the initial data are randomly partitioned into k mutually exclusive subsets or “folds,” D1, D2,..., Dk, each of approximately equal size. Training and testing is performed k times. In iteration i, partition Di is reserved as the test set, and the remaining partitions are collectively used to train the model. That is, in the first iteration, subsets D2,..., Dk collectively serve as the training set to obtain a first model, which is tested on D1; the second iteration is trained on subsets D1, D3,..., Dk and tested on D2; and so on. Unlike the holdout and random subsampling methods, here each sample is used the same number of times for training and once for testing. For classification, the accuracy estimate is the overall number of correct classifications from the k iterations, divided by the total number of tuples in the initial data.

Leave-one-out is a special case of k-fold cross-validation where k is set to the number of initial tuples. That is, only one sample is “left out” at a time for the test set. In strat-ified cross-validation, the folds are stratified so that the class distribution of the tuples in each fold is approximately the same as that in the initial data.

In general, stratified 10-fold cross-validation is recommended for estimating accu-racy (even if computation power allows using more folds) due to its relatively low bias and variance.

Bootstrap

Unlike the accuracy estimation methods just mentioned, the bootstrap method sam-ples the given training tuples uniformly with replacement. That is, each time a tuple is selected, it is equally likely to be selected again and re-added to the training set. For instance, imagine a machine that randomly selects tuples for our training set. In sam-pling with replacement, the machine is allowed to select the same tuple more than once.

Techniques to Improve Classification Accuracy

We focus on ensemble methods. An ensemble for classification is a composite model, made up of a combination of classifiers. The individual classifiers vote, and a class label prediction is returned by the ensemble based on the collection of votes. Ensembles tend to be more accurate than their component classifiers. ensemble methods in general. Bagging, boosting, and random forests are popular ensemble methods.

Traditional learning models assume that the data classes are well distributed. In many real-world data domains, however, the data are class-imbalanced, where the main class of interest is represented by only a few tuples. This is known as the class imbalance problem. We also study techniques for improving the classification accuracy of class-imbalanced data.

Introducing Ensemble Methods

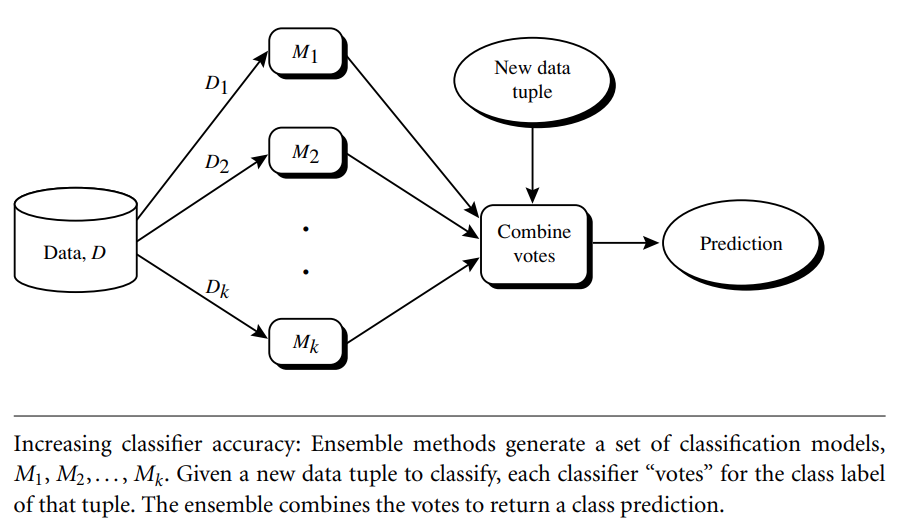
Bagging, boosting, and random forests are examples of ensemble methods.

