Chapter 8

Classification: Advanced Methods

In this chapter, you will learn advanced techniques for data classification. We start with **feature selection and engineering** (Section 8.1). Then, we will introduce **Bayesian belief networks** (Section 8.2), which unlike naïve Bayesian classifiers, do not assume class conditional independence. A powerful approach to classification known as support vector machines is presented in Section 8.3. A **support vector machine** transforms training data into a higher dimensional space, where it finds a hyperplane that separates the data by class using essential training tuples called *support vectors*. Section 8.4 describes **rule-based classifiers and classification using frequent patterns**. For the former, our classifier is in the form of a set of IF-THEN rules, while the latter explores relationships between attribute–value pairs that occur frequently in data. This methodology builds on research on frequent pattern mining (Chapters 5 and 6). Classification with weak supervision is introduced in Section 8.5. Section 8.6 introduces various techniques for classification on rich data types, such as stream data, sequence data and graph data. Other related techniques to classification, such as multiclass classification, distance metric learning, interpretability of classification, reinforcement learning, and genetic algorithms are introduced in 8.7.

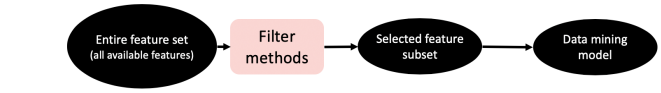
8.1 Feature Selection and Engineering

For the classification setting introduced in Chapter 7, in order to train a classifier (e.g., naïve Bayes Classifier, k-nearest-neighbor classifier), we assume that there exists a training set with n tuples, each of which is represented by p attributes or features. *"But, where do these p features come from at the first place?"* Let us consider two scenarios. In the first scenario (**Feature Selection**), you might have collected a large number of (say hundreds or thousands or even more) features. But most of them might be irrelevant with respect to the classification

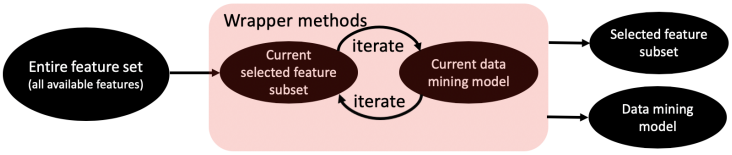
task or redundant with each other. For example, in order to predict whether or not an online student will drop out before finishing the program, the student ID is an irrelevant feature; in order to predict whether a customer will buy a computer, one of the two features, namely yearly income and monthly income, is redundant since one (e.g., yearly income) can be inferred from the other (e.g., monthly income). Including such irrelevant or redundant features during the classifier training process will not help improve the classification accuracy, yet they are likely to make the trained classifier sensitive to the noise, leading to degraded generalization performance. *"How can we select a subset of most relevant features from the initial p input features to train a classification model?* This is the main focus of this section. In the second scenario (**Feature Engineering**), you might wonder *"How can I construct p features so that all of them are critical for the classification task I have?"* or *"Given the initial p features, how can I transform them into another p' attributes so that these transformed features will be more effective for the given classification task?* These are the questions that *feature engineering* tries to answer. For example, in order to predict whether a regional disease outbreak will occur, one might have collected a large number of features from the health surveillance data, including the number of daily positive cases, number of daily tests, number of daily hospitalization, etc. It turns out a powerful indicator (or feature) to predict the disease outbreak is *weekly positive rate*. In this example, the weekly positive rate, which is the ratio of the number of positive cases and the number of tests of a week, can be constructed (or engineered) based on the initial features (e.g., daily positive cases, daily test cases). In practice, feature engineering plays a very important role in the performance of the classification model. Traditionally, feature engineering requires a lot of domain knowledge. Some data transformation techniques (e.g., DWT, DFT, and PCA), that were introduced in Chapter 3 can be viewed as feature engineering methods. The deep learning techniques that we will introduce in Chapter 11 provide an automatic way for feature engineering, capable of generating powerful features from the initial input features. The engineered features are often semantically more meaningful with a significant classification performance improvement.

In this section, we will introduce three types of feature selection methods, namely **filter methods**, **wrapper methods** and **embedded methods**. A filter method selects features based on some goodness measure which is independent of the specific classification model. A wrapper method combines the feature selection and classifier model construction steps together, and it iteratively uses the currently selected feature subset to construct a classification model, which is turn used to update the selected feature subset. An embedded method simultaneously constructs the classification model and selects the relevant features. In other words, it *embeds* the feature selection step during the classification model construction step. Figure 8.1 provides a pictorial comparison of these three methods.

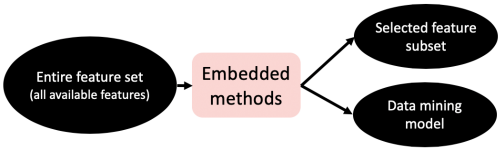
Feature selection can be used for both classification and regression. It can also be applied to the unsupervised data mining tasks, such as clustering. For both filter and wrapper methods, we will illustrate them with the classification



(a) Filter methods



(b) Wrapper methods



(c) Embedded methods

Figure 8.1: An overview of three feature selection methods.

task. We will mainly use the linear regression task (which was introduced in Section 7.5) to explain the embedded methods.

8.1.1 Filter Methods

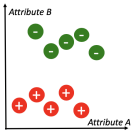
A **filter method** selects "good" features based on certain "goodness" measure of the input features. A filter method is independent of the specific classification model and is often used as a pre-processing step of other feature selection methods (e.g., wrapper or embedded methods). The idea is quite straight-forward. Suppose we have p initial features and we wish to select k out of p features (where $k < p$). If we have a goodness score for each feature, we can simply select k features with the highest goodness scores.

"So, how shall we measure the goodness of a feature?" Intuitively, we might say that a feature is good if it is highly *correlated* with the class label we want to predict. Suppose there are n training tuples. We wish to measure the correlation between a feature (i.e., attribute) x and the class label y. How can we measure the correlation between the given feature x and the class label y? If the given feature x is a categorical attribute (e.g., job title), a natural choice is χ^2 test, which was introduced in Section 2.2.3. To be specific, a higher χ^2 value indicates a stronger correlation between the given feature x and the class label y. We select k features with the highest χ^2 values.

"*But, what if the given feature x is a continuous attribute (e.g., yearly income)?*" We have two choices. First, we can discretize the continuous attribute x into a categorical attribute (e.g., high, medium vs. low income) and then use the χ^2 test to measure the correlation between the discrtized attribute and the class label to select k most correlated features. Second, we can resort to **Fisher**

tuple index	1	2	3	4	5	6	7	8	9	10
Attribute A	1	2	3	4	5	2	3	4	5	6
Attribute B	1.0	0.9	0.8	1.2	1.1	5.1	4.8	4.9	5.2	5.0
Class label	+	+	+	+	+	-	-	-	-	-

(a) Training tuples



(b) Scatter-plot

Figure 8.2: Feature section by Fisher score. (a) 10 training tuples, each of which is represented by two attributes (attribute A and attribute B) and a binary class label ('+' vs '-'). (b) scatter-plot of training tuples. Intuitively, attribute B better separates the positive training tuples from negative ones than attribute A. This is consistent with Fisher scores: $s(\text{attribute B}) = 200 > s(\text{attribute A}) = 0.125$.

score to directly measure the correlation between a continuous variable (the given feature x) and a categorical variable (the class label y).

Suppose we have a binary class label y (i.e., whether or not the customer will buy a computer). Intuitively, the feature x (e.g., income) is strongly correlated with the class label y if (1) the average income of all customers who buy a computer is significantly different from the average income of all customers who do not buy a computer, (2) all customers who buy a computer share similar income, and (3) all customers who do not buy a computer share similar income. Formally, Fisher score is defined as follows

$$s = \frac{\sum_{j=1}^{c} n_j(\mu_j - \mu)^2}{\sum_{j=1}^{c} n_j \sigma_j^2}, \tag{8.1}$$

where c is the total number of classes ($c = 2$ in our example), n_j is the number of training tuples in class j, μ_j and σ_j^2 are the mean value and variance of feature x among all tuples that belong to class j respectively, and μ is the mean value of feature x among all training tuples. Therefore, a feature x would have a high Fisher score if the following conditions hold. First, the *class-specific mean* values $u_j (j = 1, ..., c)$ are dramatically different from each other (e.g., a large numerator of the Fisher score in Eq. (8.1)). Intuitively, this implies that on average, the feature values from different classes are quite different from each other. Second, the *class-specific variance* σ_j^2 is small (e.g., a small denominator of the Fisher score in Eq. (8.1)). This indicates that within a class, different training tuples share similar feature values.

Example 8.1.1 We are given 10 training tuples in Figure 8.2(a), each of which is represented by two attributes (attribute A and attribute B) and a binary class label ('+' vs '-'). We want to use Fisher scores to decide which attribute is more correlated with the class label. There are 5 positive tuples and 5 negative tuples $n_1 = n_2 = 5$. For attribute A, the mean value among all training tuples $\mu = (1+2+3+4+5+2+3+4+5+6)/10 = 3.5$, the mean value among positive training tuples $\mu_1 = (1+2+3+4+5)/5 = 3$, and the mean value among negative training tuples $\mu_2 = (2+3+4+5+6)/5 = 4$, the variance of the positive tuples

$\sigma_1^2 = ((1-3)^2+(2-3)^2+(3-3)^2+(4-3)^2+(5-3)^2)/5 = 2$, and the variance of the negative tuples $\sigma_2^2 = ((2-4)^2+(3-4)^2+(4-4)^2+(5-4)^2+(6-4)^2)/5 = 2$. Therefore, the Fisher score for attribute A is $s(\text{attribute A}) = (5 \times (3-3.5)^2 + 5 \times (4-3.5)^2))/(5 \times 2 + 5 \times 2) = 0.125$. We compute the Fisher score for attribute B in a similar way and have that $s(\text{attribute B}) = 200$. According to Fisher scores, attribute B is more correlated with the class label than attribute A. This is consistent with the scatter-plot in Figure 8.2(b), where the positive tuples are well separated from negative tuples along the vertical axis (attribute B); whereas they are mixed together along the horizontal axis (attribute A).

In addition to correlation measures (e.g., χ^2 test for categorical feature, Fisher score for continuous feature), we might say that a feature x is good if it contains 'a lot of information' about the class label y which we want to predict. This suggests **information-theoretic goodness measures** for feature selection. For example, we can use *information gain* as the goodness measure for feature selection. The information gain, entropy and conditional entropy were introduced in Section 7.2. In a nutshell, let $H(y)$ be the entropy of the class label y and $H(y|x)$ be the condition entropy of the class label y given the feature x. The information gain of the feature x is defined as the difference between $H(y)$ and $H(y|x)$. The intuition is that a feature x with a larger information gain can better reduce the impurity (e.g., entropy) of the class label y. Thus, it contains 'more information' about predicting the class label y. In addition to information gain, another commonly used information theoretic goodness measure for feature selection is *mutual information* (*MI*). Intuitively, the mutual information between a feature x and the class label measures how much information the feature x provides to make the correct prediction of the class label y. Therefore, features with largest mutual information should be selected. The details about how to compute mutual information will be introduced in Chapter 10.

