Attribute Selection - Witten et al. (2011: Ch 7)

1

2

3---- So far, we have studied 4 machine learning methods: decision trees, knn, NB, linear classifiers, and shortly we wll see unsupervised classification clustering algorithms. All are sound, robust techniques that are eminently applicable to practical problems. But successful ML involves far more than selecting a learning algorithm and running it over your data. For one thing, many learning methods have various parameters, and suitable values must be chosen for these. In most cases, results can be improved markedly by suitable choice of parameter values, and the appropriate choice depends on the data at hand. For example, decision trees can be pruned or unpruned, and in the former case a pruning parameter may have to be chosen. In the k-nearest-neighbor method of instance-based learning, a value for k will have to be chosen. More generally, the learning scheme itself will have to be chosen from the range of schemes that are available. In all cases, the right choices depend on the data itself. It is tempting to try out several learning schemes, and several parameter values, on your data and see which works best. But be careful! The best choice is not necessarily the one that performs best on the training data. We have repeatedly cautioned about the problem of overfitting, where a learned model is too closely tied to the particular training data from which it was built. It is incorrect to assume that performance on the training data faithfully represents the level of performance that can be expected on the fresh data to which the learned model will be applied in practice. Fortunately, we have already encountered the solution to this problem in Lecture 4. There are two good methods for estimating the expected true performance of a learning scheme: 1) the use of a large dataset that is quite separate from the training data, in the case of plentiful data, 2) and cross-validation, if data is scarce. In the latter case, a single 10-fold crossvalidation is typically used in practice, although to obtain a more reliable estimate the entire procedure should be repeated 10 times. Once suitable parameters have been chosen for the learning scheme, use the whole training set—all the available training instances—to produce the final learned model that is to be applied to fresh data. Note that the performance obtained with the chosen parameter value during the tuning process is not a reliable estimate of the final model's performance, because the final model potentially overfits the data that was used for tuning. To ascertain how well it will perform, you need yet another large dataset that is quite separate from any data used during learning and tuning. The same is true for cross-validation: you need an "inner" cross-validation for parameter tuning and an "outer" cross-validation for error estimation. With 10-fold crossvalidation, this involves running the learning scheme 100 times. To summarize: when assessing the performance of a learning scheme, any parameter tuning that goes on should be treated as though it were an integral part of the training process. There are other important processes that can materially improve success when applying machine learning techniques to practical data mining problems, and these are the subject of this chapter. They constitute a kind of data engineering: engineering the input data into a form suitable for the learning scheme chosen and engineering the output model to make it more effective. You can look on them as a bag of tricks that you can apply to practical data mining problems to enhance the chance of success. Sometimes they work; other times they don't—and at the present state of the art, it's hard to say in advance whether

they will or not. In an area such as this where trial and error is the most reliable guide, it is particularly important to be resourceful and understand what the tricks are. We begin by examining four different ways in which the input can be massaged to make it more amenable for learning methods: attribute selection, attribute discretization, data transformation, and data cleansing. Consider the first, attribute selection. In many practical situations there are far too many attributes for learning schemes to handle, and some of them—perhaps the overwhelming majority—are clearly irrelevant or redundant. Consequently, the data must be preprocessed to select a subset of the attributes to use in learning. Of course, learning methods themselves try to select attributes appropriately and ignore irrelevant or redundant ones, but in practice their performance can frequently be improved by preselection. For example, experiments show that adding useless attributes causes the performance of learning schemes such as decision trees and rules, linear regression, instance-based learners, and clustering methods to deteriorate. Discretization of numeric attributes is absolutely essential if the task involves numeric attributes but the chosen learning method can only handle categorical ones. Even methods that can handle numeric attributes often produce better results, or work faster, if the attributes are prediscretized. The converse situation, in which categorical attributes must be represented numerically, also occurs (although less often); and we describe techniques for this case, too. Data transformation covers a variety of techniques. Unclean data plagues data mining. We emphasized the necessity of getting to know your data: understanding the meaning of all the different attributes, the conventions used in coding them, the significance of missing values and duplicate data, measurement noise, typographical errors, and the presence of systematic errors—even deliberate ones. Various simple visualizations often help with this task. There are also automatic methods of cleansing data, of detecting outliers, and of spotting anomalies, which we describe. Having studied how to massage the input, we turn to the question of engineering the output from machine learning schemes. In particular, we examine techniques for combining different models learned from the data. There are some surprises in store. For example, it is often advantageous to take the training data and derive several different training sets from it, learn a model from each, and combine the resulting models! Indeed, techniques for doing this can be very powerful. It is, for example, possible to transform a relatively weak learning method into an extremely strong one (in a precise sense that we will explain). Moreover, if several learning schemes are available, it may be advantageous not to choose the best-performing one for your dataset (using crossvalidation) but to use them all and combine the results. Finally, the standard, obvious way of modeling a multiclass learning situation as a two-class one can be improved using a simple but subtle technique. Many of these results are counterintuitive, at least at first blush. How can it be a good idea to use many different models together? How can you possibly do better than choose the model that performs best? Surely all this runs counter to Occam's razor, which advocates simplicity.How can you possibly obtain firstclass performance by combining indifferent models, as one of these techniques appears to do? But consider committees of humans, which often come up with wiser decisions than individual experts. Recall Epicurus's view that, faced with alternative explanations, one should retain them all. Imagine a group of specialists each of whom excels in a limited domain even though none is competent across the board. In struggling to understand how these methods work, researchers have exposed all sorts of connections and links that have led to even greater improvements.