An ensemble combines a series of k learned models (or base classifiers), M1, M2,..., Mk, with the aim of creating an improved composite classification model, M∗. A given data

set, D, is used to create k training sets, D1, D2,..., Dk, where Di (1 ≤ i ≤ k − 1) is used to generate classifier Mi. Given a new data tuple to classify, the base classifiers each vote by returning a class prediction. The ensemble returns a class prediction based on the votes of the base classifiers.

An ensemble tends to be more accurate than its base classifiers. For example, con-sider an ensemble that performs majority voting. That is, given a tuple X to classify, it collects the class label predictions returned from the base classifiers and outputs the class in majority. The base classifiers may make mistakes, but the ensemble will misclassify X only if over half of the base classifiers are in error. Ensembles yield better results when there is significant diversity among the models. That is, ideally, there is little correlation among classifiers. The classifiers should also perform better than random guessing. Each base classifier can be allocated to a different CPU and so ensemble methods are parallelizable.

To help illustrate the power of an ensemble, consider a simple two-class problem described by two attributes, x1 and x2. The problem has a linear decision boundary. The decision boundary of an ensemble of decision tree classifiers on the same problem. Although the ensemble’s decision boundary is still piecewise constant, it has a finer resolution and is better than that of a single tree.



Bagging

We now take an intuitive look at how bagging works as a method of increasing accuracy. Suppose that you are a patient and would like to have a diagnosis made based on your symptoms. Instead of asking one doctor, you may choose to ask several. If a certain diagnosis occurs more than any other, you may choose this as the final or best diagnosis. That is, the final diagnosis is made based on a majority vote, where each doctor gets an equal vote. Now replace each doctor by a classifier, and you have the basic idea behind bagging. Intuitively, a majority vote made by a large group of doctors may be more reliable than a majority vote made by a small group.

Given a set, D, of d tuples, bagging works as follows. For iteration i (i = 1, 2,..., k), a training set, Di, of d tuples is sampled with replacement from the original set of tuples, D. Note that the term bagging stands for bootstrap aggregation. Each training set is a bootstrap sample. Because sampling with replacement is used, some of the original tuples of D may not be included in Di, whereas others may occur more than once. A classifier model, Mi, is learned for each training set, Di. To classify an unknown tuple, X, each classifier, Mi, returns its class prediction, which counts as one vote. The bagged classifier, M∗, counts the votes and assigns the class with the most votes to X. Bagging can be applied to the prediction of continuous values by taking the average value of each prediction for a given test tuple.

The bagged classifier often has significantly greater accuracy than a single classifier derived from D, the original training data. It will not be considerably worse and is more robust to the effects of noisy data and overfitting. The increased accuracy occurs because the composite model reduces the variance of the individual classifiers.

Algorithm: Bagging. The bagging algorithm—create an ensemble of classification models

for a learning scheme where each model gives an equally weighted prediction.

Input:

D, a set of d training tuples;

k, the number of models in the ensemble;

a classification learning scheme (decision tree algorithm, na¨ıve Bayesian, etc.).

Output: The ensemble—a composite model, M∗.

Method:

(1) for i = 1 to k do // create k models:

(2) create bootstrap sample, Di, by sampling D with replacement;

(3) use Di and the learning scheme to derive a model, Mi;

(4) endfor

To use the ensemble to classify a tuple, X:

let each of the k models classify X and return the majority vote;

Boosting and AdaBoost

We now look at the ensemble method of boosting. As in the previous section, suppose that as a patient, you have certain symptoms. Instead of consulting one doctor, you choose to consult several. Suppose you assign weights to the value or worth of each doc-tor’s diagnosis, based on the accuracies of previous diagnoses they have made. The final diagnosis is then a combination of the weighted diagnoses. This is the essence behind boosting.

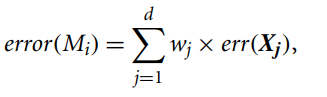
In boosting, weights are also assigned to each training tuple. A series of k classifiers is iteratively learned. After a classifier, Mi, is learned, the weights are updated to allow the subsequent classifier, Mi+1, to “pay more attention” to the training tuples that were mis-classified by Mi. The final boosted classifier, M∗, combines the votes of each individual classifier, where the weight of each classifier’s vote is a function of its accuracy.