8.1.2 Wrapper Methods

With filter methods, the general process of training a classification model is as follows (Figure 8.1(a)). Given a set of p initial features, we first use a filter method to select k features (e.g., k out of p features with the highest Fisher scores). Then, using these k selected features, we build a classifier (e.g., a logistic regression classifier). Finally, we evaluate the performance of the trained classifier, such as cross-validation accuracy. Notice that during the feature selection process, a filter method is *independent* of the specific classification model that will be trained with the selected features. Another potential limitation with a filter method is that it does not consider the interaction between different features, and thus might select *redundant* features.

A **wrapper method** adopts a different strategy for feature selection by combining the feature selection step and classifier training step together. A wrapper method is often an iterative process (Figure 8.1(b)). At each iteration, it tries to build a classifier based on the *currently* selected feature subset, and

then based on the built classifier, it updates (e.g., add, remove, swap) the selected feature subset. In other words, it *wraps* the feature selection and classifier training together, hence the name of wrapper.

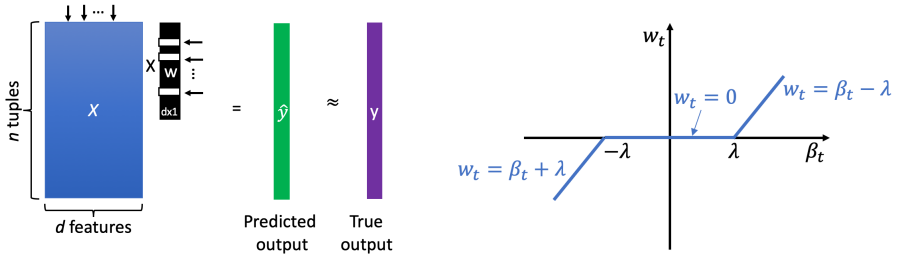
The key in a wrapper method is how to search for the best feature subset. A straight-forward way (i.e., exhaustive search) is to try all the possible subsets of the p given features. We use each subset of the feature to build a classification model and evaluate its performance, such as the classification accuracy using either the held-out set or cross-validation. The best feature subset is the one with the highest classification accuracy for the given classification model. This strategy is optimal since it finds the best feature subset with the highest classification accuracy. However, it is very expensive in terms of computation, since it needs to search and evaluate all $(2^p - 1)$ possible subsets of the p given features—an exponential number!

In practice, a wrapper method often relies on some heuristic search strategy to avoid the $(2^p - 1)$ exponential search space. Section 3.4.4 introduced different attribute subset selection strategies, which can be applied here. For example, in the *stepwise forward selection* method, it starts with an empty feature subset. At each iteration of the feature selection process, it selects an additional feature, which, when added into the current feature subset, will improve the classification model performance most (e.g., measured by the held-out method or cross-validation). The process will terminate when adding the extra features can no longer improve the classification model performance. In contrast, in the *stepwise backward elimination* method, it starts with all the p initial features, and then iteratively eliminates features from the current subset whose removal would increase the classification accuracy most. We can also combine these two strategies together. That is, at each iteration, we try to select one additional feature, and in the meanwhile might eliminate one existing feature that will improve the classification accuracy most. In addition to these three typical search strategies, some wrapper methods leverage more sophisticated techniques, such as simulated annealing and genetic algorithm. Simulated annealing is a probabilistic optimization technique, often designed for complex (e.g., non-convex) optimization problems. The latter will be introduced in Section 8.7.

By "wrapping" the feature selection and the classification model construction steps together, a wrapper method tends to have better performance than filtering methods. However, since it needs to iteratively search for the feature subset and (re-)train the classification model, the computational cost of a wrapper method is usually much more intense than filter methods. How can we simultaneously enjoy the advantages of both filter methods and wrapper methods? That is what embedded methods try to answer.

8.1.3 Embedded Methods

Embedded methods aim to combine the advantages of both filter methods and wrapper methods. On one hand, an embedded method performs feature selection and classification model construction simultaneously, so that the two can mutually benefit from each other. On the other hand, an embedded method



(a) loss function (b) soft thresholding

Figure 8.3: An illustration of LASSO. (a) illustration of the loss function of LASSO (Eq. (8.2)). The training set is represented by an $n \times d$ feature matrix X whose rows are tuples and columns are features, and an $n \times 1$ output vector y. By minimizing the sum of the squared difference between the actual and predicted output (i.e., the first term of Eq. (8.2)), the trained linear regression models tries to make the predicted output $\hat{y}$ to be as close as possible to the actual output $\hat{y}$. By minimizing the l_1 norm of the weight vector w (i.e., the second term of Eq. (8.2)), some elements of the weight vector w are zeros (indicated by the arrows), and the corresponding features (the columns of the feature matrix X) are 'un-selected'. (b) soft thresholding pushes the coefficients with small magnitudes (between $-\lambda$ and λ) to be zeros while shrinking the remaining coefficients by λ in magnitude.

tries to avoid the expensive, iterative search process in wrapper methods.

We actually have already seen an embedded method in Chapter 7! For decision tree induction that was introduced in Section 7.2, it is possible that only a fraction of all d initial attributes are present in the built decision tree model. For the example in Figure 7.2, only three attributes (i.e., age, student, credit_rating) are present in the decision tree model, albeit there might be tens of or hundred of initial attributes. This could happen if the decision tree induction algorithm terminates before it exhausts all d initial attributes, or some attributes of the initially built decision tree are removed during tree pruning process. In either case, all the attributes that appear on the non-leaf tree nodes can be viewed as the selected feature subset, and decision tree induction itself can be viewed as an embedded method for feature selection. In other words, the feature selection process (i.e., to decide which attribute(s) are used as the non-leaf tree nodes) is *embedded* in the decision tree induction process. We simultaneously accomplish both the feature selection step and the classification model construction (i.e., decision tree induction) step. This is the essence of an embedded method.

Other powerful embedded methods often rely on a technique called *sparse learning*. Let us first introduce its high-level idea, and then we will explain the details using linear regression as an example. A handful data mining models can be solved from the optimization perspective, such as linear regression model, logistic regression. In a nutshell, we build these data mining models by minimizing some objective (or loss) function that directly or indirectly measures

the performance of the corresponding data mining model. For example, in least square linear regression, we find the optimal weight vector w by minimizing the sum of the squared difference between the predicted output and actual output; in logistic regression, we find the optimal weight vector w by minimizing the negative log likelihood. Now, let us modify the objective function so that it also 'penalizes' the number of the features it uses. By minimizing the modified objective function, the trained data mining model might only use a subset of all the d initial features, and thus accomplish the task of feature selection. In this way, we will be able to embed the feature selection process (by penalizing the number of features used in the final model) in the model training process.

"So, how can we penalize the number of features used in a data mining model, and how can we solve the modified optimization problem accordingly?" Let us explain the details using least square linear regression (which was introduced in Section 7.5) as an example. Recall that a multi-linear regression model assumes $\hat{y}_i = w^T x_i = w_0 + w_1 x_{i,1} + ... + w_d x_{i,d}$, where $\hat{y}_i$ is the predicted output for the i^{th} tuple, $x_i = (1, x_{i,1}, ...x_{i,d})$ is the attribute (feature) vector of the i^{th} tuple, and $w = (w_0, w_1, ..., w_d)$ is the weight vector. We find the optimal weight vector w by minimizing the loss function $L(w) = \frac{1}{2}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 = \frac{1}{2}\sum_{i=1}^{n}(y_i - w^T x_i)^2$, which measures the sum of the squared difference between the predicted output ($\hat{y}_i$) and the actual output (y_i). *"How can we 'embed' the feature selection in the processing of training such a linear regression model?"* For the j^{th} feature, if the corresponding weight $w_j = 0$, it has no contribution the linear regression model. In other words, this feature is 'un-selected'. This naturally suggests that we can use the l_0 norm[1] of the weight vector w, which counts the number of non-zero elements in the weight vector w, to measure how many features are used (i.e., selected) in the trained linear regression model. Therefore, if we train a linear regression model by minimizing the following modified loss function $\tilde{L}(w) = \frac{1}{2}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \lambda\|w\|_0 = \frac{1}{2}\sum_{i=1}^{n}(y_i - w^T x_i)^2 + \lambda\|w\|_0$, the optimal weight vector w is likely to contain some zero elements. Those features whose corresponding weights in vector w are non-zero are selected. The parameter $\lambda > 0$ balances two terms in the modified loss function. Generally speaking, the larger the λ, the less number of the features are likely to be selected (i.e., more elements in the weight vector w are likely to be zeros).

However, finding the optimal weight vector w that minimizes the modified loss function $\tilde{L}(w)$ is very hard. This is because the l_0 norm of the weight vector w, which tells how many features are selected, is *non-convex*. To address this issue, we replace the l_0 norm by another norm that is convex. It turns out the l_1 norm $\|w\|_1 = \sum_{j=0}^{d}|w_j|$ is the best convex approximation of the l_0 norm, where $|w_j|$ is the absolute value of w_j. Thus, we have a new loss function as follows.

$$\hat{L}(w) = \frac{1}{2}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \lambda\|w\|_1 = \frac{1}{2}\sum_{i=1}^{n}(y_i - w^T x_i)^2 + \lambda\sum_{j=0}^{d}|w_j| \qquad (8.2)$$

[1] l_0 is a special case of the l_p norm when p approaches 0. l_p norm was introduced in Chapter 2.4.

The regression model that minimizes the new loss function $\hat{L}(w)$ in Eq. (8.2) is called **LASSO**, which stands for Least Absolute Shrinkage and Selection Operator. The optimal weight vector w of $\hat{L}$ is often *sparse*, meaning that some of its elements might be zeros. The non-zero elements of the optimal vector w tell that the corresponding features are selected by the linear regression model. Figure 8.3(a) presents an illustration of the loss function of LASSO (Eq. (8.2)).

"So, how can we find the optimal weight vector w that minimizes $\hat{L}$ in Eq. (8.2)?" The good news is that unlike function $\tilde{L}$ which is non-convex, the loss function $\hat{L}$ in Eq. (8.2) is a convex function. There existing many numerical optimization packages that can be used to solve it. Here, we introduce one of them, called *coordinate descent*.

Unlike the least square regression which has a closed-form solution, the closed-form solution for LASSO does not exists. Coordinate descent finds the optimal weight vector w in an iterative way, and it works as follows. First, we initialize the weight vector w. (We can simply set each element in w as zero.) Then, the algorithm iterates until it converges or some stopping criteria is met, for example, a maximum iteration number has been reached. At each iteration, the algorithm tries to update each element in the weight vector w one-by-one, while fixing all the remaining elements in w. Therefore, it boils down to the following question. *"How can we update a single element (say w_t $(0 \leq t \leq d)$) while fixing all other elements?"* We take the following three steps. First, we compute the residual for each training tuple $r_i = y_i - \sum_{j=0, j\neq t}^{d} w_j x_{i,j}$. The intuition of the residual r_i is that it measures the prediction error for the i^{th} tuple if we use all but the t^{th} features. Second, we train a least square regression model for all the input tuples, where each tuple is represented by a single input feature $x_{i,t}$ and its output is the residual r_i. The weight (i.e., coefficient) for the t^{th} feature from such a least square regression model is represented as β_t. (Recall that we can use the close-form solution of least square regression to find the coefficient β_t, which was introduced in Section 7.5). The intuition is that if we fix all but the t^{th} features, the coefficient β_t is the best coefficient that minimizes the overall least square prediction error. Third, we update the weight w_t as follows.

$$w_t = \begin{cases} \beta_t - \lambda & \text{if } \beta_t \geq \lambda \\ \beta_t + \lambda & \text{if } \beta_t \leq -\lambda \\ 0 & \text{otherwise} \end{cases} \tag{8.3}$$

The third step is called *soft thresholding*, and its intuition is as follows. If $|\beta_t|$ is greater than the regularization parameter λ, the soft thresholding step would reduce the magnitude of β_t by λ which is used as the updated coefficient w_t; otherwise the coefficient w_t is simply set as zeros. In other words, the soft thresholding pushes the coefficients with small magnitude as zero while shrinking the remaining coefficients. In this way, the final weight vector w is likely to be sparse with many zero elements, and thus achieves the purpose of feature selection. Figure 8.3(b) presents an illustration of soft thresholding.