4--- Most machine learning algorithms are designed to learn which are the most appropriate attributes to use for making their decisions. For example, decision tree

methods choose the most promising attribute to split on at each point and should—in theory—never select irrelevant or unhelpful attributes. Having more features should surely—in theory—result in more discriminating power, never less. "What's the difference between theory and practice?" an old question asks. "There is no difference," the answer goes, "—in theory. But in practice, there is." Here there is, too: in practice, adding irrelevant or distracting attributes to a dataset often "confuses" machine learning systems. Experiments with a decision tree learner (C4.5) have shown that adding to standard datasets a random binary attribute generated by tossing an unbiased coin affects classification performance, causing it to deteriorate (typically by 5% to 10% in the situations tested). This happens because at some point in the trees that are learned the irrelevant attribute is invariably chosen to branch on, causing random errors when test data is processed. How can this be, when decision tree learners are cleverly designed to choose the best attribute for splitting at each node? The reason is subtle. As you proceed further down the tree, less and less data is available to help make the selection decision. At some point, with little data, the random attribute will look good just by chance. Because the number of nodes at each level increases exponentially with depth, the chance of the rogue attribute looking good somewhere along the frontier multiplies up as the tree deepens. The real problem is that you inevitably reach depths at which only a small amount of data is available for attribute selection. If the dataset were bigger it wouldn't necessarily help—you'd probably just go deeper. Divide-and-conquer tree learners and separate-and-conquer rule learners both suffer from this effect because they inexorably reduce the amount of data on which they base judgments. Instance-based learners are very susceptible to irrelevant attributes because they always work in local neighborhoods, taking just a few training instances into account for each decision. Indeed, it has been shown that the number of training instances needed to produce a predetermined level of performance for instance-based learning increases exponentially with the number of irrelevant attributes present. Naïve Bayes, by contrast, does not fragment the instance space and robustly ignores irrelevant attributes. It assumes by design that all attributes are independent of one another, an assumption that is just right for random "distracter" attributes. But through this very same assumption,Naïve Bayes pays a heavy price in other ways because its operation is damaged by adding redundant attributes. The fact that irrelevant distracters degrade the performance of state-of-theart decision tree and rule learners is, at first, surprising. Even more surprising is that relevant attributes can also be harmful. For example, suppose that in a two-class dataset a new attribute were added which had the same value as the class to be predicted most of the time (65%) and the opposite value the rest of the time, randomly distributed among the instances. Experiments with standard datasets have shown that this can cause classification accuracy to deteriorate (by 1% to 5% in the situations tested). The problem is that the new attribute is (naturally) chosen for splitting high up in the tree. This has the effect of fragmenting the set of instances available at the nodes below so that other choices are based on sparser data. Because of the negative effect of irrelevant attributes on most machine learning schemes, it is common to precede learning with an attribute selection stage that strives to eliminate all but the most relevant attributes. The best way to select relevant attributes is manually, based on a deep understanding of the learning problem and what the attributes actually mean. However, automatic methods can also be useful. Reducing the dimensionality of the data by deleting unsuitable attributes improves the performance of learning algorithms. It also speeds them up, although this may be outweighed by the computation involved in attribute selection. More importantly, dimensionality reduction

yields a more compact, more easily interpretable representation of the target concept, focusing the user's attention on the most relevant variables.

