Praktika5 Attribute Selection

11.8 ATTRIBUTE SELECTION

Figure 11.33 shows that part of Weka's attribute selection panel where you specify the attribute evaluator and search method; Tables 11.9 and 11.10 list the choices. Attribute selection is normally done by searching the space of attribute subsets, evaluating each one (see Section 7.1, page 307). This is achieved by combining 1 of the 6 attribute subset evaluators in Table 11.9 with 1 of the 10 search methods in Table 11.10. A potentially faster but less accurate approach is to evaluate the attributes individually and sort them, discarding attributes that fall below a chosen cutoff point. This is achieved by selecting one of the 11 single-attribute evaluators in Table 11.9 and using the ranking method in Table 11.10. The Weka interface allows both possibilities by letting the user choose a selection method from Table 11.9 and a search method from Table 11.10, producing an error message if you select an inappropriate combination.

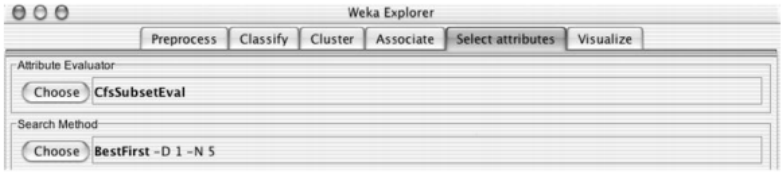

FIGURE 11.33

Attribute selection: specifying an evaluator and a search method.

The status line refers you to the error log for the message (see the end of Section 11.1). Attribute Subset Evaluators Subset evaluators take a subset of attributes and return a numerical measure that guides the search. They are configured like any other Weka object. CfsSubsetEval assesses the predictive ability of each attribute individually and the degree of redundancy among them, preferring sets of attributes that are highly correlated with the class but with low intercorrelation (see Section 7.1, page 310). An option iteratively adds attributes that have the highest correlation with the class, provided that the set does not already contain an attribute whose correlation with the attribute in question is even higher. Missing can be treated as a separate value, or its counts can be distributed among other values in proportion to their frequency. ConsistencySubsetEval evaluates attribute sets by the degree of consistency in class values when the training instances are projected onto the set. The consistency of any subset of attributes can never improve on that of the full set, so this evaluator is usually used in conjunction with a random or exhaustive search that seeks the smallest subset with a consistency that is the same as that of the full attribute set. Whereas the previously mentioned subset

evaluators are filter methods of attribute selection (see Section 7.1, page 309), ClassifierSubsetEval and WrapperSubsetEval are wrapper methods. ClassifierSubsetEval uses a classifier, specified in the object editor as a parameter, to evaluate sets of attributes on the training data or on a separate holdout set. WrapperSubsetEval also uses a classifier to evaluate attribute sets, but it employs cross-validation to estimate the accuracy of the learning scheme for each set. The remaining two subset evaluators are meta-evaluators—that is, they augment a base subset evaluator with preprocessing options. CostSensitiveSubsetEval takes a base subset evaluator and makes it cost sensitive by weighting or resampling the training data according to a supplied cost matrix. FilteredSubsetEval applies a filter to the training data before attribute selection is performed. Selecting a filter that alters the number or ordering of the original attributes generates an error message.

Table 11.9 Attribute Evaluation Methods for Attribute Selection

	Name	Function
Attribute Subset Evaluator	*CfsSubsetEval*	Consider predictive value of each attribute individually, along with the degree of redundancy among them
	ClassifierSubsetEval	Use a classifier to evaluate the attribute set
	ConsistencySubsetEval	Project training set onto attribute set and measure consistency in class values
	CostSensitiveSubsetEval	Makes its base subset evaluator cost sensitive
	FilteredSubsetEval	Apply a subset evaluator to filtered data
	WrapperSubsetEval	Use a classifier plus cross-validation
Single-Attribute Evaluator	*ChiSquaredAttributeEval*	Compute the chi-squared statistic of each attribute with respect to the class
	CostSensitiveAttributeEval	Make its base attribute evaluator cost sensitive
	FilteredAttributeEval	Apply an attribute evaluator to filtered data
	GainRatioAttributeEval	Evaluate attribute based on gain ratio
	InfoGainAttributeEval	Evaluate attribute based on information gain
	LatentSemanticAnalysis	Perform a latent semantic analysis and transformation
	OneRAttributeEval	Use *OneR*'s methodology to evaluate attributes
	PrincipalComponents	Perform principal components analysis and transformation
	ReliefFAttributeEval	Instance-based attribute evaluator
	SVMAttributeEval	Use a linear support vector machine to determine the value of attributes
	SymmetricalUncertAttributeEval	Evaluate attribute based on symmetrical uncertainty