An earlier method for solving LASSO is called **LAR** which stands for least

angle regression. Recall that for linear regression introduced in Section 7.5, we could add the squared l_2 norm of the weight vector into the loss function to prevent over-fitting (i.e., Ridge regression). We can add both l_1 norm and the squared l_2 norm into the loss function L. Such a regression model is called **Elastic net**, and the features selected by Elastic net tend to be less correlated with each other, compared with LASSO. We can use a very similar idea as LASSO to embed the feature selection process in the classification model. For example, we can introduce an l_1 norm regularization term in the objective function of logistic regression, so that the weight vector of the trained logistic regression classifier is likely to be sparse. In other words, it only uses a few selected features).

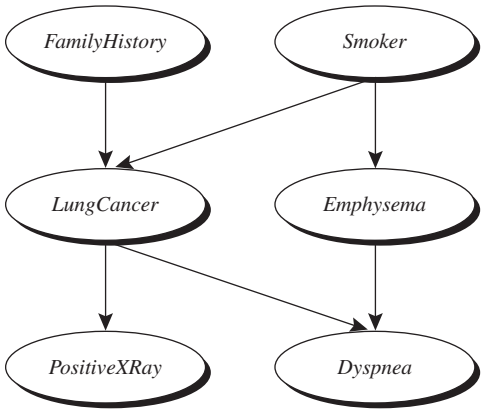
8.2 Bayesian Belief Networks

Chapter 7 introduced Bayes' theorem and naïve Bayesian classification. In this chapter, we describe *Bayesian belief networks*—probabilistic graphical models, which unlike naïve Bayesian classifiers allow the representation of dependencies among subsets of attributes. Bayesian belief networks can be used for classification. Section 8.2.1 introduces the basic concepts of Bayesian belief networks. In Section 8.2.2, you will learn how to train such models.

8.2.1 Concepts and Mechanisms

The naïve Bayesian classifier makes the assumption of class conditional independence, that is, given the class label of a tuple, the values of the attributes are assumed to be conditionally independent of one another. This simplifies computation. When the assumption holds true, the naïve Bayesian classifier is the most accurate in comparison with all other classifiers. In practice, however, dependencies can exist between variables (i.e., attributes) **Bayesian belief networks** specify joint probability distributions. They allow class conditional independence to be defined between subsets of variables. They provide a graphical model of causal relationships, on which learning can be performed. Trained Bayesian belief networks can be used for classification. Bayesian belief networks are also known as **belief networks**, **Bayesian networks**, and **probabilistic networks**. For brevity, we will refer to them as belief networks.

A belief network is defined by two components—a *directed acyclic graph* and a set of *conditional probability tables* (Figure 8.4). Each node in the directed acyclic graph represents a random variable. The variables may be discrete- or continuous-valued. They may correspond to actual attributes given in the data or to "hidden variables" believed to form a relationship (e.g., in the case of medical data, a hidden variable may indicate a syndrome, representing a number of symptoms that, together, characterize a specific disease). Each arc represents a probabilistic dependence. If an arc is drawn from a node Y to a node Z, then Y is a **parent** or **immediate predecessor** of Z, and Z is a **descendant** of Y. *Each variable is conditionally independent of its nondescendants in the graph, given its parents.*



	FH, S	FH, ~S	~FH, S	~FH, ~S
LC	0.8	0.5	0.7	0.1
~LC	0.2	0.5	0.3	0.9

(b)

Figure 8.4: Simple Bayesian belief network. (a) A proposed causal model, represented by a directed acyclic graph. (b) The conditional probability table for the values of the variable *LungCancer* (LC) showing each possible combination of the values of its parent nodes, *FamilyHistory* (FH) and *Smoker* (S). *Source:* Adapted from Russell, Binder, Koller, and Kanazawa [RBKK95].

Figure 8.4 is a simple belief network, adapted from Russell, Binder, Koller, and Kanazawa [RBKK95] for six Boolean variables. The arcs in Figure 8.4(a) allow a representation of causal knowledge. For example, having lung cancer is influenced by a person's family history of lung cancer, as well as whether or not the person is a smoker. Note that the variable *PositiveXRay* is independent of whether the patient has a family history of lung cancer or is a smoker, given that we know the patient has lung cancer. In other words, once we know the outcome of the variable *LungCancer*, then the variables *FamilyHistory* and *Smoker* do not provide any additional information regarding *PositiveXRay*. The arcs also show that the variable *LungCancer* is conditionally independent of *Emphysema*, given its parents, *FamilyHistory* and *Smoker*. On the other hand, we cannot say that *LungCancer* is conditionally independent of *Dyspnea*, given its parents. Why? This is because *Dyspnea* is a child of *LungCancer* in the belief network.

A belief network has one **conditional probability table (CPT)** for each variable. The CPT for a variable Y specifies the conditional distribution $P(Y|Parents(Y))$, where $Parents(Y)$ are the parents of Y. Figure 8.4(b) shows a CPT for the variable *LungCancer*. The conditional probability for each known value of *LungCancer* is given for each possible combination of the values of its parents. For instance, from the upper leftmost and bottom rightmost entries, respectively, we see that

$$P(LungCancer = yes \,|\, FamilyHistory = yes,\ Smoker = yes) = 0.8$$
$$P(LungCancer = no \,|\, FamilyHistory = no,\ Smoker = no) = 0.9.$$

Let $\boldsymbol{X} = (x_1, \ldots, x_n)$ be a data tuple described by the variables or attributes

$Y_1, \ldots, Y_n$, respectively. Recall that each variable is conditionally independent of its nondescendants, given its parents. This allows the belief network to provide a complete representation of the joint probability distribution by the following equation:

$$P(x_1, \ldots, x_n) = \prod_{i=1}^{n} P(x_i | Parents(Y_i)), \tag{8.4}$$

where $P(x_1, \ldots, x_n)$ is the probability of a particular combination of values of $\boldsymbol{X}$, and the values for $P(x_i|Parents(Y_i))$ correspond to the entries in the CPT for attribute Y_i.

A node within the belief network can be selected as an "output" node, representing a class label attribute. There may be more than one output node. Various algorithms for inference and learning can be applied to the network. Rather than returning a single class label, the classification process can return a probability distribution that gives the probability of each class. Belief networks can be used to answer probability of evidence queries (e.g., what is the probability that an individual will have *LungCancer*, given that they have both *PositiveXRay* and *Dyspnea*?) and most probable explanation queries (e.g., which group of the population is most likely to have both *PositiveXRay* and *Dyspnea*?).

Belief networks have been used to model a number of well-known problems. One example is genetic linkage analysis (e.g., the mapping of genes onto a chromosome). By casting the gene linkage problem in terms of inference on Bayesian networks, and using efficient algorithms, the scalability of such analysis has advanced considerably. Other applications that have benefited from the use of belief networks include computer vision (e.g., image restoration and stereo vision), document and text analysis, decision-support systems, financial fraud detection, and sensitivity analysis. The ease with which many applications can be reduced to Bayesian network inference is advantageous in that it curbs the need to invent specialized algorithms for each such application.

8.2.2 Training Bayesian Belief Networks

"*How does a Bayesian belief network learn?*" In the learning or training of a belief network, a number of scenarios are possible. The network **topology** (or "layout" of nodes and arcs) may be constructed by human experts or inferred from the data. The network variables may be *observable* or *hidden* in all or some of the training tuples. The hidden data case is also referred to as *missing values* or *incomplete data*.

Several algorithms exist for learning the network topology from the training data given observable variables. The problem is one of discrete optimization. For solutions, please see the bibliographic notes at the end of this chapter. Human experts usually have a good grasp of the direct conditional dependencies that hold in the domain under analysis, which helps in network design. Experts must specify conditional probabilities for the nodes that participate in direct

dependencies. These probabilities can then be used to compute the remaining probability values.

If the network topology is known and the variables are observable, then training the network is straightforward. It consists of computing the CPT entries, as is similarly done when computing the probabilities involved in naïve Bayesian classification.

When the network topology is given and some of the variables are hidden, there are various methods to choose from for training the belief network. We will describe an effective method based on *gradient descent*, which was also used to train a logistic regression classifier in Chapter 7. For those without an advanced math background, the description of a gradient descent method may look rather intimidating with its calculus-packed formulae. However, packaged software exists to solve these equations. Let us recap the general idea behind a gradient descent method, which is easy to follow.

Let D be a training set of data tuples, $\boldsymbol{X}_1, \boldsymbol{X}_2, \ldots, \boldsymbol{X}_{|D|}$. Training the belief network means that we must learn the values of the CPT entries. Let w_{ijk} be a CPT entry for the variable $Y_i = y_{ij}$ having the parents $U_i = u_{ik}$, where $w_{ijk} \equiv P(Y_i = y_{ij}|U_i = u_{ik})$. For example, if w_{ijk} is the upper leftmost CPT entry of Figure 8.4(b), then Y_i is *LungCancer*; y_{ij} is its value, "*yes*"; U_i lists the parent nodes of Y_i, namely, {*FamilyHistory, Smoker*}; and u_{ik} lists the values of the parent nodes, namely, {*"yes", "yes"*}. The w_{ijk} are viewed as weights, analogous to the weights in logistic regression. The set of weights is collectively referred to as $\boldsymbol{W}$. The weights are initialized to random probability values. A *gradient descent* strategy performs greedy hill-descending. At each iteration, the weights are updated and will eventually converge to a local optimum solution.

A **gradient descent** strategy is used to search for the optimal values of certain variables that best minimize an objective function, based on the assumption that each of the possible values is equally likely. Such a strategy is iterative. It searches for a solution along the negative of the gradient (i.e., steepest descent) of an objective function. In our setting, we want to find the set of weights, $\boldsymbol{W}$, that maximize an objective function.[2] To start with, the weights are initialized to random probability values. The gradient ascent method performs greedy hill-climbing in that, at each iteration or step along the way, the algorithm moves toward what appears to be the best solution at the moment, without backtracking. The weights are updated at each iteration. Eventually, they converge to a local optimum solution.

For our problem, we maximize the objective function $P_w(D) = \prod_{d=1}^{|D|} P_w(\boldsymbol{X}_d)$. This can be done by following the gradient of $\ln P_w(S)$, which makes the problem simpler. (Recall that we have used the same trick to train a logistic regression classifer in Chapter 7.) Given the network topology and initialized w_{ijk}, the algorithm proceeds as follows:

[2]In order to apply gradient descent strategy to maximize, instead of minimize, an objective function, we actually do gradient *ascent* where we update the current solution along the direction of the gradient (i.e., gradient ascent).