5---- Some classification and clustering algorithms deal with nominal attributes only and cannot handle ones measured on a numeric scale. To use them on general datasets, numeric attributes must first be "discretized" into a small number of distinct ranges. Even learning algorithms that do handle numeric attributes sometimes process them in ways that are not altogether satisfactory. Statistical clustering methods often assume that numeric attributes have a normal distribution— often not a very plausible assumption in practice—and the standard extension of the Naïve Bayes classifier to handle numeric attributes adopts the same assumption. Although most decision tree and decision rule learners can handle numeric attributes, some implementations work much more slowly when numeric attributes are present because they repeatedly sort the attribute values. For all these reasons the question arises: what is a good way to discretize numeric attributes into ranges before any learning takes place? We have already encountered some methods for discretizing numeric attributes. The 1R learning scheme described in Chapter 4 uses a simple but effective technique: sort the instances by the attribute's value and assign the value into ranges at the points that the class value changes—except that a certain minimum number of instances in the majority class (six) must lie in each of the ranges, which means that any given range may include a mixture of class values. This is a "global" method of discretization that is applied to all continuous attributes before learning starts. Decision tree learners, on the other hand, deal with numeric attributes on a local basis, examining attributes at each node of the tree when it is being constructed to see whether they are worth branching on—and only at that point deciding on the best place to split continuous attributes. Although the treebuilding method we examined in Chapter 6 only considers binary splits of continuous attributes, one can imagine a full discretization taking place at that point, yielding a multiway split on a numeric attribute. The pros and cons of the local versus the global approach are clear. Local discretization is tailored to the actual context provided by each tree node, and will produce different discretizations of the same attribute at different places in the tree if that seems appropriate.However, its decisions are based on less data as tree depth increases, which compromises their reliability. If trees are developed all the way out to single-instance leaves before being pruned back, as with the normal technique of backward pruning, it is clear that many discretization decisions will be based on data that is grossly inadequate. When using global discretization before applying a learning method, there are two possible ways of presenting the discretized data to the learner. The most obvious is to treat discretized attributes like nominal ones: each discretization interval is represented by one value of the nominal attribute. However, because a discretized attribute is derived from a numeric one, its values are ordered, and treating it as nominal discards this potentially valuable ordering information. Of course, if a learning scheme can handle ordered attributes directly, the solution is obvious: each discretized attribute is declared to be of type "ordered." If the learning method cannot handle ordered attributes, there is still a simple way of enabling it to exploit the ordering information: transform each discretized attribute into a set of binary attributes before the learning scheme is applied. Assuming the discretized attribute has k values, it is transformed into k - 1 binary attributes, the first i - 1 of which are set to false whenever the ith value of the discretized attribute is present in the data and to true otherwise. The remaining attributes are set to false. In other words, the (i - 1)th binary attribute represents whether the discretized attribute is less than i. If a decision tree learner splits on this attribute, it implicitly uses the ordering

information it encodes. Note that this transformation is independent of the particular discretization method being applied: it is simply a way of coding an ordered attribute using a set of binary attributes.

6--- There is a converse problem to discretization. Some learning algorithms— notably the nearest-neighbor instance-based method and numeric prediction techniques involving regression—naturally handle only attributes that are numeric. How can they be extended to nominal attributes? In instance-based learning (ex kNN), discrete attributes can be treated as numeric by defining the "distance" between two nominal values that are the same as 0 and between two values that are different as 1—regardless of the actual values involved. Rather than modifying the distance function, this can be achieved using an attribute transformation: replace a k-valued nominal attribute with k synthetic binary attributes, one for each value indicating whether the attribute has that value or not. If the attributes have equal weight, this achieves the same effect on the distance function. The distance is insensitive to the attribute values because only "same" or "different" information is encoded, not the shades of difference that may be associated with the various possible values of the attribute. More subtle distinctions can be made if the attributes have weights reflecting their relative importance. If the values of the attribute can be ordered, more possibilities arise.

