5

Combining Base-Learners

Model combination consists of creating a single learning system from a collection of learning algorithms. In some sense, model combination may be viewed as a variation on the theme of combining data mining operations discussed in Chapter 4. There are two basic approaches to model combination. The first one exploits variability in the application's data and combines multiple copies of a single learning algorithm applied to different subsets of that data. The second one exploits variability among learning algorithms and combines several learning algorithms applied to the same application's data.

The main motivation for combining models is to reduce the probability of misclassification based on any single induced model by increasing the system's area of expertise through combination. Indeed, one of the implicit assumptions of model selection in metalearning is that there exists an optimal learning algorithm for each task. Although this clearly holds in the sense that, given a task ϕ and a set of learning algorithms $\{A_k\}$, there is a learning algorithm A_ϕ in $\{A_k\}$ that performs better than all of the others on ϕ, the actual performance of A_ϕ may still be poor. In some cases, one may mitigate the risk of settling for a suboptimal learning algorithm by replacing single model selection with model combination.

Because it draws on information about base-level learning — in terms of either the characteristics of various subsets of data or the characteristics of various learning algorithms — model combination is often considered a form of metalearning. This chapter is dedicated to a brief overview of model combination. We limit our presentation to a description of each individual technique and leave it to the interested reader to follow the references and other relevant literature for discussions of comparative performance among them.

To help with understanding and to motivate the chapter's organization, Table 5 summarizes, for each combination technique, the underlying philosophy, the type of base-level information used to drive the combination at the meta level (i.e., metadata), and the nature of the metaknowledge generated,

whether explicitly or implicitly. Further details are in the corresponding sections.

Table 5.1. Model combination techniques summary

Technique	Philosophy	metadata	metaknowledge
Bagging	Variation in data		Implicit in voting scheme
Boosting		Errors (updated distribution)	Voting scheme's weights
Stacking	Variation among learners (multi-expert)	Class predictions or probabilities	Mapping from metadata to class predictions
Cascade generalization		Class probabilities and base level attributes	Mapping from metadata to class predictions
Cascading	Variation among learners (multistage)	Confidence on predictions (updated distribution)	Implicit in selection scheme
Delegating		Confidence on predictions	Implicit in delegation scheme
Arbitrating	Variation among learners (refereed)	Correctness of class predictions, base level attributes and internal propositions	Mappings from metadata to correctness (one for each learner)
Meta-decision trees	Variation in data and among learners	Class distribution properties (from samples)	Mapping from metadata to best model

5.1 Bagging and Boosting

Perhaps the most well-known techniques for exploiting variation in data are bagging and boosting. Both bagging and boosting combine multiple models built from a single learning algorithm by systematically varying the training data.

5.1.1 Bagging

Bagging

Bagging, which stands for bootstrap aggregating, is due to Breiman [43]. Given a learning algorithm A and a set of training data T, bagging first draws N samples $S_1, \ldots, S_N$, with replacement, from T. It then applies A independently to each sample to induce N models $h_1, \ldots, h_N$.[1] When classifying a new query instance q, the induced models are combined via a simple voting scheme, where the class assigned to the new instance is the class that is predicted most often among the N models, as illustrated in Figure 5.1. The bagging algorithm for classification is shown in Figure 5.2.

Bagging is easily extended to regression by replacing the voting scheme of line 5 of the algorithm by an average of the models' predictions:

[1] To be consistent with the literature, note that we shall use the term *model* rather than *hypothesis* throughout this chapter. However, we shall retain our established mathematical notation and denote a model by h.

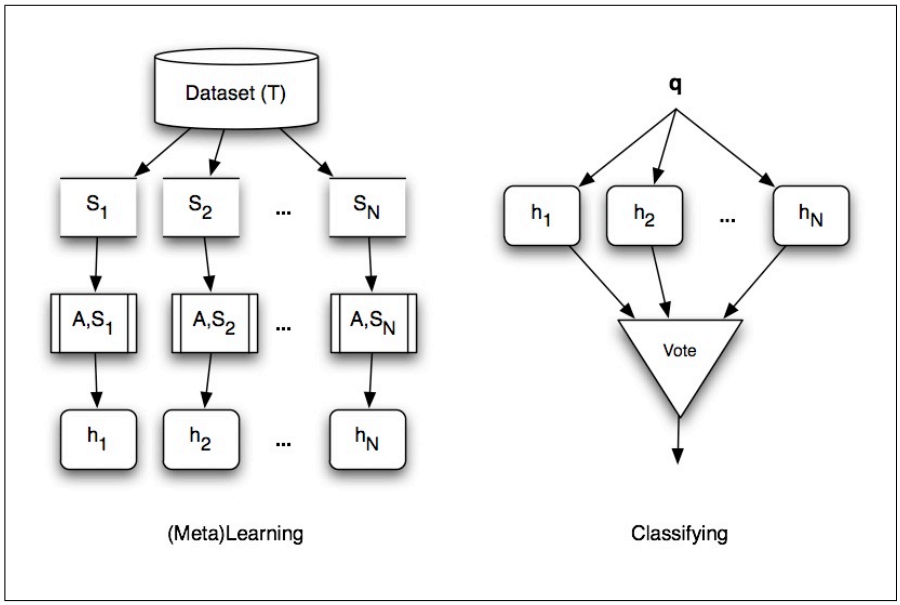

Fig. 5.1. Bagging

Algorithm Bagging(T, A, N, d)
1. For $k = 1$ to N
2. S_k = random sample of size d drawn from T, with replacement
3. h_k = model induced by A from S_k
4. For each new query instance q
5. Class(q) = $argmax_{y \in \mathcal{Y}} \sum_{k=1}^{N} \delta(y, h_i(q))$

where:
T is the training set
A is the chosen learning algorithm
N is the number of samples or bags, each of size d, drawn from T
$\mathcal{Y}$ is the finite set of target class values
δ is the generalized Kronecker function ($\delta(a, b) = 1$ if $a = b$; 0 otherwise)

Fig. 5.2. Bagging algorithm for classification

$$\text{Value}(q) = \frac{\sum_{i=1}^{N} h_i(q)}{N}$$

Bagging is most effective when the base-learner is *unstable*. A learner is unstable if it is highly sensitive to data, in the sense that small perturbations in the data cause large changes in the induced model. One simple example of

instability is order dependence, where the order in which training instances are presented has a significant impact on the learner's output.