1. **Compute the gradients:** For each i, j, k, compute

$$\frac{\partial \ln P_w(D)}{\partial w_{ijk}} = \sum_{d=1}^{|D|} \frac{\partial \ln(P(Y_i = y_{ij}, U_i = u_{ik}|\boldsymbol{X}_d))}{\partial w_{ijk}}. \tag{8.5}$$

   The probability on the right side of Eq. (8.5) is to be calculated for each training tuple, $\boldsymbol{X}_d$, in D. For brevity, let's refer to this probability simply as p. When the variables represented by Y_i and U_i are hidden for some $\boldsymbol{X}_d$, the corresponding probability p can be computed from the observed variables of the tuple using standard algorithms for Bayesian network inference such as those available in the commercial software package HUGIN (*www.hugin.dk*).

2. **Take a small step in the direction of the gradient:** The weights are updated by

$$w_{ijk} \leftarrow w_{ijk} + \eta \frac{\partial \ln P_w(D)}{\partial w_{ijk}}, \tag{8.6}$$

   where η is the **learning rate** representing the step size and $\frac{\partial \ln P_w(D)}{\partial w_{ijk}}$ is computed from Eq. (8.5). The learning rate is set to a small constant and helps with convergence.

3. **Renormalize the weights:** Because the weights w_{ijk} are probability values, they must be between 0.0 and 1.0, and $\sum_j w_{ijk}$ must equal 1 for all i, k. These criteria are achieved by renormalizing the weights after they have been updated by Eq.(8.6).

Algorithms that follow this learning form are called *adaptive probabilistic networks*. Other methods for training belief networks are referenced in the bibliographic notes at the end of this chapter. Belief networks could be computationally intensive. Because belief networks provide explicit representations of causal structure, a human expert can provide prior knowledge to the training process in the form of network topology or conditional probability values. This can significantly improve the learning speed.

8.3 Support Vector Machines

In this section, we study **support vector machines (SVMs)**, a method for the classification of both linear and nonlinear data. In a nutshell, an **SVM** is an algorithm that works as follows. It uses a nonlinear mapping to transform the original training data into a higher dimensional space. Within this new space, it searches for the linear optimal separating hyperplane (i.e., a "decision boundary" separating the tuples of one class from another). With an appropriate nonlinear mapping to a sufficiently high dimensional space, data from two classes can always be separated by a hyperplane. The SVM finds this hyperplane using *support vectors* ("essential" training tuples) and *margins* (defined by the support vectors). We will delve more into these new concepts later.

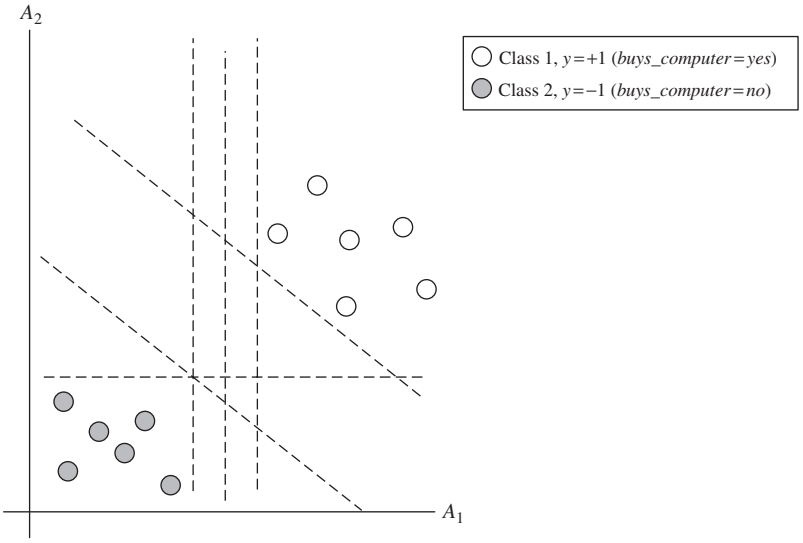


Figure 8.5: The 2-D training data that are linearly separable. There are an infinite number of possible separating hyperplanes or "decision boundaries", some of which are shown here as dashed lines. Which one is best?

"*I've heard that SVMs have attracted a great deal of attention lately. Why?*" The first paper on support vector machines was presented in 1992 by Vladimir Vapnik and colleagues Bernhard Boser and Isabelle Guyon, although the groundwork for SVMs has been around since the 1960s (including early work by Vapnik and Alexei Chervonenkis on statistical learning theory). Although the training of even the fastest SVMs could be slow, they are highly accurate, owing to their ability to model complex nonlinear decision boundaries. They are much less prone to overfitting than other methods. The support vectors also provide a compact description of the learned model. SVMs can be used for numeric prediction as well as classification. They have been applied to a number of areas, including handwritten digit recognition, object recognition, and speaker identification, as well as benchmark time-series prediction tasks.

8.3.1 Linear Support Vector Machines

To explain the mystery of SVMs, let's first look at the simplest case—a two-class problem where the classes are linearly separable. Let the data set D be given as $(\boldsymbol{X}_1, y_1), (\boldsymbol{X}_2, y_2), \ldots, (\boldsymbol{X}_{|D|}, y_{|D|})$, where $\boldsymbol{X}_i$ is the set of training tuples with associated class labels, y_i. Each y_i can take one of two values, either $+1$ or -1 (i.e., $y_i \in \{+1, -1\}$), corresponding to the classes *buys_computer* $=$ *yes* and *buys_computer* $=$ *no*, respectively. To aid in visualization, let's consider an example based on two input attributes, A_1 and A_2, as shown in Figure 8.5. From the graph, we see that the 2-D data are **linearly separable** (or "linear"

for short), because a straight line can be drawn to separate all the tuples of class +1 from all the tuples of class −1.

There are an infinite number of separating lines that could be drawn. We want to find the "best" one, that is, one that (we hope) will have the minimum classification error on previously unseen tuples. How can we find this best line? Note that if our data were 3-D (i.e., with three attributes), we would want to find the best separating *plane*. Generalizing to n dimensions, we want to find the best *hyperplane*. We will use "hyperplane" to refer to the decision boundary that we are seeking, regardless of the number of input attributes. So, in other words, how can we find the best hyperplane?

An SVM approaches this problem by searching for the **maximum margin hyperplane**. Consider Figure 8.6, which shows two possible separating hyperplanes and their associated margins. Before we get into the definition of margins, let's take an intuitive look at this figure. Both hyperplanes can correctly classify all the given data tuples. Intuitively, however, we expect the hyperplane with the larger margin to be more accurate at classifying future data tuples than the hyperplane with the smaller margin. This is why (during the learning or training phase) the SVM searches for the hyperplane with the largest margin, that is, the *maximum marginal hyperplane* (MMH). The associated margin gives the largest separation between classes.

Getting to an informal definition of **margin**, we can say that the shortest distance from a hyperplane to one side of its margin is equal to the shortest distance from the hyperplane to the other side of its margin, where the "sides" of the margin are parallel to the hyperplane. When dealing with the MMH, this distance is, in fact, the shortest distance from the MMH to the closest training tuple of either class.

A separating hyperplane is essentially a linear classifier. Similar to other linear classifiers (such as perceptron, logistic regression) that were introduced in Chapter 7.5, it can be written as

$$\boldsymbol{W} \cdot \boldsymbol{X} + b = 0, \tag{8.7}$$

where $\boldsymbol{W}$ is a weight vector, namely, $\boldsymbol{W} = \{w_1, w_2, \ldots, w_n\}$; n is the number of attributes; and b is a scalar, often referred to as a bias. To aid in visualization, let's consider two input attributes, A_1 and A_2, as in Figure 8.6(b). Training tuples are 2-D (e.g., $\boldsymbol{X} = (x_1, x_2)$), where x_1 and x_2 are the values of attributes A_1 and A_2, respectively, for $\boldsymbol{X}$. Eq.(8.7) can be written as

$$b + w_1 x_1 + w_2 x_2 = 0. \tag{8.8}$$

Thus, any point that lies above the separating hyperplane satisfies

$$b + w_1 x_1 + w_2 x_2 > 0. \tag{8.9}$$

Similarly, any point that lies below the separating hyperplane satisfies

$$b + w_1 x_1 + w_2 x_2 < 0. \tag{8.10}$$

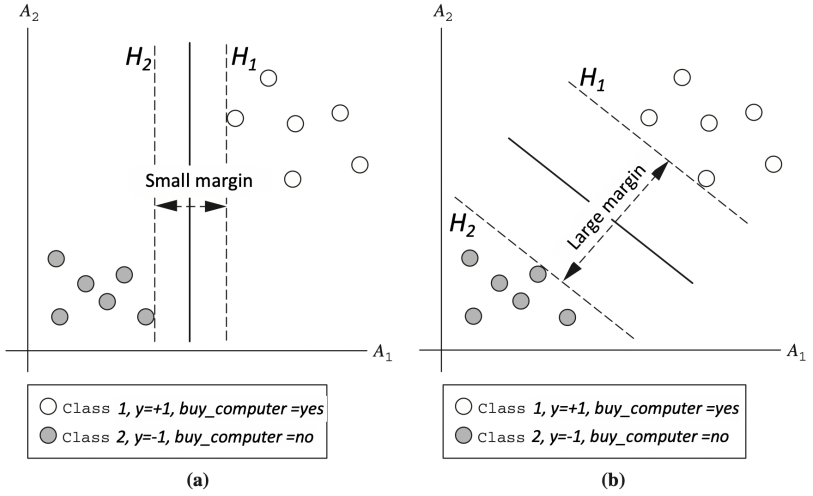


Figure 8.6: Here we see just two possible separating hyperplanes and their associated margins. Which one is better? The one with the larger margin (b) should have greater generalization accuracy.

The weights can be adjusted so that the hyperplanes defining the "sides" of the margin can be written as

$$H_1 : b + w_1x_1 + w_2x_2 \geq 1 \quad \text{for } y_i = +1, \tag{8.11}$$
$$H_2 : b + w_1x_1 + w_2x_2 \leq -1 \quad \text{for } y_i = -1. \tag{8.12}$$

That is, any tuple that falls on or above H_1 belongs to class $+1$, and any tuple that falls on or below H_2 belongs to class -1. Combining the two inequalities of Eqs. (8.11) and (8.12), we get

$$y_i(b + w_1x_1 + w_2x_2) \geq 1, \ \forall i. \tag{8.13}$$

Any training tuples that fall on hyperplanes H_1 or H_2 (i.e., $y_i(b + w_1x_1 + w_2x_2) = 1$) are called **support vectors**. That is, they are equally close to the (separating) MMH. In Figure 8.7, the support vectors are shown encircled with a thicker border. Essentially, the support vectors are the most difficult tuples to classify and give the most information regarding classification.