Table 11.10 Search Methods for Attribute Selection

	Name	Function
Search Method	*BestFirst*	Greedy hill climbing with backtracking
	ExhaustiveSearch	Search exhaustively
	GeneticSearch	Search using a simple genetic algorithm
	GreedyStepwise	Greedy hill climbing without backtracking; optionally generate ranked list of attributes
	LinearForwardSelection	Extension of *BestFirst* that considers a restricted number of the remaining attributes when expanding the current point in the search
	RaceSearch	Use race search methodology
	RandomSearch	Search randomly
	RankSearch	Sort the attributes and rank promising subsets using an attribute subset evaluator
	ScatterSearchV1	Search using an evolutionary scatter search algorithm
	SubsetSizeForwardSelection	Extension of *LinearForwardSelection* that performs an internal cross-validation in order to determine the optimal subset size
Ranking Method	*Ranker*	Rank individual attributes (not subsets) according to their evaluation

Single-Attribute Evaluators

Single-attribute evaluators are used with the Ranker search method to generate a ranked list from which Ranker discards a given number (explained in the next section). They can also be used in the RankSearch method. ReliefFAttributeEval is instancebased: It samples instances randomly and checks neighboring instances of the same and different classes (see Section 7.1, page 310). It operates on discrete and continuous class data. Parameters specify the number of instances to sample, the number of neighbors to check, whether to weight neighbors by distance, and an exponential function that governs how rapidly weights decay with distance. InfoGainAttributeEval evaluates attributes by measuring their information gain with respect to the

class. It discretizes numeric attributes first using the MDL-based discretization method (it can be set to binarize them instead). This method, along with the next three, can treat missing as a separate value or distribute the counts among other values in proportion to their frequency. ChiSquaredAttributeEval evaluates attributes by computing the chi-squared statistic with respect to the class. GainRatioAttributeEval evaluates attributes by measuring their gain ratio with respect to the class. SymmetricalUncertAttributeEval evaluates an attribute by measuring its symmetrical uncertainty with respect to the class (see Section 7.1, page 310). OneRAttributeEval uses the simple accuracy measure adopted by the OneR classifier. It can use the training data for evaluation, as OneR does, or it can apply internal cross-validation: The number of folds is a parameter. It adopts OneR's simple discretization method: The minimum bucket size is a parameter. SVMAttributeEval evaluates attributes using recursive feature elimination with a linear support vector machine (see Section 7.1, page 309). Attributes are selected one by one based on the size of their coefficients, relearning after each one. To speed things up a fixed number (or proportion) of attributes can be removed at each stage. Indeed, a proportion can be used until a certain number of attributes remain, thereupon switching to the fixed-number method—rapidly eliminating many attributes and then considering each remaining one more intensively. Various parameters are passed on to the support vector machine: complexity, epsilon, tolerance, and the filtering method used. Unlike other single-attribute evaluators, PrincipalComponents and LatentSemanticAnalysis transform the set of attributes. In the case of PrincipalComponents, the new attributes are ranked in order of their eigenvalues (see Section 7.3, page 324). Optionally, a subset is selected by choosing sufficient eigenvectors to account for a given proportion of the variance (95% by default). Finally, the reduced data can be transformed back to the original space. LatentSemanticAnalysis applies a singular value decomposition to the training data. Singular value decomposition is related to principal components analysis— both produce directions that are linear combinations of the original attribute values— but differs in that it is computed from a matrix containing the original data values rather than the attribute correlation or covariance matrix. Selecting the k directions with the highest singular values gives a rank k approximation to the original data matrix. Latent semantic analysis is so named because of its application to text mining, where instances represent documents and attributes represent the terms that occur in them. In some sense, the directions that the technique produces can be thought of as merging terms with similar meaning. LatentSemanticAnalysis allows the user to specify the number of directions to extract (i.e., the rank) and whether or not the data is normalized before the analysis is performed. The remaining two attribute evaluators, CostSensitiveAttributeEval and FilteredAttributeEval, are meta-evaluators: They are the single-attribute versions of their subset-based counterparts described earlier. The former augments a base evaluator by weighting or resampling the training data according to a cost matrix; the latter