Bagging typically increases accuracy. However, if A produces interpretable models (e.g., decision trees, rules), that interpretability is lost when bagging is applied to A.

5.1.2 Boosting

Boosting

Boosting is due to Schapire [215]. While bagging exploits data variation through a learner's instability, boosting tends to exploit it through a learner's *weakness*. A learner is weak if it generally induces models whose performance is only slightly better than random. Boosting is based on the observation that finding many rough rules of thumb (i.e., weak learning) can be a lot easier than finding a single, highly accurate prediction rule (i.e., strong learning). Boosting then assumes that a weak learner can be made strong by repeatedly running it on various distributions D_i over the training data T (i.e., varying the focus of the learner), and then combining the weak classifiers into a single composite classifier, as illustrated in Figure 5.3.

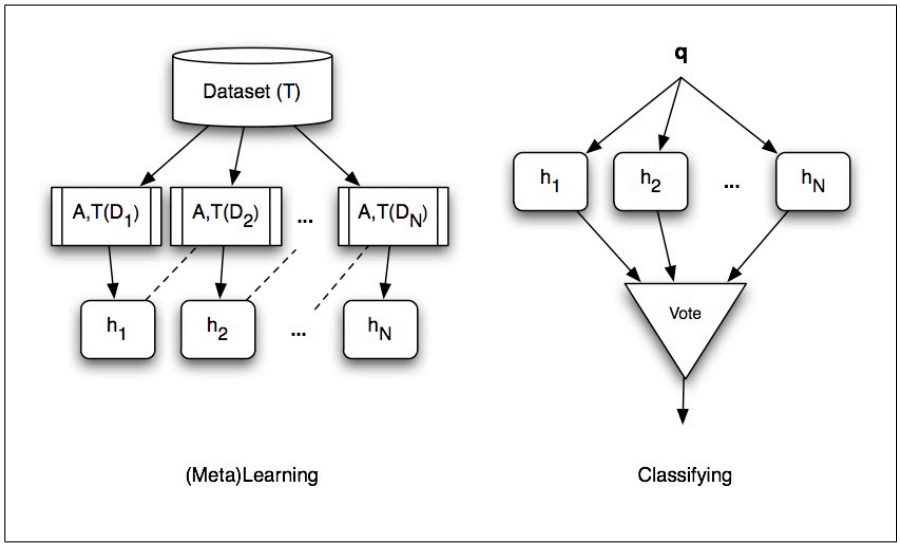

Fig. 5.3. Boosting

Unlike bagging, boosting tries actively to force the (weak) learning algorithm to change its induced model by changing the distribution over the training instances as a function of the errors made by previously generated models. The initial distribution D_1 over the dataset T is uniform, with each instance assigned a constant weight, i.e., probability of being selected for training, of

Algorithm AdaBoost.M1(T, A, N)

1. For $k = 1$ to $|T|$
2. $\quad D_1(x_k) = \frac{1}{|T|}$
3. For $i = 1$ to N
4. $\quad h_i =$ model induced by A from T with distribution D_i
5. $\quad \epsilon_i = \sum_{k:h_i(x_k) \neq y_k} D_i(x_k)$
6. $\quad$ If $\epsilon_i > .5$
7. $\quad\quad N = i - 1$
8. $\quad\quad$ Abort loop
9. $\quad \beta_i = \frac{\epsilon_i}{1-\epsilon_i}$
10. $\quad$ For $k = 1$ to $|T|$
11. $\quad\quad D_{i+1}(x_k) = \frac{D_i(x_k)}{Z_i} \times \begin{cases} \beta_i \text{ if } h_i(x_k) = y_k \\ 1 \quad \text{otherwise} \end{cases}$
12. For each new query instance q
13. $\quad \text{Class}(q) = argmax_{y \in \mathcal{Y}} \sum_{i:h_i(q)=y} log \frac{1}{\beta_i}$

where:
T is the training set
A is the chosen learning algorithm
N is the number of iterations to perform over T
$\mathcal{Y}$ is the finite set of target class values
Z_i is a normalization constant, chosen so that D_{i+1} is a distribution

Fig. 5.4. Boosting algorithm for classification (AdaBoost.M1)

$1/|T|$, and a first model is induced. At each subsequent iteration, the weights of misclassified instances are increased, thus focusing the next model's attention on them. This procedure goes on until either a fixed number of iterations has been performed or the total weight of the misclassified instances exceeds 0.5. The popular AdaBoost.M1 [101] boosting algorithm for classification is shown in Figure 5.4.

The class of a new query instance q is given by a weighted vote of the induced models. The case of regression is more complex. The regression version of AdaBoost, known as AdaBoost.R, is based on decomposition into infinitely many classes. The reader is referred to [100] for details.

Although the argument for boosting originated with weak learners, boosting may actually be successfully applied to any learner.

5.2 Stacking and Cascade Generalization

While bagging and boosting exploit variation in the data, stacking and cascade generalization exploit differences among learners. They make explicit two levels of learning: the base level where learners are applied to the task

at hand, and the meta level where a new learner is applied to data obtained from learning at the base level.