From this, we can obtain a formula for the size of the maximal margin. The distance from the separating hyperplane to any point on H_1 is $\frac{1}{||\boldsymbol{W}||}$, where $||\boldsymbol{W}||$ is the Euclidean norm of $\boldsymbol{W}$, that is, $\sqrt{\boldsymbol{W} \cdot \boldsymbol{W}}$.[3] By definition, this is equal to the distance from any point on H_2 to the separating hyperplane. Therefore, the maximal margin is $\frac{2}{||\boldsymbol{W}||}$. This suggests that we should minimize $\|\mathbf{W}\|^2$ in order to make the margin as large as possible. Notice that if the tuples are

[3]If $\boldsymbol{W} = \{w_1, w_2, \ldots, w_n\}$, then $\sqrt{\boldsymbol{W} \cdot \boldsymbol{W}} = \sqrt{w_1^2 + w_2^2 + \cdots + w_n^2}$.

in n dimensional space, Eq (8.13) becomes $y_i(\mathbf{W}'\mathbf{X}_i + b)) \geq 1$. Therefore, we have the following mathematical formulation of SVM . The intuition of such a formulation is that we want find a linear classifier (i.e., hyperplane), such that (1) its margin is as large as possible (i.e., $\min \|\mathbf{W}\|^2$), and (2) each training tuple is correctly classified (i.e., $y_i(\mathbf{W}'\mathbf{X}_i + b)) \geq 1, \quad \forall i$). The corresponding classifier is often called *hard-margin linear SVM*.

$$\begin{aligned} &\min \|\mathbf{W}\|^2, \\ &\text{s.t.} \quad y_i(\mathbf{W}'\mathbf{X}_i + b) \geq 1, \quad \forall i. \end{aligned} \tag{8.14}$$

"*So, how does an SVM find the MMH and the support vectors?*" Using some "fancy math tricks", we can rewrite Eq. (8.14) so that it becomes what is known as a (convex) quadratic programming problem. Such fancy math tricks are beyond the scope of this book. Advanced readers may be interested to note that the tricks involve rewriting Eq. (8.14) as its dual form using Lagrangian formulation and then solving for the solution using Karush-Kuhn-Tucker (KKT) conditions. Details can be found in the bibliographic notes at the end of this chapter (Section 8.10).

If the data are relatively small (say, with a few thousand training tuples), any optimization software package for solving convex quadratic programming problems can then be used to find the support vectors and MMH. For larger data, special and more efficient algorithms for training SVMs can be used instead, the details of which exceed the scope of this book. Once we've found the support vectors and MMH (note that the support vectors define the MMH!), we

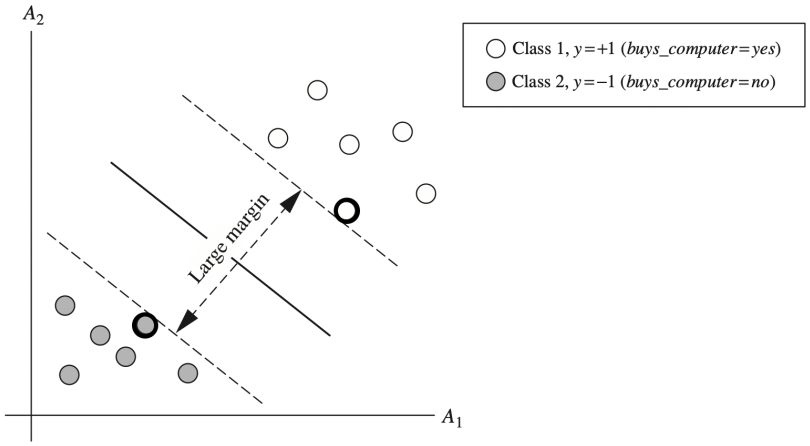


Figure 8.7: Support vectors. The SVM finds the maximum margin separating hyperplane, that is, the one with maximum distance between the nearest training tuples. The support vectors are shown with a thicker border.

have a trained support vector machine. The MMH is a linear class boundary, and so the corresponding SVM can be used to classify linearly separable data.

"*Once I've got a trained support vector machine, how do I use it to classify test (i.e., new) tuples?*" Based on the Lagrangian formulation mentioned before, the MMH can be rewritten as the decision boundary

$$d(\boldsymbol{X}) = \sum_{i=1}^{l} y_i \alpha_i \boldsymbol{X}' \boldsymbol{X}_i + b, \tag{8.15}$$

where y_i is the class label of support vector $\boldsymbol{X}_i$; $\boldsymbol{X}$ is a test tuple and $'$ denotes the transpose of a vector; α_i and b are numeric parameters that were determined automatically by the optimization or SVM algorithm noted before; and l is the number of support vectors, which is often much smaller than the total number of training tuples. Interested readers may note that the α_i are Lagrangian multipliers. For linearly separable data, the support vectors are a subset of the actual training tuples. slight twist regarding this when dealing with nonlinearly separable data, as we shall see in the following).

Given a test tuple, $\boldsymbol{X}$, we plug it into Eq. (8.15), and then check to see the sign of the result. This tells us on which side of the hyperplane the test tuple falls. If the sign is positive, then $\boldsymbol{X}$ falls above the MMH, and so the SVM predicts that $\boldsymbol{X}$ belongs to class $+1$ (representing *buys_computer* $=$ *yes*, in our case). If the sign is negative, then $\boldsymbol{X}$ falls below the MMH and the class prediction is -1 (representing *buys_computer* $=$ *no*).

Notice that the Lagrangian formulation of our problem Eq. (8.15) contains a dot product between support vector $\boldsymbol{X}_i$ and test tuple $\boldsymbol{X}$. This will prove very useful for finding the MMH and support vectors of a non-linear SVM when the given data are linearly inseparable, as described further in the next section. But before that, let's briefly introduce how we can modify the formulation of hard-margin linear SVM (Eq. (8.14)) for the nonlinear case. That is, we still wish to find a linear classifier (i.e. a hyperplane) when the training tuples are linearly inseparable. Here, the trick is that we allows some training tuples to be misclassified. To be specific, we can introduce a non-negative slack variable $\xi_i \geq 0$ for each training tuple, $\mathbf{X}_i$. If $\xi_i = 0$, it means that the corresponding tuple $\mathbf{X}_i$ is correctly classified by the hyperplane (i.e., $y_i(\mathbf{W}'\mathbf{X}_i + b) \geq 1$). In other words, a training example with $\xi_i = 0$ is just like the one in the hard-margin linear SVM. However, If $\xi_i > 0$, it means that the tuple $\mathbf{X}_i$ is incorrectly classified by the hyperplane and its magnitude $|\xi|$ indicates how far the training tuple is away from its corresponding side (i.e., H_1 for a positive training example, and H_2 for a negative training example). See Figure 8.8(a) for an illustration.

Then, we have the following alternative mathematical formulation of SVM. The corresponding classifier is often called *soft-margin linear SVM*. Different from hard-margin linear SVM, our new objective function has two terms, including (1) $\|\mathbf{W}\|^2$ which measures the size of margin (i.e,., the smaller $\|\mathbf{W}\|^2$, the larger margin), and (2) the sum of all slack variables $\sum_{i=1}^{N} \xi_i$ which measures the (approximate) number of incorrectly classified training tuples (i.e., the training error). In Eq. (8.16), N is the total number of training tuples and $C > 0$

is a user-tuned parameter that balances the size of margin and the training error. Note that we can use same optimization technique (i.e., convex quadratic programming) to solve Eq. (8.16) as for hard-margin linear SVM. Likewise, the resulting soft-margin linear classifier uses the same equation (Eq. (8.15)) to classify a test tuple.

$$\begin{aligned} &\min \|\mathbf{W}\|^2 + C\sum_{i=1}^{N}\xi_i, \\ &\text{s.t.} \quad y_i(\mathbf{W}'\mathbf{X}_i + b) \geq 1 - \xi_i, \quad \forall i. \end{aligned} \tag{8.16}$$

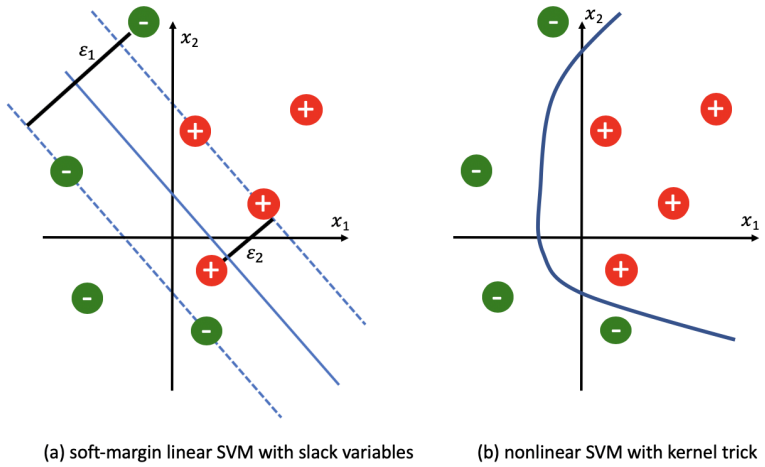
We end this section with two important things to note. The complexity of the learned classifier is characterized by the number of support vectors rather than the dimensionality of the data. Hence, SVMs tend to be less prone to overfitting than some other methods. The support vectors are the essential or critical training tuples—they lie closest to the decision boundary (MMH). If all other training tuples were removed and training were repeated, the same separating hyperplane would be found. Furthermore, the number of support vectors found can be used to compute an (upper) bound on the expected error rate of the SVM classifier, which is independent of the data dimensionality. An SVM with a small number of support vectors can have good generalization, even when the dimensionality of the data is high.

8.3.2 Nonlinear Support Vector Machines

In Section 8.3.1, we learned about hard-margin linear SVMs for classifying linearly separable data. We also learned about soft-margin linear SVMs when the training data are linearly inseparable, by allowing a small fraction of training tuples to be mis-classified. But, what if we want a 'better' classifier to avoid such mis-classifications? For linearly inseparable cases (e.g., Figure 8.8), no straight line can be found that would perfectly separate the classes.

The good news is that the approaches described for linear SVMs with both hard-margin and soft margin can be extended to create *nonlinear SVMs* for the classification of *linearly inseparable data* (also called *nonlinearly separable data*, or *nonlinear data* for short). Such SVMs are capable of finding nonlinear decision boundaries (i.e., nonlinear hypersurfaces) in input space.

"*So*," you may ask, "*how can we extend the linear approach?*" We obtain a nonlinear SVM by extending the approach for linear SVMs as follows. There are two main steps. In the first step, we transform the original input data into a higher dimensional space using a nonlinear mapping. Several common nonlinear mappings can be used in this step, as we will further describe next. Once the data have been transformed into the new higher dimensional space, the second step searches for a linear separating hyperplane in the new space. We again end up with an optimization problem that can be solved using the linear SVM formulation (i.e., convex quadratic programming). The maximal margin hyperplane found in the new space corresponds to a nonlinear separating



(a) soft-margin linear SVM with slack variables (b) nonlinear SVM with kernel trick

Figure 8.8: A simple 2-D case showing linearly inseparable data, where each tuple is represented by two attributes (x_1 and x_2). Unlike the linearly separable data of Figure 8.5, here it is not possible to draw a straight line to perfectly separate the two classes. We could use a soft-margin linear SVM, with the help of slack variables (ϵ_1 and ϵ_2), to produce a linear decision boundary at the expense of two training tuples being mis-classified (a). Alternatively, we could seek for a nonlinear decision boundary (b).

hypersurface in the original space.