applies a filter to the training data before the base evaluator is applied. Search methods Search methods traverse the attribute space to find a good subset. Quality is measured by the chosen attribute subset evaluator. Each search method can be configured with Weka's object editor. BestFirst performs greedy hill climbing with backtracking; you can specify how many consecutive nonimproving nodes must be encountered before the system backtracks. It can search forward from the empty set of attributes, backward from the full set, or start at an intermediate point (specified by a list of attribute indexes) and search in both directions by considering all possible single-attribute additions and deletions. Subsets that have been evaluated are cached for efficiency; the cache size is a parameter. GreedyStepwise searches greedily through the space of attribute subsets. Like BestFirst, it may progress forward from the empty set or backward from the full set. Unlike BestFirst, it does not backtrack but terminates as soon as adding or deleting the best remaining attribute decreases the evaluation metric. In an alternative mode, it ranks attributes by traversing the space from empty to full (or vice versa) and recording the order in which attributes are selected. You can specify the number of attributes to retain or set a threshold below which attributes are discarded. LinearForwardSelection and SubsetSizeForwardSelection are extensions of BestFirst aimed at, respectively, reducing the number of evaluations performed during the search and producing a compact final subset (Gutlein et al., 2009). LinearForwardSelection limits the number of attribute expansions in each forward selection step. There are two modes of operation; both begin by ranking the attributes individually using a specified subset evaluator. In the first mode, called fixed set, a forward best-first search is performed on just the k top-ranked attributes. In the second mode, called fixed width, the search considers expanding the best subset at each step by selecting an attribute from the k top-ranked attributes. However, rather than shrinking the available pool of attributes after every expansion, it is held at a constant size k by adding further attributes from the initial ranked list (so long as any remain). The mode of operation and the value of k are parameters. Like BestFirst, you can set the degree of backtracking, the size of the lookup cache, and a list of attributes to begin the search. As well as the standard forward search, LinearForwardSelection offers an option to perform a floating forward search, which considers a number of consecutive single-attribute elimination steps after each forward step—so long as this results in an improvement. SubsetSizeForwardSelection extends LinearForwardSelection with a process to determine the optimal subset size. This is achieved by performing an m-fold crossvalidation on the training data. LinearForwardSelection is applied m times—once for each training set in the cross-validation. A given test fold is used to evaluate each size of subset explored by LinearForwardSelection in its corresponding training set. The performance for each size of subset is then averaged over the folds. Finally, LinearForwardSelection is performed on all the data to find a subset of that optimal size. As well as the options provided by LinearForwardSelection, the

evaluator to use in determining the optimal subset size can be specified, along with the number of folds. When paired with wrapper-based evaluation, both LinearForwardSelection and SubsetSizeForwardSelection have been shown to combat the overfitting that can occur when standard forward selection or best-first searches are used with wrappers. Moreover, both select smaller final subsets than standard forward selection and bestfirst, while maintaining comparable accuracy (provided k is chosen to be sufficiently large). LinearForwardSelection is faster than standard forward selection and bestfirst selection. RaceSearch, used with ClassifierSubsetEval, calculates the cross-validation error of competing attribute subsets using race search (see Section 7.1). The four different searches described on page 313 are implemented: forward selection, backward elimination, schemata search, and rank racing. In the last case, a separate attribute evaluator (which can also be specified) is used to generate an initial ranking. Using forward selection, it is also possible to generate a ranked list of attributes by continuing racing until all attributes have been selected: The ranking is set to the order in which they are added. As with GreedyStepwise, you can specify the number of attributes to retain or set a threshold below which attributes are discarded. GeneticSearch uses a simple genetic algorithm (Goldberg, 1989). Parameters include population size, number of generations, and probabilities of crossover and mutation. You can specify a list of attribute indexes as the starting point, which becomes a member of the initial population. Progress reports can be generated every so many generations. RandomSearch randomly searches the space of attribute subsets. If an initial set is supplied, it searches for subsets that improve on (or equal) the starting point and have fewer (or the same number of) attributes. Otherwise, it starts from a random point and reports the best subset found. Placing all attributes in the initial set yields Liu and Setiono's (1996) probabilistic feature-selection algorithm. You can determine the fraction of the search space to explore. ExhaustiveSearch searches through the space of attribute subsets, starting from the empty set, and reports the best subset found. If an initial set is supplied, it searches backward from this starting point and reports the smallest subset with a better (or equal) evaluation. RankSearch sorts attributes using a single-attribute evaluator and then ranks promising subsets using an attribute subset evaluator. The latter was specified earlier in the top box of Figure 11.33, as usual; the attribute evaluator is specified as a property in RankSearch's object editor. It starts by sorting the attributes with the single-attribute evaluator and then evaluates subsets of increasing size using the subset evaluator—the best attribute, the best attribute plus the next best one, and so on—reporting the best subset. This procedure has low computational complexity: The number of times both evaluators are called is linear in the number of attributes.

Using a simple single-attribute evaluator (e.g., GainRatioAttributeEval), the selection procedure is very fast. ScatterSearchV1 uses an evolution-based scatter search algorithm (Laguna and Marti, 2003). Parameters include the population, size, random number seed, and strategy used to generate new population members from existing ones. Finally, we describe Ranker, which as noted earlier is not a search method for attribute subsets but a ranking scheme for individual attributes. It sorts attributes by their individual evaluations and must be used in conjunction with one of the singleattribute evaluators in the lower part of Table 11.9—not an attribute subset evaluator. Ranker not only ranks attributes but also performs attribute selection by removing the lower-ranking ones. You can set a cutoff threshold below which attributes are discarded, or specify how many attributes to retain. You can specify certain attributes that must be retained regardless of their rank.