5.2.1 Stacking

Stacking

The idea of stacked generalization is due to Wolpert [284]. Stacking takes a number of learning algorithms $\{A_1, \ldots, A_N\}$ and runs them against the dataset T under consideration (i.e., base-level data) to produce a series of models $\{h_1, \ldots, h_N\}$. Then, a new dataset $\mathcal{T}$ is constructed by replacing the description of each instance in the base-level dataset by the predictions of each base-level model for that instance.[2] This new metadataset is in turn presented to a new learner A_{meta} that builds a metamodel h_{meta} mapping the predictions of the base-level learners to target classes, as illustrated in Figure 5.5. The stacking algorithm for classification is shown in Figure 5.6.

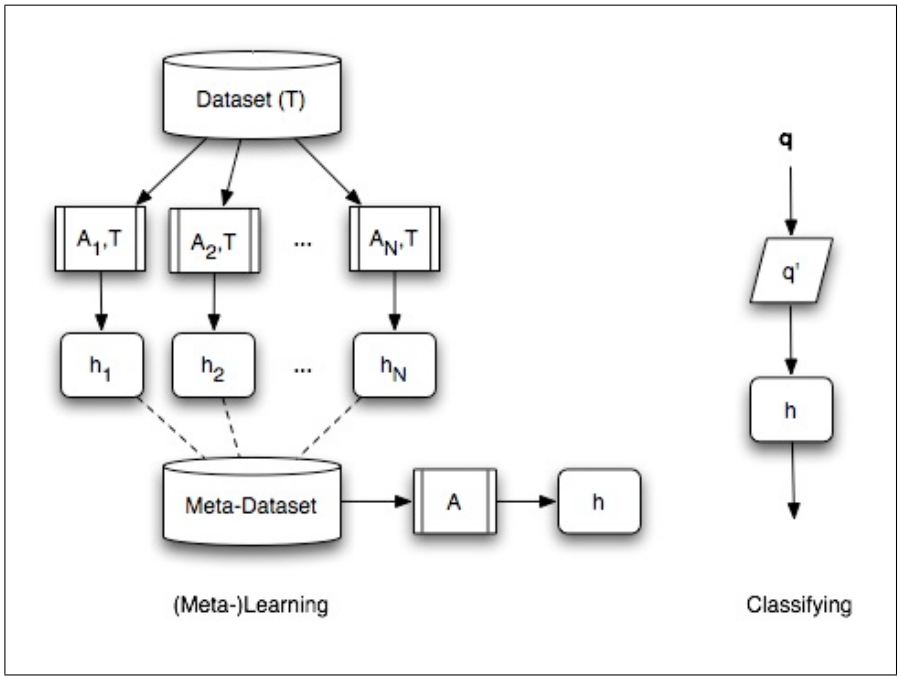

Fig. 5.5. Stacking

Meta-instance

A new query instance q is first run through all the base-level learners to compose the corresponding query meta-instance q', which serves as input to

[2] In some versions of stacking, the base-level description is not replaced by the predictions, but rather the predictions are appended to the base-level description, resulting in a kind of hybrid meta-example.

Algorithm Stacking(T, $\{A_1, \ldots, A_N\}$, A_{meta})
1. For $i = 1$ to N
2. h_i = model induced by A_i from T
3. $\mathcal{T} = \emptyset$
4. For $k = 1$ to $|T|$
5. $E_k =< h_1(x_k), h_2(x_k), \ldots, h_N(x_k), y_k >$
6. $\mathcal{T} = \mathcal{T} \cup \{E_k\}$
7. h_{meta} = model induced by A_{meta} from $\mathcal{T}$
8. For each new query instance q
9. Class$(q) = h_{meta}(< h_1(q), h_2(q), \ldots, h_N(q) >)$

where:
T is the base-level training set
N is the number of base-level learning algorithms
$\{A_1, \ldots, A_N\}$ is the set of base-level learning algorithms
A_{meta} is the chosen meta-level learner

Fig. 5.6. Stacking algorithm

the metamodel to produce the final classification for q.

Note that the base-level models' predictions in line 5 (Figure 5.6) are obtained by running each instance through the models induced from the base-level dataset (lines 1 and 2). Alternatively, more statistically reliable predictions could be obtained through cross-validation as proposed in [85]. In this case, lines 1 through 6 are replaced with the following:

1. For $i = 1$ to N
2. For $k = 1$ to $|T|$
3. $E_k[i] = h_i(x_k)$ obtained by cross-validation
4. $\mathcal{T} = \emptyset$
5. For $k = 1$ to $|T|$
6. $\mathcal{T} = \mathcal{T} \cup \{E_k\}$

A variation on stacking is proposed in [259], where the predictions of the base-level classifiers in the metadataset are replaced by class probabilities. A meta-level example thus consists of a set of N (the number of base-level learning algorithms) vectors of $m = |\mathcal{Y}|$ (the number of classes) coordinates, where p_{ij} is the posterior probability, as given by learning algorithm A_i, that the corresponding base-level example belongs to class j. Other forms of stacking, based on using partitioned data rather than full datasets, or using the same learning algorithm on multiple, independent data batches, have also been proposed (e.g., see [59, 260]).

The transformation applied to the base-level dataset, whether through the addition of predictions or class probabilities, is intended to give information about the behavior of the various base-level learners on each instance, and thus constitutes a form of metaknowledge.

5.2.2 Cascade Generalization

Cascade generalization

Gama and Brazdil proposed another model combination technique known as cascade generalization, that also exploits differences among learners [108]. In cascade generalization, the classifiers are used in sequence rather than in parallel as in stacking. Instead of the data from the base-level learners feeding into a single meta-level learner, each base-level learner A_{i+1} (except for the first one, i.e., $i > 0$) also acts as a kind of meta-level learner for the base-level learner A_i that precedes it. Indeed, the inputs to A_{i+1} consist of the inputs to A_i together with the class probabilities produced by h_i, the model induced by A_i. A single learner is used at each step and there is, in principle, no limit on the number of steps, as illustrated in Figure 5.7. The basic cascade generalization algorithm for two steps is shown in Figure 5.8.