Example 8.3.1 **Nonlinear transformation of original input data into a higher dimensional space.** Consider the following example. A 3-D input vector $\boldsymbol{X} = (x_1, x_2, x_3)$ is mapped into a 6-D space, $\boldsymbol{Z}$, using the mappings $\phi_1(\boldsymbol{X}) = x_1$, $\phi_2(\boldsymbol{X}) = x_2$, $\phi_3(\boldsymbol{X}) = x_3$, $\phi_4(\boldsymbol{X}) = (x_1)^2$, $\phi_5(\boldsymbol{X}) = x_1x_2$, and $\phi_6(\boldsymbol{X}) = x_1x_3$. A decision hyperplane in the new space is $d(\boldsymbol{Z}) = \boldsymbol{W}'\boldsymbol{Z}+b$, where $\boldsymbol{W}$ and $\boldsymbol{Z}$ are vectors. This is linear. We solve for $\boldsymbol{W}$ and b and then substitute back so that the linear decision hyperplane in the new ($\boldsymbol{Z}$) space corresponds to a nonlinear second-order polynomial in the original 3-D input space:

$$\begin{aligned} d(\boldsymbol{Z}) &= w_1x_1 + w_2x_2 + w_3x_3 + w_4(x_1)^2 + w_5x_1x_2 + w_6x_1x_3 + b \\ &= w_1z_1 + w_2z_2 + w_3z_3 + w_4z_4 + w_5z_5 + w_6z_6 + b. \end{aligned}$$

But there are some problems. First, how do we choose the nonlinear mapping to a higher dimensional space? Second, the computation involved will be costly. Refer to Eq. (8.15) for the classification of a test tuple, $\boldsymbol{X}$. Given the test tuple, we have to compute its dot product with every one of the support vectors.[4] In

[4]The dot product of two vectors, $\boldsymbol{X} = (x_1, x_2, \ldots, x_n)$ and $\boldsymbol{X}_i = (x_{i1}, x_{i2}, \ldots, x_{in})$ is $x_1x_{i1} + x_2x_{i2} + \cdots + x_nx_{in}$. Note that this involves one multiplication and one addition for each of the n dimensions.

training, we have to compute a similar dot product for each pair of training tuples in order to find the MMH. This is especially expensive. Hence, the dot product computation required is very heavy and costly. We need another trick!

Luckily, we can use another math trick. It so happens that in solving the quadratic optimization problem of the linear SVM (i.e., when searching for a linear SVM in the new higher dimensional space), the training tuples appear only in the form of dot products, $\phi(\boldsymbol{X}_i) \cdot \phi(\boldsymbol{X}_j)$, where $\phi(\boldsymbol{X})$ is simply the nonlinear mapping function applied to transform the training tuples. Instead of computing the dot product on the transformed data tuples, it turns out that it is mathematically equivalent to instead applying a *kernel function*, $K(\boldsymbol{X}_i, \boldsymbol{X}_j)$, to the original input data. That is,

$$K(\boldsymbol{X}_i, \boldsymbol{X}_j) = \phi(\boldsymbol{X}_i) \cdot \phi(\boldsymbol{X}_j). \tag{8.17}$$

In other words, everywhere that $\phi(\boldsymbol{X}_i) \cdot \phi(\boldsymbol{X}_j)$ appears in the training algorithm, we can replace it with $K(\boldsymbol{X}_i, \boldsymbol{X}_j)$. In this way, all calculations are made in the original input space, which is of potentially much lower dimensionality! We can safely avoid the mapping—it turns out that we don't even have to know what the mapping is! We will talk more later about what kinds of functions can be used as kernel functions for this problem. After applying this trick, we can then proceed to find a maximal margin separating hyperplane. The procedure is similar to that described in Section 8.3.1.

Example 8.3.2 Figure 8.9(a) shows a training set with 4 positive tuples and 4 negative tuples. In the original feature space, each tuple is represented by two features (x_1 and x_2), where the training set is linearly inseparable (Figure 8.9(b)). If we transform the original feature space into a 3-D space: $\Phi_1 = x_1^2$, $\Phi_2 = x_2^2$ and $\Phi_3 = \sqrt{2}x_1x_2$. In the transformed feature space (Figure 8.9(c)), the positive tuples are linearly separable from the negative tuples. In other words, we can use a hyperplane $\Phi_1 + \Phi_2 = 2.5$ to perfectly separate all positive tuples from all negative tuples. The hyperplane in the transformed feature space is equivalent to a nonlinear decision boundary in the original 2-D space $x_1^2 + x_2^2 = 2.5$. Note that the dot product of two tuples (X_i and X_j) in the transformed feature space can be computed directly from the original feature space: $\Phi(X_i) \cdot \Phi(X_i) = (X_i \cdot X_j)^2$.

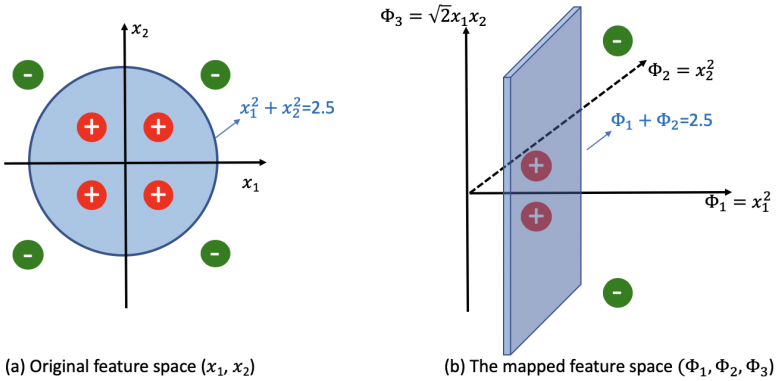
"*What are some of the kernel functions that could be used?*" Properties of the kinds of kernel functions that could be used to replace the dot product have been studied. Three admissible kernel functions are

$$\begin{aligned}
\textbf{Polynomial kernel of degree } \boldsymbol{h}\textbf{:} \quad & K(\boldsymbol{X}_i, \boldsymbol{X}_j) = (\boldsymbol{X}_i \cdot \boldsymbol{X}_j + 1)^h \\
\textbf{Gaussian radial basis function kernel:} \quad & K(\boldsymbol{X}_i, \boldsymbol{X}_j) = e^{-\|\boldsymbol{X}_i - \boldsymbol{X}_j\|^2/2\sigma^2} \\
\textbf{Sigmoid kernel:} \quad & K(\boldsymbol{X}_i, \boldsymbol{X}_j) = \tanh(\kappa \boldsymbol{X}_i \cdot \boldsymbol{X}_j - \delta)
\end{aligned}$$

Each of these results in a different nonlinear classifier in (the original) input space. There are no golden rules for determining which admissible kernel will result in the most accurate SVM. In practice, the kernel chosen does not generally make a large difference in resulting accuracy.

	tuple index	1	2	3	4	5	6	7	8
Original features	x_1	1	1	-1	-1	2	-2	2	-2
	x_2	1	-1	1	-1	2	2	-2	-2
	Class label	+	+	+	+	-	-	-	-
Mapped features	x_1^2	1	1	1	1	4	4	4	4
	x_2^2	1	1	1	1	4	4	4	4
	$\sqrt{2}x_1x_2$	$\sqrt{2}$	$-\sqrt{2}$	$-\sqrt{2}$	$\sqrt{2}$	$4\sqrt{2}$	$-4\sqrt{2}$	$-4\sqrt{2}$	$4\sqrt{2}$

(a) Training tuples in the original feature space (x_1, x_2) and in the mapped feature space (shaded)



(a) Original feature space (x_1, x_2) (b) The mapped feature space (Φ_1, Φ_2, Φ_3)

Figure 8.9: An example of kernel trick. (a) Training tuples in the original 2-D feature space and the transformed 3-D space (shaded). Training tuples in the original feature space are linearly inseparable (b), but become linearly separable in the transformed feature space (c). A linear decision boundary (i.e., a hyperplane) in the transformed feature space is equivalent to a nonlinear decision boundary in the original feature space. The dot product of two tuples in the transformed feature space can be computed directly from the original feature space. **yu: the subfigures are numbered as (a), (a), (b). In the third subfigure, the hyperplane looks perpendicular to axis-Φ_1?**

So far, we have described linear and nonlinear SVMs for binary (i.e., two-class) classification. SVM classifiers can be combined for the multiclass case. See Section 8.7.1 for some strategies, such as training one classifier per class and the use of error-correcting codes.

A major research goal regarding SVMs is to improve the speed in training and testing so that SVMs may become a more feasible option for very large data sets (e.g., millions of support vectors). A very efficient strategy is to train SVMs in its prime form directly (e.g., Eqs. (8.14) and (8.16)) based on stochastic sub-gradient descent. Recall that in Chapter 7.5, we have used a similar technique called stochastic gradient descent to address the scalability issue of logistic regression classifier. Other issues include (1) determining the best kernel for a given data set and finding more efficient methods for the multiclass case, (2) making the SVMs more robust to the noise in the training

data by using alternative norms of the weight vector $\mathbf{W}$ (e.g., l_1 norm SVM, $l_{2,1}$ norm SVM, capped l_p norm SVM). A key idea behind nonlinear SVM is the kernel trick, where we find a nonlinear classifier without explicitly constructing the non-linear mapping. The kernel trick has been broadly applied to other data mining tasks, including regression, clustering, and so on.

8.4 Rule-Based and Pattern-Based Classification

In this section, we look at rule-based and pattern-based classifiers. For the former, the learned model is represented as a set of IF-THEN rules. We first examine how such rules are used for classification (Section 8.4.1). We then study ways in which they can be generated, either from a decision tree (Section 8.4.2) or directly from the training data using a *sequential covering algorithm* (Section 8.4.3). Based on that, we introduce pattern-based classifiers, where frequent patterns are used for classification. Section 8.4.4 explores **associative classification**, where association rules are generated from frequent patterns and used for classification. The general idea is that we can search for strong associations between frequent patterns (conjunctions of attribute–value pairs) and class labels. Associative classification is a form of rule-based classifier, in that we often organize the mined association rule to form a rule-based classifier. Section 8.4.5 explores **discriminative frequent pattern–based classification**, where frequent patterns serve as combined features, which are considered in addition to single features when building a classification model. Because frequent patterns explore highly confident associations among multiple attributes, frequent pattern–based classification may overcome some constraints introduced by decision tree induction, which often only considers one attribute at a time. Studies have shown many frequent pattern–based classification methods to have greater accuracy and scalability than some traditional classification methods such as C4.5.

8.4.1 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 expression 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

$$R1\text{: } (age = youth) \wedge (student = yes) \Rightarrow (buys\_computer = yes).$$

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, $\boldsymbol{X}$, from a class-labeled data set, D, let n_{covers} be the number of tuples covered by R; $n_{correct}$ 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

$$coverage(R) = \frac{n_{covers}}{|D|} \tag{8.18}$$

$$accuracy(R) = \frac{n_{correct}}{n_{covers}}. \tag{8.19}$$

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.

Example 8.4.1 **Rule accuracy and coverage.** Let's go back to our data in Section 7.2 Table 7.1. These are class-labeled tuples from the *AllElectronics* customer database. Our task is to predict whether a customer will buy a computer. Consider rule $R1$, which covers 2 of the 14 tuples. It can correctly classify both tuples. Therefore, $coverage(R1) = 2/14 = 14.28\%$ and $accuracy(R1) = 2/2 = 100\%$.