7--- Resourceful data miners have a toolbox full of techniques, such as discretization, for transforming data. Data mining is hardly ever a matter of simply taking a dataset and applying a learning algorithm to it. Every problem is different. You need to think about the data and what it means, and examine it from diverse points of view—creatively!—to arrive at a suitable perspective. Transforming it in different ways can help you get started. You don't have to make your own toolbox by implementing the techniques yourself. You do not necessarily need a detailed understanding of how they are implemented. What you do need is to understand what the tools do and how they can be applied. In Part II we list, and briefly describe, all the transformations in the Weka data mining workbench. Data often calls for general mathematical transformations of a set of attributes. It might be useful to define new attributes by applying specified mathematical functions to existing ones. Two date attributes might be subtracted to give a third attribute representing age—an example of a semantic transformation driven by the meaning of the original attributes. Other transformations might be suggested by known properties of the learning algorithm. If a linear relationship involving two attributes, A and B, is suspected, and the algorithm is only capable of axis-parallel splits (as most decision tree and rule learners are), the ratio A/B might be defined as a new attribute. The transformations are not necessarily mathematical ones but may involve world knowledge such as days of the week, civic holidays, or chemical atomic numbers. They could be expressed as operations in a spreadsheet or as functions that are implemented by arbitrary computer programs. Or you can reduce several nominal attributes to one by concatenating their values, producing a single k1 ¥ k2-valued attribute from attributes with k1 and k2 values, respectively. Discretization converts a numeric attribute to nominal, and we saw earlier how to convert in the other direction too. As another kind of transformation, you might apply a clustering procedure to the dataset and then define a new attribute whose value for any given instance is the cluster that contains it using an arbitrary labeling for clusters. Alternatively, with probabilistic clustering, you could augment each instance with its membership probabilities for each cluster, including as many new attributes as there are clusters. Sometimes it is useful to add noise to data, perhaps to test the robustness of a learning algorithm. To take a nominal attribute and change a given percentage of its values. To obfuscate data by

renaming the relation, attribute names, and nominal and string attribute values—because it is often necessary to anonymize sensitive datasets. To randomize the order of instances or produce a random sample of the dataset by resampling it. To reduce a dataset by removing a given percentage of instances, or all instances that have certain values for nominal attributes, or numeric values above or below a certain threshold. Or to remove outliers by applying a classification method to the dataset and deleting misclassified instances. Different types of input call for their own transformations. If you can input sparse data files, you may need to be able to convert datasets to a nonsparse form, and vice versa. But first we look at two general techniques for transforming data with numeric attributes into a lower-dimensional form that may be more useful for data mining.

8 --- we introduced string attributes (lab 2, task 4) that contain pieces of text and remarked that the value of a string attribute is often an entire document. String attributes are basically nominal, with an unspecified number of values. If they are treated simply as nominal attributes, models can be built that depend on whether the values of two string attributes are equal or not. But that does not capture any internal structure of the string or bring out any interesting aspects of the text it represents. You could imagine decomposing the text in a string attribute into paragraphs, sentences, or phrases. Generally, however, the word is the most useful unit. The text in a string attribute is usually a sequence of words, and is often best represented in terms of the words it contains. For example, you might transform the string attribute into a set of numeric attributes, one for each word, that represent how often the word appears. The set of words—that is, the set of new attributes— is determined from the dataset and is typically quite large. If there are several string attributes whose properties should be treated separately, the new attribute names must be distinguished, perhaps by a user-determined prefix. Conversion into words—tokenization—is not such a simple operation as it sounds. Tokens may be formed from contiguous alphabetic sequences with nonalphabetic characters discarded. If numbers are present, numeric sequences may be retained too. Numbers may involve + or - signs, may contain decimal points, and may have exponential notation—in other words, they must be parsed according to a defined number syntax. An alphanumeric sequence may be regarded as a single token. Perhaps the space character is the token delimiter; perhaps white space (including the tab and new-line characters) is the delimiter, and perhaps punctuation is, too. Periods can be difficult: sometimes they should be considered part of the word (e.g., with initials, titles, abbreviations, and numbers), but sometimes they should not (e.g., if they are sentence delimiters). Hyphens and apostrophes are similarly problematic. All words may be converted to lowercase before being added to the dictionary. Words on a fixed, predetermined list of function words or stopwords—such as the, and, and but—could be ignored. Note that stopword lists are language dependent. In fact, so are capitalization conventions (German capitalizes all nouns), number syntax (Europeans use the comma for a decimal point), punctuation conventions (Spanish has an initial question mark), and, of course, character sets. Text is complicated! Low-frequency words such as hapax legomena3 are often discarded, too. Sometimes it is found beneficial to keep the most frequent k words after stopwords have been removed—or perhaps the top k words for each class. Along with all these tokenization options, there is also the question of what the value of each word attribute should be. The value may be the word count—the number of times the word appears in the string—or it may simply indicate the word's presence or absence.Word frequencies could be normalized to give each document's attribute vector the same Euclidean length. Alternatively, the frequencies fij for word i in

document j can be transformed in various standard ways. One standard logarithmic term frequency measure is log (1 + fij). A measure that is widely used in information retrieval is TF ¥ IDF, or "term frequency times inverse document frequency." Here, the term frequency is modulated by a factor that depends on how commonly the word is used in other documents. The TF ¥ IDF metric is typically defined as (read) The idea is that a document is basically characterized by the words that appear often in it, which accounts for the first factor, except that words used in every document or almost every document are useless as discriminators, which accounts for the second. TF ¥ IDF is used to refer not just to this particular formula but also to a general class of measures of the same type. For example, the frequency factor fij may be replaced by a logarithmic term such as log (1 + fij).