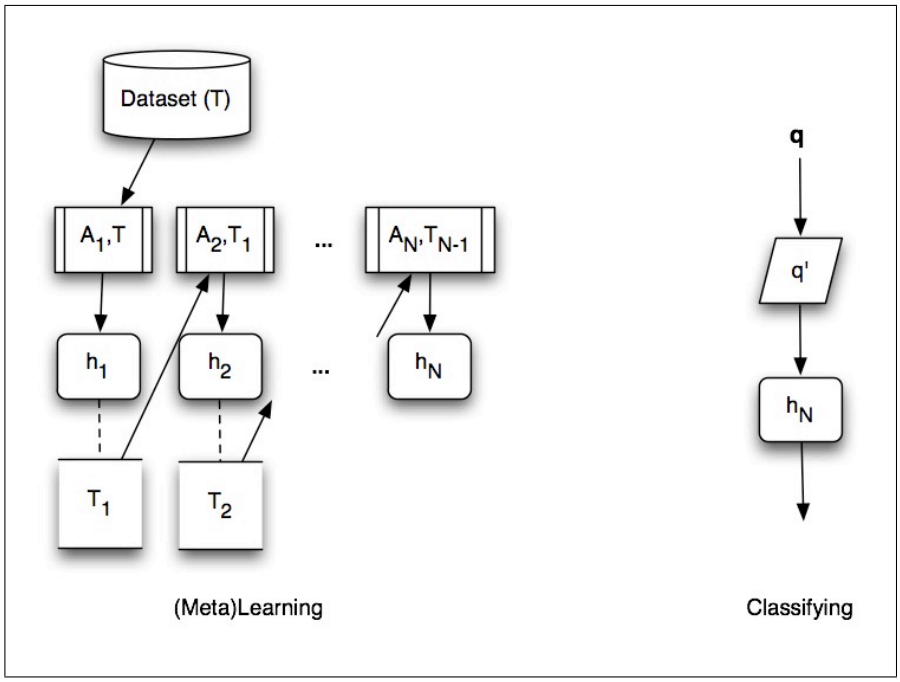

Fig. 5.7. Cascade generalization

This two-step algorithm is easily extended to an arbitrary number of steps — defined by the number of available classifiers — through successive invo-

Algorithm CascadeGeneralization($\{A_1, A_2\}$, T)
1. h_1 = model induced by A_1 from T
2. T_1 = ExtendDataset(h_1, T)
3. h_2 = model induced by A_2 from T_1
4. For each new query instance q
5. q' = ExtendDataset(h_1, $\{q\}$)
6. Class(q) = $h_2(q')$

where:
T is the original base level training set
A_1 and A_2 are base level learning algorithms

Algorithm ExtendDataset(h, T)
1. $newT = \emptyset$
2. For each $\mathbf{e} = (\mathbf{x}, y) \in T$
3. For $j = 1$ to $|\mathcal{Y}|$
4. p_j = probability that e belongs to y_j according to h
5. $e' = (\mathbf{x}, p_1, \ldots, p_{|\mathcal{Y}|}, y)$
6. $newT = newT \cup \{e'\}$
7. Return $newT$

where:
h is a model induced by a learning algorithm
T is the dataset to be extended with data generated from h
$\mathcal{Y}$ is the finite set of target class values

Fig. 5.8. Cascade generalization algorithm (two steps)

cation of the ExtendDataset function, as illustrated in Figure 5.9, where the recursive algorithm begins with $i = 1$.[3]

A new query instance q is first extended into a meta-instance q' as it gathers metadata through the steps of the cascade. The final classification is then given by the output of the last model in the cascade on q'.

5.3 Cascading and Delegating

Like stacking and cascade generalization, cascading and delegating exploit differences among learners. However, whereas the former produce multi-expert classifiers (all constituent base classifiers are used for classification), the latter

[3] To use this N-step version of cascade generalization for classification, it may be advantageous to implement it iteratively rather than recursively, so that intermediate models may be stored and used when extending new queries.

Algorithm CascadeGeneralizationN($\{A_1, \ldots, A_N\}$, T, i)
1. h = model induced by A_i from T
2. If ($i == N$)
3. Return h
4. $T' = \text{ExtendDataset}(h, T)$
5. CascadeGeneralizationN($\{A_1, \ldots, A_N\}$, T', $i + 1$)

where:
T is the original base-level training set
N is the number of steps in the cascade
$\{A_1, \ldots, A_N\}$ is the set of base-level learning algorithms

Fig. 5.9. Cascade generalization for arbitrary number of steps

produce multistage classifiers, in which not all base classifiers need be consulted when predicting the class of a new query instance. Hence, classification time is reduced.

5.3.1 Cascading

Cascading

Alpaydin and Kaynak [4, 140] developed the idea of cascading, which may be viewed as a kind of multilearner version of boosting. Like boosting, cascading varies the distribution over the training instances, here as a function of the confidence of the previously generated models.[4] Unlike boosting, however, cascading does not strengthen a single learner, but uses a small number of different classifiers of increasing complexity, in a cascade-like fashion, as shown in Figure 5.10.

The initial distribution D_1 over the dataset T is uniform, with each training instance assigned a constant weight of $1/|T|$, and a model h_1 is induced with the first base-level learning algorithm A_1. Then, each base-level learner A_{i+1} is trained from the same dataset T, but with a new distribution D_{i+1}, determined by the confidence of the base-level learner A_i that precedes it. The confidence of the model h_i, induced by A_i, on a training instance x is defined as $\delta_i(x) = \max_{y \in \mathcal{Y}} P(y|x, h_i)$. At step $i + 1$, the weights of instances whose classification is uncertain under h_i (i.e., below a predefined confidence threshold) are increased, thus making them more likely to be sampled when training A_{i+1}. Early classifiers are generally semi-parametric (e.g., multilayer perceptrons) and the final classifier is always non-parametric (e.g., k-nearest-neighbor). Thus, a cascading system can be viewed as creating rules, which

[4] This is a generalization of boosting's function of the errors of the previously generated models. Rather than biasing the distribution to only those instances the previous layers *misclassify*, cascading biases the distribution to those instances the previous layers are *uncertain* about.