Let's see how we can use rule-based classification to predict the class label of a given tuple, $\boldsymbol{X}$. If a rule is satisfied by $\boldsymbol{X}$, the rule is said to be **triggered**. For example, suppose we have

$$\boldsymbol{X}\text{= } (age = youth,\ income = medium,\ student = yes,\ credit\_rating = fair).$$

We would like to classify $\boldsymbol{X}$ according to *buys_computer*. $\boldsymbol{X}$ satisfies $R1$, which triggers the rule.

If $R1$ is the only rule satisfied, then the rule **fires** by returning the class prediction for $\boldsymbol{X}$. Note that triggering does not always mean firing because there may be more than one rule that is satisfied! If more than one rule is triggered, we have a potential problem. What if they each specify a different class? Or what if no rule is satisfied by $\boldsymbol{X}$?

We tackle the first question. If more than one rule is triggered, we need a **conflict resolution strategy** to figure out which rule gets to fire and assign

its class prediction to $\boldsymbol{X}$. There are many possible strategies. We look at two, namely *size ordering* and *rule ordering*.

The **size ordering** scheme assigns the highest priority to the triggering rule that has the "toughest" requirements, where toughness is measured by the rule antecedent *size*. That is, the triggering rule with the most attribute tests is fired.

The **rule ordering** scheme prioritizes the rules beforehand. The ordering may be *class-based* or *rule-based*. With **class-based ordering**, the classes are sorted in order of decreasing "importance" such as by decreasing *order of prevalence*. That is, all the rules for the most prevalent (or most frequent) class come first, the rules for the next prevalent class come next, and so on. Alternatively, they may be sorted based on the misclassification cost per class. Within each class, the rules are not ordered—they don't have to be because they all predict the same class (and so there can be no class conflict!).

With **rule-based ordering**, the rules are organized into one long priority list, according to some measure of rule quality, such as accuracy, coverage, or size (number of attribute tests in the rule antecedent), or based on advice from domain experts. When rule ordering is used, the rule set is known as a **decision list**. With rule ordering, the triggering rule that appears earliest in the list has the highest priority, and so it gets to fire its class prediction. Any other rule that satisfies $\boldsymbol{X}$ is ignored. Most rule-based classification systems use a class-based rule-ordering strategy.

Note that in the first strategy, overall the rules are *unordered*. They can be applied in any order when classifying a tuple. That is, a disjunction (logical OR) is implied between different rules. Each rule represents a standalone nugget or piece of knowledge. This is in contrast to the rule ordering (decision list) scheme for which rules must be applied in the prescribed order so as to avoid conflicts. Each rule in a decision list implies the negation of the rules that come before it in the list. Hence, rules in a decision list are more difficult to interpret.

Now that we have seen how we can handle conflicts, let's go back to the scenario where there is no rule satisfied by $\boldsymbol{X}$. How, then, can we determine the class label of $\boldsymbol{X}$? In this case, a fallback or **default rule** can be set up to specify a default class, based on a training set. This may be the class in majority or the majority class of the tuples that were not covered by any rule. The default rule is evaluated at the end, if and only if no other rule covers $\boldsymbol{X}$. The condition in the default rule is empty. In this way, the rule fires when no other rule is satisfied.

In the following sections, we examine how to build a rule-based classifier.

8.4.2 Rule Extraction from a Decision Tree

In Section 7.2, we learned how to build a decision tree classifier from a set of training data. Decision tree classifiers are a popular method of classification—it is easy to understand 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).

Example 8.4.1 **Extracting classification rules from a decision tree.** The decision tree of Figure 7.2 can be converted to classification IF-THEN rules by tracing the path from the root node to each leaf node in the tree. The rules extracted from Figure 7.2 are as follows:

$R1$: IF	$age = youth$	AND $student = no$	THEN $buys\_computer = no$
$R2$: IF	$age = youth$	AND $student = yes$	THEN $buys\_computer = yes$
$R3$: IF	$age = middle\_aged$		THEN $buys\_computer = yes$
$R4$: IF	$age = senior$	AND $credit\_rating = excellent$	THEN $buys\_computer = yes$
$R5$: IF	$age = senior$	AND $credit\_rating = fair$	THEN $buys\_computer = no$

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. As an example, Figure 7.7 shows decision trees that suffer from subtree repetition and replication. 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 compensate 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.

8.4.3 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 in Figure 8.10. 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 as many as possible) of the training tuples of class C and none (or as few as possible) of the tuples 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 (Figure 8.11). 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 conjunction to the existing condition of the rule antecedent. Suppose 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 at-