AdaBoost (short for Adaptive Boosting) is a popular boosting algorithm. Suppose we want to boost the accuracy of a learning method. We are given D, a data set of d class-labeled tuples, (X1,y1),(X2,y2),...,(Xd,yd), where yi is the class label of tuple Xi. Initially, AdaBoost assigns each training tuple an equal weight of 1/d. Generating k classifiers for the ensemble requires k rounds through the rest of the algorithm. In round i, the tuples from D are sampled to form a training set, Di, of size d. Sampling

with replacement is used—the same tuple may be selected more than once. Each tuple’s chance of being selected is based on its weight. A classifier model, Mi, is derived from the training tuples of Di. Its error is then calculated using Di as a test set. The weights of the training tuples are then adjusted according to how they were classified.

If a tuple was incorrectly classified, its weight is increased. If a tuple was correctly classified, its weight is decreased. A tuple’s weight reflects how difficult it is to classify—the higher the weight, the more often it has been misclassified. These weights will be used to generate the training samples for the classifier of the next round. The basic idea is that when we build a classifier, we want it to focus more on the misclassified tuples of the previous round. Some classifiers may be better at classifying some “difficult” tuples than others. In this way, we build a series of classifiers that complement each other. The algorithm is summarized below.

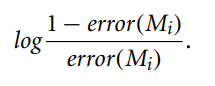
Now, let’s look at some of the math that’s involved in the algorithm. To compute the error rate of model Mi, we sum the weights of each of the tuples in Di that Mi misclassified. That is,



where err(Xj) is the misclassification error of tuple Xj: If the tuple was misclassified, then err(Xj) is 1; otherwise, it is 0. If the performance of classifier Mi is so poor that its error exceeds 0.5, then we abandon it. Instead, we try again by generating a new Di training set, from which we derive a new Mi.

The error rate of Mi affects how the weights of the training tuples are updated. If a tuple in round i was correctly classified, its weight is multiplied by error(Mi)/(1 − error(Mi)). Once the weights of all the correctly classified tuples are updated, the weights for all tuples (including the misclassified ones) are normalized so that their sum remains the same as it was before. To normalize a weight, we multiply it by the sum of the old weights, divided by the sum of the new weights. As a result, the weights of mis-classified tuples are increased and the weights of correctly classified tuples are decreased, as described before.

“Once boosting is complete, how is the ensemble of classifiers used to predict the class label of a tuple, X?” Unlike bagging, where each classifier was assigned an equal vote, boosting assigns a weight to each classifier’s vote, based on how well the classifier performed. The lower a classifier’s error rate, the more accurate it is, and therefore, the higher its weight for voting should be. The weight of classifier Mi’s vote is



For each class, c, we sum the weights of each classifier that assigned class c to X. The class with the highest sum is the “winner” and is returned as the class prediction for tuple X. “How does boosting compare with bagging?” Because of the way boosting focuses on the misclassified tuples, it risks overfitting the resulting composite model to such data.

Algorithm: AdaBoost. A boosting algorithm—create an ensemble of classifiers. Each one