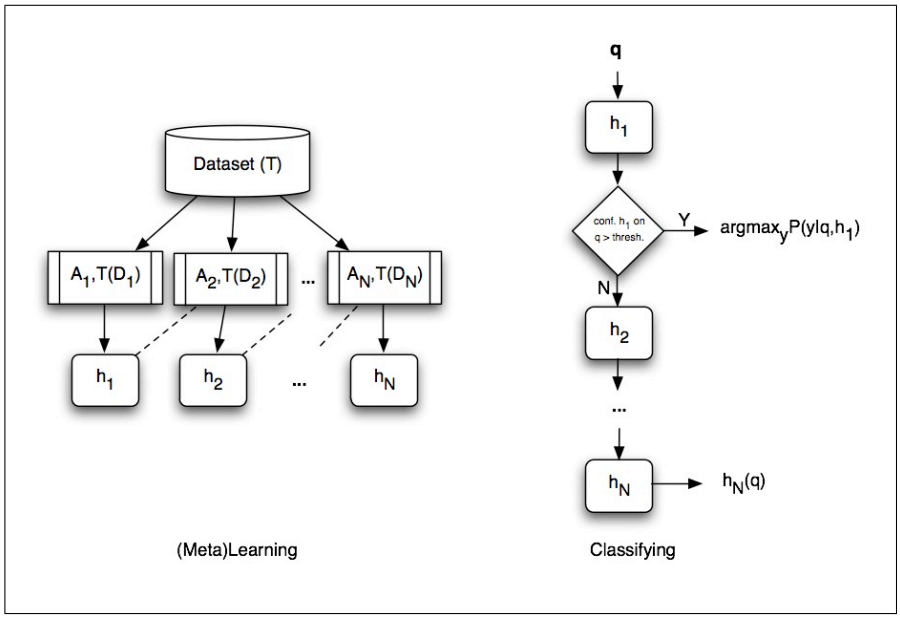

Fig. 5.10. Cascading

account for most instances, in the early steps, and catching exceptions at the final step. The generic cascading algorithm is shown in Figure 5.11.

When classifying a new query instance q, the system sends q to all of the models and looks for the first model, h_k, from 1 to N, whose confidence on q is above the confidence threshold. If h_k is an intermediate model in the cascade, the class of the new query instance is the class with highest probability (line 15, Figure 5.11). If h_k is the final (non-parametric) model in the cascade, the class of the new query instance is the output of $h_k(q)$ (line 13, Figure 5.11).

Although the weighted iterative approach is similar, cascading differs from boosting in several significant ways. First, cascading uses different learning algorithms at each step, thus increasing the variety of the ensemble. Second, the final k-NN step can be used to place a limit on the number of steps in the cascade, so that a small number of classifiers is used to reduce complexity. Finally, when classifying a new instance, there is no vote across the induced models; only one model is used to make the prediction.

5.3.2 Delegating

A cautious, delegating classifier is a classifier that provides classifications only for instances above a predefined confidence threshold, and passes (or delegates) other instances to another classifier. The idea of delegating classifiers comes from Ferri et al. [95]. It is similar in spirit to cascading. In cascading,

Delegating

Algorithm Cascading(T, $\{A_1, \ldots, A_N\}$)
1. For $k = 1$ to $|T|$
2. $D_1(x_k) = \frac{1}{|T|}$
3. For $i = 1$ to $N - 1$
4. h_i = model induced by A_i from T with distribution D_i
5. For $k = 1$ to $|T|$
6. $D_{i+1}(x_k) = \frac{1-\delta_i(x_k)}{\sum_{m=1}^{|T|} 1-\delta_i(x_m)}$
7. $h_N = k$-NN
8. For each new query instance q
9. i = 1
10. While $i < N$ and $\delta_i(q) < \Theta_i$
11. i = i + 1
12. If $i = N$ Then
13. Class(q) = $h_N(q)$
14. Else
15. Class(q) = $argmax_{y \in \mathcal{Y}} P(y|q, h_i)$

where:
T is the base-level training set
N is the number of base-level learning algorithms
$A_1, \ldots, A_N$ are the base-level learning algorithms
Θ_i is the confidence threshold associated with A_i, s.t. $\Theta_{i+1} \geq \Theta_i$
$\mathcal{Y}$ is the finite set of target class values
$\delta_i(x) = \max_{y \in \mathcal{Y}} P(y|x, h_i)$ is the confidence function for model h_i

Fig. 5.11. Cascading algorithm

however, all instances are (re-)weighted and processed at each step. In delegating, the next classifier is specialized to those instances for which the previous one lacks confidence, through training *only* on the delegated instances, as illustrated in Figure 5.12. The delegation stops either when there are no instances left to delegate or when a predefined number of delegation steps has been performed. The delegating algorithm is shown in Figure 5.13.

The function getThreshold(h, T) may be implemented in two different ways as follows:

- *Global Percentage.* $\tau = max\{t : |\{e \in T : h^{CONF}(e) > t| \geq \rho.|T|\}$, where ρ is a user-defined fraction.
- *Stratified Percentage.* For each class c, $\tau^c = max\{t : |\{e \in T_c : h^{PROB_c}(e) > t| \geq \rho.|T_c|\}$, where $h^{PROB_c}(e)$ is the probability of class c under model h for example e, and T_c is the set of examples of class c in T.