9--- A problem that plagues practical data mining is poor quality of the data. Errors in large databases are extremely common. Attribute values, and class values too, are frequently unreliable and corrupted. Although one way of addressing this problem is to painstakingly check through the data, data mining techniques themselves can sometimes help to solve the problem. Improving decision trees It is a surprising fact that decision trees induced from training data can often be simplified, without loss of accuracy, by discarding misclassified instances from the training set, relearning, and then repeating until there are no misclassified instances. Experiments on standard datasets have shown that this hardly affects the classification accuracy of C4.5, a standard decision tree induction scheme. In some cases it improves slightly; in others it deteriorates slightly. The difference is rarely statistically significant—and even when it is, the advantage can go either way.What the technique does affect is decision tree size. The resulting trees are invariably much smaller than the original ones, even though they perform about the same. What is the reason for this? When a decision tree induction method prunes away a subtree, it applies a statistical test that decides whether that subtree is "justified" by the data. The decision to prune accepts a small sacrifice in classification accuracy on the training set in the belief that this will improve test-set performance. Some training instances that were classified correctly by the unpruned tree will now be misclassified by the pruned one. In effect, the decision has been taken to ignore these training instances. But that decision has only been applied locally, in the pruned subtree. Its effect has not been allowed to percolate further up the tree, perhaps resulting in different choices being made of attributes to branch on. Removing the misclassified instances from the training set and relearning the decision tree is just taking the pruning decisions to their logical conclusion. If the pruning strategy is a good one, this should not harm performance. It may even improve it by allowing better attribute choices to be made. It would no doubt be even better to consult a human expert. Misclassified training instances could be presented for verification, and those that were found to be wrong could be deleted—or better still, corrected. Notice that we are assuming that the instances are not misclassified in any systematic way. If instances are systematically corrupted in both training and test sets—for example, one class value might be substituted for another—it is only to be expected that training on the erroneous training set would yield better performance on the (also erroneous) test set. Interestingly enough, it has been shown that when artificial noise is added to attributes (rather than to classes), test-set performance is improved if the same noise is added in the same way to the training set. In other words, when attribute noise is the problem it is not a good idea to train on a "clean" set if performance is to be assessed on a "dirty" one. A learning method can learn to compensate for attribute noise, in some measure, if given a chance. In essence, it can learn which attributes are unreliable and, if they are all

unreliable, how best to use them together to yield a more reliable result. To remove noise from attributes for the training set denies the opportunity to learn how best to combat that noise. But with class noise (rather than attribute noise), it is best to train on noise-free instances if possible.

10 --- A serious problem with any form of automatic detection of apparently incorrect data is that the baby may be thrown out with the bathwater. Short of consulting a human expert, there is really no way of telling whether a particular instance really is an error or whether it just does not fit the type of model that is being applied. In statistical regression, visualizations help. It will usually be visually apparent, even to the nonexpert, if the wrong kind of curve is being fitted—a straight line is being fitted to data that lies on a parabola, for example. But most problems cannot be so easily visualized: the notion of "model type" is more subtle than a regression line.And although it is known that good results are obtained on most standard datasets by discarding instances that do not fit a decision tree model, this is not necessarily of great comfort when dealing with a particular new dataset. The suspicion will remain that perhaps the new dataset is simply unsuited to decision tree modeling. One solution that has been tried is to use several different learning schemes— such as a decision tree, and a nearest-neighbor learner, and a linear discriminant function—to filter the data. A conservative approach is to ask that all three schemes fail to classify an instance correctly before it is deemed erroneous and removed from the data. In some cases, filtering the data in this way and using the filtered data as input to a final learning scheme gives better performance than simply using the three learning schemes and letting them vote on the outcome. Training all three schemes on the filtered data and letting them vote can yield even better results. However, there is a danger to voting techniques: some learning algorithms are better suited to certain types of data than others, and the most appropriate method may simply get out-voted! The lesson, as usual, is to get to know your data and look at it in many different ways. One possible danger with filtering approaches is that they might conceivably just be sacrificing instances of a particular class (or group of classes) to improve accuracy on the remaining classes. Although there are no general ways to guard against this, it has not been found to be a common problem in practice. Finally, it is worth noting once again that automatic filtering is a poor substitute for getting the data right in the first place. If this is too time consuming and expensive to be practical, human inspection could be limited to those instances that are identified by the filter as suspect.

the end for now---