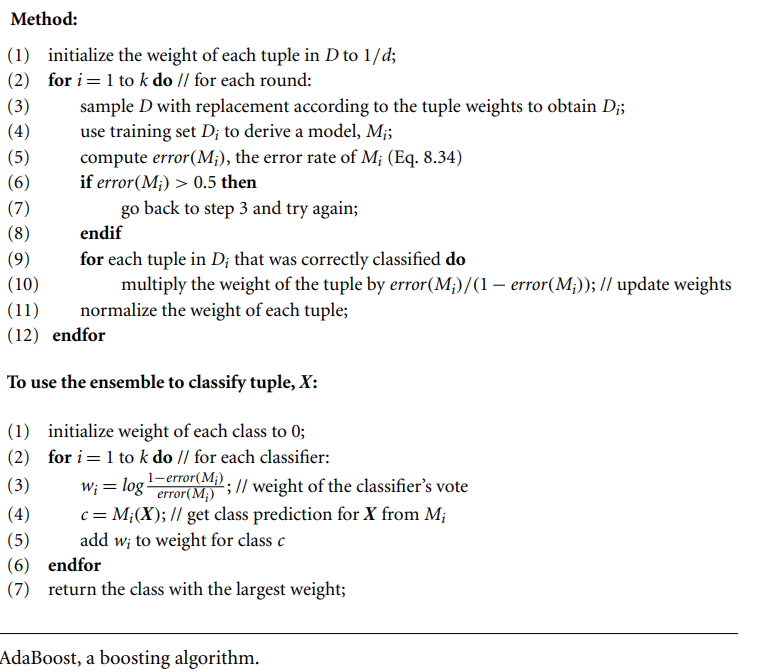
gives a weighted vote.

Input:

D, a set of d class-labeled training tuples;

k, the number of rounds (one classifier is generated per round); a classification learning scheme.

Output: A composite model.



Therefore, sometimes the resulting “boosted” model may be less accurate than a single model derived from the same data. Bagging is less susceptible to model overfitting. While both can significantly improve accuracy in comparison to a single model, boosting tends to achieve greater accuracy.

Random Forests

We now present another ensemble method called random forests. Imagine that each of the classifiers in the ensemble is a decision tree classifier so that the collection of classifiers

is a “forest.” The individual decision trees are generated using a random selection of attributes at each node to determine the split. More formally, each tree depends on the values of a random vector sampled independently and with the same distribution for all trees in the forest. During classification, each tree votes and the most popular class is returned.

Random forests can be built using bagging in tandem with random attribute selection. A training set, D, of d tuples is given. The general procedure to gen-erate k decision trees for the ensemble is as follows. For each iteration, i (i = 1, 2,..., k), a training set, Di, of d tuples is sampled with replacement from D. That is, each Di is a bootstrap sample of D (Section 8.5.4), so that some tuples may occur more than once in Di, while others may be excluded. Let F be the number of attributes to be used to determine the split at each node, where F is much smaller than the number of avail-able attributes. To construct a decision tree classifier, Mi, randomly select, at each node, F attributes as candidates for the split at the node. The CART methodology is used to grow the trees. The trees are grown to maximum size and are not pruned. Random forests formed this way, with random input selection, are called Forest-RI.

Another form of random forest, called Forest-RC, uses random linear combinations of the input attributes. Instead of randomly selecting a subset of the attributes, it creates new attributes (or features) that are a linear combination of the existing attributes. That is, an attribute is generated by specifying L, the number of original attributes to be combined. At a given node, L attributes are randomly selected and added together with coefficients that are uniform random numbers on [−1,1]. F linear combinations are generated, and a search is made over these for the best split. This form of random forest is useful when there are only a few attributes available, so as to reduce the correlation between individual classifiers.

Random forests are comparable in accuracy to AdaBoost, yet are more robust to errors and outliers. The generalization error for a forest converges as long as the num-ber of trees in the forest is large. Thus, overfitting is not a problem. The accuracy of a random forest depends on the strength of the individual classifiers and a measure of the dependence between them. The ideal is to maintain the strength of individual classifiers without increasing their correlation. Random forests are insensitive to the number of attributes selected for consideration at each split. Typically, up to log2d + 1 are chosen.(An interesting empirical observation was that using a single random input attribute may result in good accuracy that is often higher than when using several attributes.) Because random forests consider many fewer attributes for each split, they are efficient on very large databases. They can be faster than either bagging or boosting. Random forests give internal estimates of variable importance.