Note that there are actually four ways to compute the threshold, based on the value of the parameter *Rel*. When *Rel* is true (i.e., each threshold is computed relative to the examples delegated by the previous classifier), the approaches

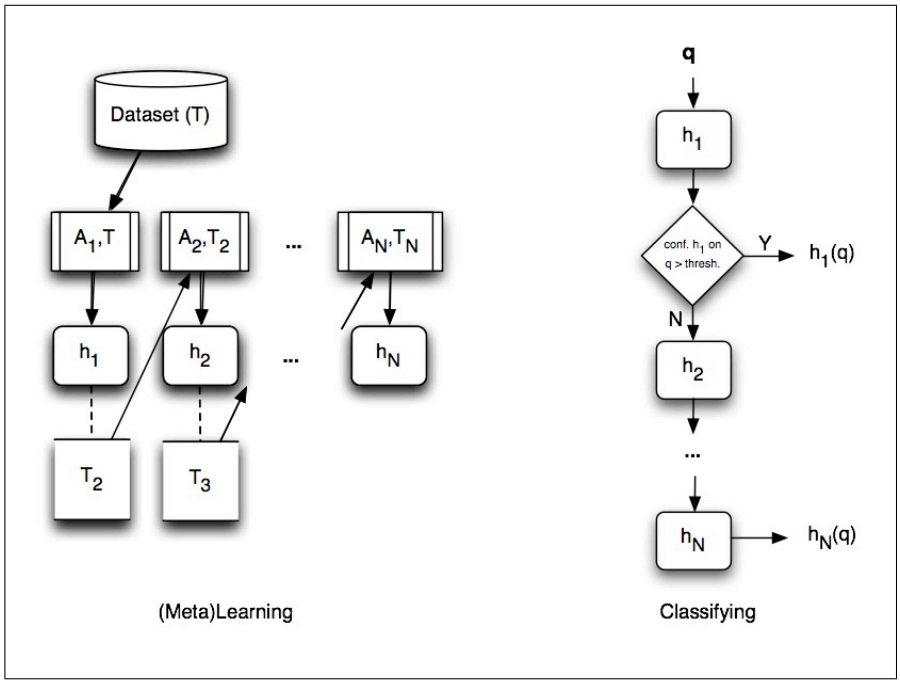

Fig. 5.12. Delegating

are called Global Relative Percentage and Stratified Relative Percentage, respectively; and when *Rel* is false, they are called Global Absolute Percentage and Stratified Absolute Percentage, respectively.

When classifying a new query instance q, the system first sends q to h_1 and produces an output for q based on one of several delegation mechanisms, generally taken from the following alternatives.

- Round-rebound (only applicable to two-stage delegation): h_1 defers to h_2 when its confidence is too low, but h_2 rebounds to h_1 when its own confidence is also too low.
- Iterative delegation: h_1 defers to h_2, which in turn defers to h_3, which in turn defers to h_4, and so on until a model h_k is found whose confidence on q is above threshold or h_N is reached. The algorithm of Figure 5.13 implements this mechanism (lines 14 to 16).

Delegation may be viewed as a generalization of divide-and-conquer methods (e.g., see [98, 103]), with a number of advantages including:

- Improved efficiency: each classifier learns from a decreasing number of examples,
- No loss of comprehensibility: there is no combination of models; each instance is classified by a single classifier, and

Algorithm Delegating(T, $\{A_1, \ldots, A_N\}$, N, Rel)
1. $T_1 = T$
2. $i = 0$
3. Repeat
4. $\quad i = i + 1$
5. $\quad h_i$ = model induced by A_i from T_i
6. $\quad$ If (Rel = True and $i > 1$) Then
7. $\quad\quad \tau_i$ = getThreshold(h_i, T_{i-1})
8. $\quad$ Else
9. $\quad\quad \tau_i$ = getThreshold(h_i, T)
10. $\quad T^{>}_{h_i} = \{e \in T_i : h_i^{CONF}(e) > \tau_i\}$
11. $\quad T^{\leq}_{h_i} = \{e \in T_i : h_i^{CONF}(e) \leq \tau_i\}$
12. $\quad T_{i+1} = T^{\leq}_{h_i}$
13. Until $T^{>}_{h_i} = \emptyset$ or $i > N$
14. For each new query instance q
15. $\quad m = \min_k\{h_k(q) \geq \tau_k\}$
16. $\quad$ Class(q) = $h_m(q)$

where:
T is the base-level training set
N is the maximum number of delegating stages
$A_1, \ldots, A_N$ are the base-level learning algorithms
$h_i^{CONF}(e)$ is the confidence of the prediction of model h_i for example e
Rel is a Boolean flag (true if τ_i is to be computed relative to delegated examples)
getThreshold(h, T) returns a confidence threshold for classifier h relative to T

Fig. 5.13. Delegating algorithm

- Possibility to simplify the overall multi-classifier: see for example the notion of grafting for decision trees [279].

5.4 Arbitrating

Arbitrating

A mechanism for combining classifiers by way of arbitration, originally introduced as Model Applicability Induction, has been proposed by Ortega et al. [185, 186].[5] As with delegating, the basic intuition behind arbitrating is that various classifiers have different areas of expertise (i.e., portions of the input space on which they perform well). However, unlike in delegating, where successive classifiers are specialized to instances for which previous classifiers lack confidence, all classifiers in arbitrating are trained on the full dataset T and specialization is performed at run time when a query instance is presented to the system. At that time, the classifier whose confidence is highest

[5] Interestingly, two other sets of researchers developed very similar arbitration mechanisms independently. See [153, 269].

in the area of input space close to the query instance is selected to produce the classification. The process is illustrated in Figure 5.14.

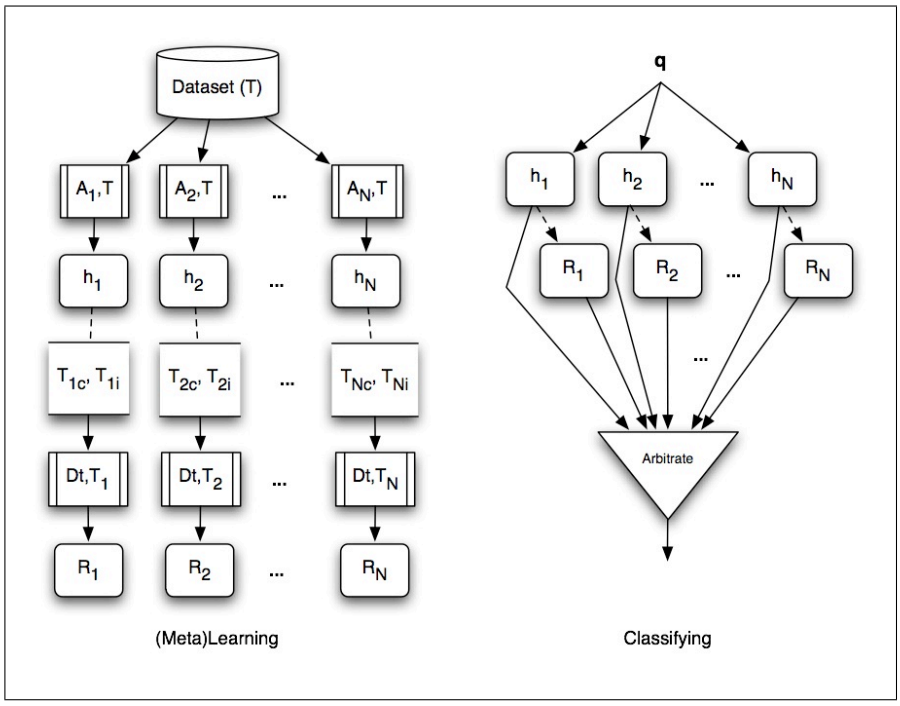

Fig. 5.14. Arbitrating

The area of expertise of each classifier is learned by its corresponding referee. The referee, although it can be any learned model, is typically a decision tree which predicts whether the associated classifier is correct or incorrect on some subset of the data, and with what reliability. The features used in building the referee decision tree consists of at least the primitive attributes that define the base-level dataset, possibly augmented by computed features (e.g., activation values of internal nodes in a neural network, conditions at various nodes in a decision tree) known as internal propositions, which assist in diagnosing examples for which the base-level classifier is unreliable (see [186] for details). The basic idea is that a referee holds meta-information on the area of expertise of its associated classifier, and can thus tell when that classifier reliably predicts the outcome. Several classifiers are then combined through an arbitration mechanism, in which the final prediction is that of the classifier whose referee is the most reliably correct. The arbitrating algorithm is shown in Figure 5.15.

Meta-information

Algorithm Arbitrating(T, $\{A_1, \ldots, A_N\}$)
1. For $i = 1$ to N
2. h_i = model induced by A_i from T
3. R_i = LearnReferee(h_i, T)
4. For each new query instance q
5. For $i = 1$ to N
6. c_i = correctness of h_i on q as per R_i
7. r_i = reliability of h_i on q as per R_i
8. $h^\star = argmax_{h_i : c_i is `correct'} r_i$
9. Class(q) = $h^\star(q)$

where:
T is the base-level training set
N is the number of base-level learning algorithms
$A_1, \ldots, A_N$ are the base-level learning algorithms
LearnReferee(A, T) returns a referee for learner A and dataset T

Function LearnReferee(h, T)
1. T_c = examples in T correctly classified by h
2. T_i = examples in T incorrectly classified by h
3. Select a set of features, including the attributes defining the examples and class, as well as additional features
4. Dt = pruned decision tree induced from T
5. For each leaf L in Dt
6. $N_c(L)$ = number of examples in T_c classified to L
7. $N_i(L)$ = number of examples in T_i classified to L
8. $r = \frac{max(|N_c(L), N_i(L)|)}{|N_c(L)| + |N_i(L)| + \frac{1}{2}}$
9. If $|N_c(L)| > |N_i(L)|$ Then
10. L's correctness is 'correct'
11. Else
12. L's correctness is 'incorrect'
13. Return Dt

Fig. 5.15. Arbitrating algorithm

Interestingly, the neural network community has also proposed techniques that employ referee functions to arbitrate among the predictions generated by several classifiers. These are generally known as Mixture of Experts (e.g., see [131, 132, 278]).

Finally, note that a different approach to arbitration was proposed by Chan and Stolfo [58, 59], where there is generally a unique arbiter for the entire set of N base-level classifiers. The arbiter is just another classifier learned by some learning algorithm on training examples that cannot be reliably predicted by the set of base-level classifiers. A typical rule for selecting training examples for

the arbiter is as follows: select example **e** if none of the target classes gather a majority vote (i.e., $> N/2$ votes) for **e**. The final prediction for a query example is then generally given by a plurality of votes on the predictions of the base-level classifiers and the arbiter, with ties being broken by the arbiter. An extension, involving the notion of an arbiter tree is also discussed, where several arbiters are built recursively in a tree-like structure. In this case, when a query example is presented, its prediction propagates upward in the tree from the leaves (base learners) to the root, with arbitration taking place at each level along the way.

5.5 Meta-decision Trees

Another approach to combining inductive models is found in the work of Todorovski and Dzeroski on meta-decision trees (MDTs) [264]. The general idea in MDT is similar to stacking in that a metamodel is induced from information obtained using the results of base-level learning, as shown in Figure 5.16. However, MDTs differ from stacking in the choice of what information to use, as well as in the metalearning task. In particular, MDTs build decision trees where each leaf node corresponds to a classifier rather than a classification. Hence, given a new query example, a meta-decision tree indicates the classifier that appears most suitable for predicting the example's class label. The MDT building algorithm is shown in Figure 5.17.

Meta-decision trees

Class distribution properties are extracted from examples using the base-level learners on different subsets of the data (lines 7 to 9, Figure 5.17). These properties, in turn, become the attributes of the metalearning task. Unlike metalearning for algorithm selection where these attributes are extracted from complete datasets (and thus there is one meta-example per dataset), MDTs have one meta-example per base-level example, simply substituting the base-level attributes with the new computed properties. The metamodel MDT is induced from these meta-examples, T_{MDT}, with a metalearning algorithm $\mathcal{A}$. Typically, $\mathcal{A}$ is MLC4.5, an extension of the well-known C4.5 decision tree learning algorithm [197].

Interestingly, in addition to improving accuracy, MDTs, being comprehensible, also provide some insight about base-level learning. In some sense, each leaf of the MDT captures the relative area of expertise of one of the base-level learners (e.g., C4.5, LTree, CN2, k-NN and Naïve Bayes).

5.6 Discussion

The list of methods presented in this chapter is not intended to be exhaustive. Methods included have been selected because they represent classes of model combination approaches and are most closely connected to the subject of metalearning. A number of so-called *ensemble* methods have been proposed

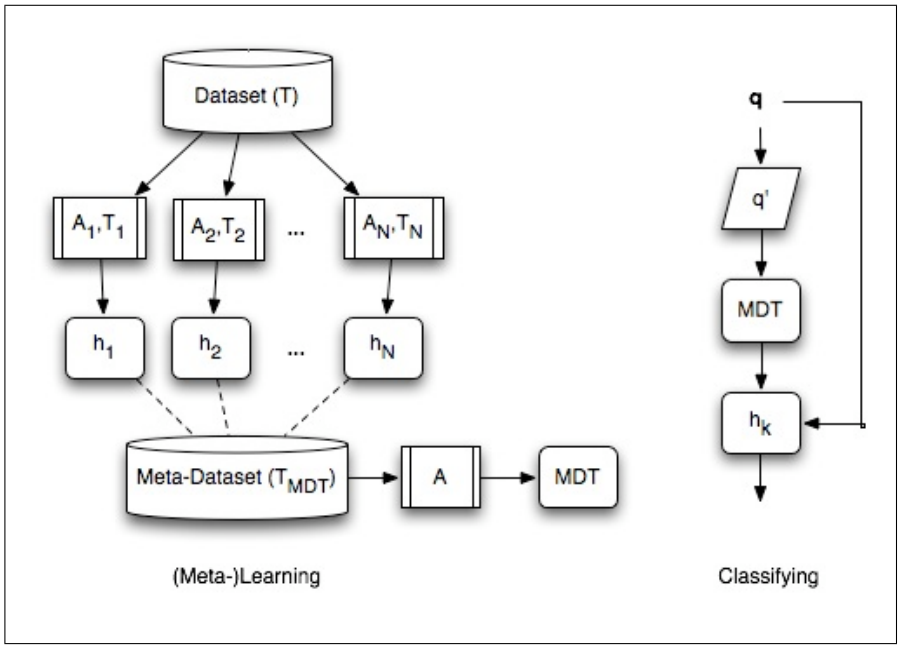

Fig. 5.16. Meta-decision tree

that combine many algorithms into a single learning system (e.g., see [143, 184, 52, 46]). The interested reader is referred to the literature for descriptions and evaluations of other combination and ensemble methods.

Because it uses results at the base level to construct a classifier at the meta level, model combination may clearly be regarded as a form of metalearning. However, its motivation is generally rather different from that of traditional metalearning. Whereas metalearning explicitly attempts to derive knowledge about the learning process itself, model combination focuses almost exclusively on improving base-level accuracy. Although they do learn at the meta level, most model combination methods fail to produce any real generalizable insight about learning, except in the case of arbitrating and meta-decision trees where new metaknowledge is explicitly derived in the combination process. As stated in [277], "by learning or explaining what causes a learning system to be successful or not on a particular task or domain, [metalearning seeks to go] beyond the goal of producing more accurate learners to the additional goal of understanding the conditions (e.g., types of example distributions) under which a learning strategy is most appropriate."

Algorithm MDTBuilding(T, $\{A_1, \ldots, A_N\}$, m)
1. $\{T_1, \ldots, T_m\}$ = StratifiedPartition(T, m)
2. $T_{MDT} = \emptyset$
3. For $i = 1$ to m
4. For $j = 1$ to N
5. h_j = model induced by A_j from $T - T_i$
6. For each $x \in T_i$
7. $maxprob(x) = \max_{y \in \mathcal{Y}} P_{h_j}(y|x)$
8. $entropy(x) = -\sum_{y \in \mathcal{Y}} P_{h_j}(y|x) \log P_{h_j}(y|x)$
9. $weight(x)$ = fraction of training examples used by h_j to estimate the class distribution of x
10. $E_j(x) =< maxprob(x), entropy(x), weight(x) >$
11. $E_j = \cup_{x \in T_i} E_j(x)$
12. $T_{MDT} = T_{MDT} \cup \text{join}_{j=1}^{N} E_j$
13. MDT = model induced by MLC4.5 from T_{MDT}
14. Return MDT
15. For each new query instance q
16. Class(q) = $MDT(< E_1(q), E_2(q), \ldots, E_N(q) >)$

where:
T is the base-level training set
N is the number of base-level learning algorithms
$A_1, \ldots, A_N$ are the base-level learning algorithms
m is the number of disjoint subsets into which T is partitioned
StratifiedPartition(T, m) returns a stratified partition of T into m equally-sized subsets

Fig. 5.17. Meta-decision tree building algorithm