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;
```

Figure 8.10: Basic sequential covering algorithm.

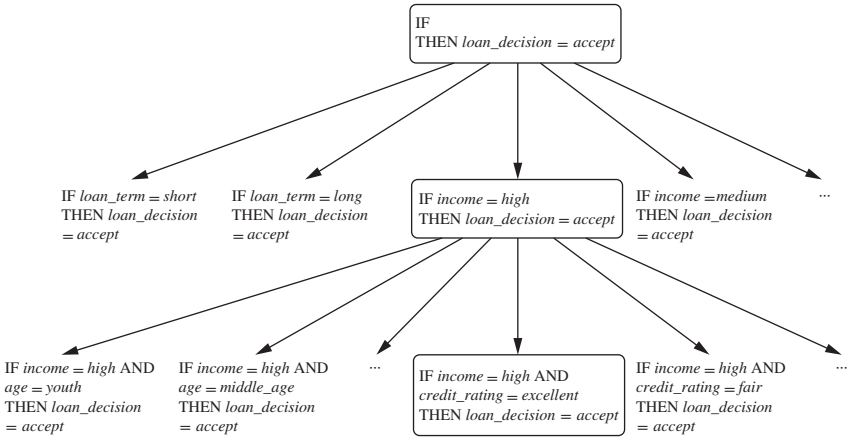


Figure 8.11: A general-to-specific search through rule space.

tributes with their associated values. For example, for an attribute–value pair (att, val), we can consider attribute tests such as $att = val$, $att \leq val$, $att > val$, and so on. Typically, the training data will contain many attributes, each of which may have several possible values. Finding 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 (conjunction) to the current rule, it picks the one that improves the rule quality most, 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. Getting back to our example with Figure 8.11, 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.

Rule Quality Measures

Learn_One_Rule needs a measure of rule quality. Every time it considers an attribute test, it must check to see if appending such a test to the current rule's condition will result in an improved rule. Accuracy may seem like an obvious choice at first, but consider Example 8.4.1.

Example 8.4.1 **Choosing between two rules based on accuracy.** Consider the two rules as illustrated in Figure 8.12. Both are for the class *loan_decision* = *accept*. We use "*a*" to represent the tuples of class "*accept*" and "*r*" for the tuples of class "*reject*." Rule $R1$ correctly classifies 38 of the 40 tuples it covers. Rule $R2$ covers only two tuples, which it correctly classifies. Their respective accuracies are 95% and 100%. Thus, $R2$ has greater accuracy than $R1$, but it is not the better rule because of its small coverage.

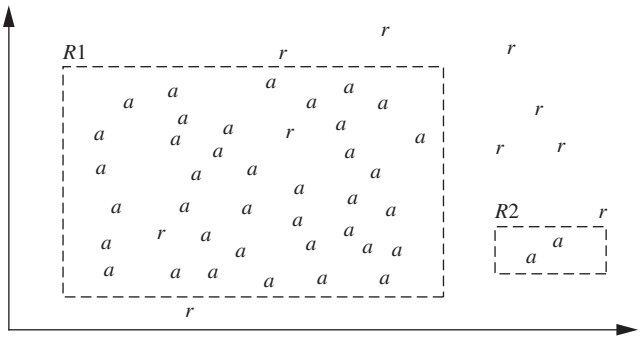


Figure 8.12: Rules for the class *loan_decision* = *accept*, showing *accept* (a) and *reject* (r) tuples.

From this example, we see that accuracy on its own is not a reliable estimate of rule quality. Coverage on its own is not useful either—for a given class we could have a rule that covers many tuples, most of which belong to other classes! Thus, we seek other measures for evaluating rule quality, which may integrate aspects of accuracy and coverage. Here we will look at a few, namely *entropy*, another based on *information gain*, and a *statistical test* that considers coverage. For our discussion, suppose we are learning rules for the class c. Our current rule is R: IF *condition* THEN $class = c$. We want to see if logically ANDing a given attribute test to *condition* would result in a better rule. We call the new condition, *condition*$'$, where R': IF *condition*$'$ THEN $class = c$ is our potential new rule. In other words, we want to see if R' is any better than R.

We have already seen entropy in our discussion of the information gain measure used for attribute selection in decision tree induction. It is also known as the *expected information* needed to classify a tuple in data set, D. Here, D is the set of tuples covered by *condition*$'$ and p_i is the probability of class C_i in D. The lower the entropy, the better *condition*$'$ is. Entropy prefers conditions that cover a large number of tuples of a single class and few tuples of other classes.

Another measure is based on information gain and was proposed in **FOIL** (First Order Inductive Learner), a sequential covering algorithm that learns first-order logic rules. Learning first-order rules is more complex because such rules contain variables, whereas the rules we are concerned with in this section are propositional (i.e., variable-free).[5] In machine learning, the tuples of the class for which we are learning rules are called *positive* tuples, while the remaining tuples are *negative*. Let pos (neg) be the number of positive (negative) tuples covered by R. Let pos' (neg') be the number of positive (negative) tuples covered by R'. FOIL assesses the information gained by extending *condition*$'$ as

$$FOIL\_Gain = pos' \times \left(\log_2 \frac{pos'}{pos' + neg'} - log_2 \frac{pos}{pos + neg} \right). \tag{8.20}$$

[5]Incidentally, FOIL was also proposed by Quinlan, the father of ID3.

It favors rules that have high accuracy and cover many positive tuples.

We can also use a statistical test of significance to determine if the apparent effect of a rule is not attributed to chance but instead indicates a genuine correlation between attribute values and classes. The test compares the observed distribution among classes of tuples covered by a rule with the expected distribution that would result if the rule made predictions at random. We want to assess whether any observed differences between these two distributions may be attributed to chance. We can use the **likelihood ratio statistic**,

$$Likelihood\_Ratio = 2\sum_{i=1}^{m} f_i \log\left(\frac{f_i}{e_i}\right), \tag{8.21}$$

where m is the number of classes.

For tuples satisfying the rule, f_i is the observed frequency of each class i among the tuples. e_i is what we would expect the frequency of each class i to be if the rule made random predictions. The statistic has a χ^2 distribution with $m-1$ degrees of freedom. The higher the likelihood ratio, the more likely that there is a *significant* difference in the number of correct predictions made by our rule in comparison with a "random guessor." That is, the performance of our rule is not due to chance. The ratio helps identify rules with insignificant coverage.

CN2 uses entropy together with the likelihood ratio test, while FOIL's information gain is used by RIPPER.

Rule Pruning

Learn_One_Rule does not employ a test set when evaluating rules. Assessments of rule quality as described previously are made with tuples from the original training data. These assessments are optimistic because the rules will likely overfit the data. That is, the rules may perform well on the training data, but less well on subsequent unseen data (i.e., test data). To compensate for this, we can prune the rules. A rule is pruned by removing a conjunction (attribute test). We choose to prune a rule, R, if the pruned version of R has greater quality, as assessed on an independent set of tuples. As in decision tree pruning, we refer to this set as a *pruning set*.

FOIL uses a simple yet effective method. Given a rule, R,

$$FOIL\_Prune(R) = \frac{pos - neg}{pos + neg}, \tag{8.22}$$

where *pos* and *neg* are the number of positive and negative tuples covered by R, respectively. This value will increase with the accuracy of R on a pruning set. Therefore, if the *FOIL_Prune* value is higher for the pruned version of R, then we prune R.

By convention, RIPPER starts with the most recently added conjunction when considering pruning. Conjunctions are pruned one at a time as long as this results in an improvement.

8.4.4 Associative Classification

In this section, you will learn about associative classification. The methods discussed are CBA, CMAR, and CPAR.

Before we begin, however, let's look at association rule mining in general. Association rules are mined in a two-step process consisting of *frequent itemset mining* followed by *rule generation*. The first step searches for patterns of attribute–value pairs that occur repeatedly in a data set, where each attribute–value pair is considered an *item*. The resulting attribute–value pairs form *frequent itemsets* (also referred to as *frequent patterns*). The second step analyzes the frequent itemsets to generate association rules. All association rules must satisfy certain criteria regarding their "accuracy" (or *confidence*) and the proportion of the data set that they actually represent (referred to as *support*). For example, the following is an association rule mined from a data set, D, shown with its confidence and support:

$$age = youth \wedge credit = OK \Rightarrow buys\_computer = yes \quad [support = 20\%, confidence = 93\%], \tag{8.23}$$

where $\wedge$ represents a logical "AND". We will say more about confidence and support later.

More formally, let D be a data set of tuples. Each tuple in D is described by n attributes, $A_1, A_2, \ldots, A_n$, and a class label attribute, A_{class}. All continuous attributes are discretized and treated as categorical (or nominal) attributes. An **item**, p, is an attribute–value pair of the form (A_i, v), where A_i is an attribute taking a value, v. A data tuple $\boldsymbol{X} = (x_1, x_2, \ldots, x_n)$ satisfies an item, $p = (A_i, v)$, if and only if $x_i = v$, where x_i is the value of the ith attribute of $\boldsymbol{X}$. Association rules can have any number of items in the rule antecedent (left side) and any number of items in the rule consequent (right side). However, when mining association rules for use in classification, we are only interested in association rules of the form $p_1 \wedge p_2 \wedge \ldots p_l \Rightarrow A_{class} = C$, where the rule antecedent is a conjunction of items, $p_1, p_2, \ldots, p_l$ $(l \leq n)$, associated with a class label, C. For a given rule, R, the percentage of tuples in D satisfying the rule antecedent that also have the class label C is called the **confidence** of R.

From a classification point of view, this is akin to rule accuracy. For example, a confidence of 93% for Rule in Eq. (8.23) means that 93% of the customers in D who are young and have an OK credit rating belong to the class *buys_computer = yes*. The percentage of tuples in D satisfying the rule antecedent and having class label C is called the **support** of R. A support of 20% for Rule in Eq. (8.23) means that 20% of the customers in D are young, have an OK credit rating, and belong to the class *buys_computer = yes*.

In general, associative classification consists of the following steps:

1. Mine the data for frequent itemsets, that is, find commonly occurring attribute–value pairs in the data.

2. Analyze the frequent itemsets to generate association rules per class, which satisfy confidence and support criteria.

3. Organize the rules to form a rule-based classifier.

Methods of associative classification differ primarily in the approach used for frequent itemset mining and in how the derived rules are analyzed and used for classification. We now look at some of the various methods for associative classification.

One of the earliest and simplest algorithms for associative classification is **CBA** (Classification Based on Associations). CBA uses an iterative approach to frequent itemset mining, similar to that described for Apriori in Section 6.2.1**hh: cross check the section number later**, where multiple passes are made over the data and the derived frequent itemsets are used to generate and test longer itemsets. In general, the number of passes made is equal to the length of the longest rule found. The complete set of rules satisfying minimum confidence and minimum support thresholds are found and then analyzed for inclusion in the classifier. CBA uses a heuristic method to construct the classifier, where the rules are ordered according to decreasing precedence based on their confidence and support. If a set of rules has the same antecedent, then the rule with the highest confidence is selected to represent the set. When classifying a new tuple, the first rule satisfying the tuple is used to classify it. The classifier also contains a default rule, having the lowest precedence, which specifies a default class for any new tuple that is not satisfied by any other rule in the classifier. In this way, the set of rules making up the classifier form a *decision list*. In general, CBA was empirically found to be more accurate than C4.5 on a good number of data sets.

CMAR (Classification based on Multiple Association Rules) differs from CBA in its strategy for frequent itemset mining and its construction of the classifier. It also employs several rule pruning strategies with the help of a tree structure for efficient storage and retrieval of rules. CMAR adopts a variant of the *FP-growth* algorithm to find the complete set of rules satisfying the minimum confidence and minimum support thresholds. FP-growth was described in Section 6.2.4**hh: cross check the section number later**. FP-growth uses a tree structure, called an *FP-tree*, to register all the frequent itemset information contained in the given data set, D. This requires only two scans of D. The frequent itemsets are then mined from the FP-tree. CMAR uses an enhanced FP-tree that maintains the distribution of class labels among tuples satisfying each frequent itemset. In this way, it is able to combine rule generation together with frequent itemset mining in a single step.

CMAR employs another tree structure to store and retrieve rules efficiently and to prune rules based on confidence, correlation, and database coverage. Rule pruning strategies are triggered whenever a rule is inserted into the tree. For example, given two rules, $R1$ and $R2$, if the antecedent of $R1$ is more general than that of $R2$ and $conf(R1) \geq conf(R2)$, then $R2$ is pruned. The rationale is that highly specialized rules with low confidence can be pruned if a more generalized version with higher confidence exists. CMAR also prunes rules for which the rule antecedent and class are not positively correlated, based on an χ^2 test of statistical significance.

"*If more than one rule applies, which one do we use?*" As a classifier, CMAR operates differently than CBA. Suppose that we are given a tuple $\boldsymbol{X}$ to classify and that only one rule satisfies or matches $\boldsymbol{X}$.[6] This case is trivial—we simply assign the rule's class label. Suppose, instead, that more than one rule satisfies $\boldsymbol{X}$. These rules form a set, S. Which rule would we use to determine the class label of $\boldsymbol{X}$? CBA would assign the class label of the most confident rule among the rule set, S. CMAR instead considers multiple rules when making its class prediction. It divides the rules into groups according to class labels. All rules within a group share the same class label and each group has a distinct class label.

CMAR uses a weighted χ^2 measure to find the "strongest" group of rules, based on the statistical correlation of rules within a group. It then assigns $\boldsymbol{X}$ the class label of the strongest group. In this way it considers multiple rules, rather than a single rule with highest confidence, when predicting the class label of a new tuple. In experiments, CMAR had slightly higher average accuracy in comparison with CBA. Its runtime, scalability, and use of memory were found to be more efficient.

"*Is there a way to cut down on the number of rules generated?*" CBA and CMAR adopt methods of frequent itemset mining to generate *candidate* association rules, which include all conjunctions of attribute–value pairs (items) satisfying minimum support. These rules are then examined, and a subset is chosen to represent the classifier. However, such methods generate quite a large number of rules. **CPAR** (Classification based on Predictive Association Rules) takes a different approach to rule generation, based on a rule generation algorithm for classification known as FOIL (Section 8.4.3). FOIL builds rules to distinguish positive tuples (e.g., *buys_computer* = *yes*) from negative tuples (e.g., *buys_computer* = *no*). For multiclass problems, FOIL is applied to each class. That is, for a class, C, all tuples of class C are considered positive tuples, while the rest are considered negative tuples. Rules are generated to distinguish C tuples from all others. Each time a rule is generated, the positive samples it satisfies (or *covers*) are removed until all the positive tuples in the data set are covered. In this way, fewer rules are generated. CPAR relaxes this step by allowing the covered tuples to remain under consideration, but reducing their weight. The process is repeated for each class. The resulting rules are merged to form the classifier rule set.

During classification, CPAR employs a somewhat different multi-rule strategy than CMAR. If more than one rule satisfies a new tuple, $\boldsymbol{X}$, the rules are divided into groups according to class, similar to CMAR. However, CPAR uses the best k rules of each group to predict the class label of $\boldsymbol{X}$, based on expected accuracy. By considering the best k rules rather than all of a group's rules, it avoids the influence of lower-ranked rules. CPAR's accuracy on numerous data sets was shown to be close to that of CMAR. However, since CPAR generates far fewer rules than CMAR, it shows much better efficiency with large sets of training data.

[6] If a rule's antecedent satisfies or matches $\boldsymbol{X}$, then we say that the rule satisfies $\boldsymbol{X}$.

In summary, associative classification offers an alternative classification scheme by building rules based on conjunctions of attribute–value pairs that occur frequently in data.

8.4.5 Discriminative Frequent Pattern–Based Classification

From work on associative classification, we see that frequent patterns reflect strong associations between attribute–value pairs (or items) in data and are useful for classification.

"*But just how discriminative are frequent patterns for classification?*" Frequent patterns represent feature combinations. Let's compare the discriminative power of frequent patterns and single features. Figure 8.13 plots the information gain of frequent patterns and single features (i.e., of pattern length 1) for three UCI data sets[7]. The discrimination power of some frequent patterns is higher than that of single features. Frequent patterns map data to a higher dimensional space. They capture more underlying semantics of the data, and thus can hold greater expressive power than single features.

"*Why not consider frequent patterns as combined features, in addition to single features when building a classification model?*" This notion is the basis of **frequent pattern–based classification**—the learning of a classification model in the feature space of single attributes *as well as* frequent patterns. In this way, we transfer the original feature space to a larger space. This will likely increase the chance of including important features.

Let's get back to our earlier question: How discriminative are frequent patterns? Many of the frequent patterns generated in frequent itemset mining are indiscriminative because they are solely based on support, without considering predictive power. That is, by definition, a pattern must satisfy a user-specified minimum support threshold, *min_sup*, to be considered frequent. For example, if *min_sup*, is, say, 5%, a pattern is frequent if it occurs in 5% of the data tuples. Consider Figure 8.14, which plots information gain versus pattern frequency (support) for three UCI data sets. A theoretical upper bound on information gain, which was derived analytically, is also plotted. The figure shows that the discriminative power (assessed here as information gain) of low-frequency patterns is bounded by a small value. This is due to the patterns' limited coverage of the data set. Similarly, the discriminative power of very high-frequency patterns is also bounded by a small value, which is due to their commonness in the data. The upper bound of information gain is a function of pattern frequency. These observations can be confirmed analytically. Patterns with medium-large supports (e.g., $support = 300$ in Figure 8.14(a)) may be discriminative or not. Thus, not every frequent pattern is useful.

If we were to add all the frequent patterns to the feature space, the resulting feature space would be huge. This slows down the model learning process and

[7]The University of California at Irvine (UCI) archives several large data sets at *http://kdd.ics.uci.edu/*. These are commonly used by researchers for the testing and comparison of machine learning and data mining algorithms.

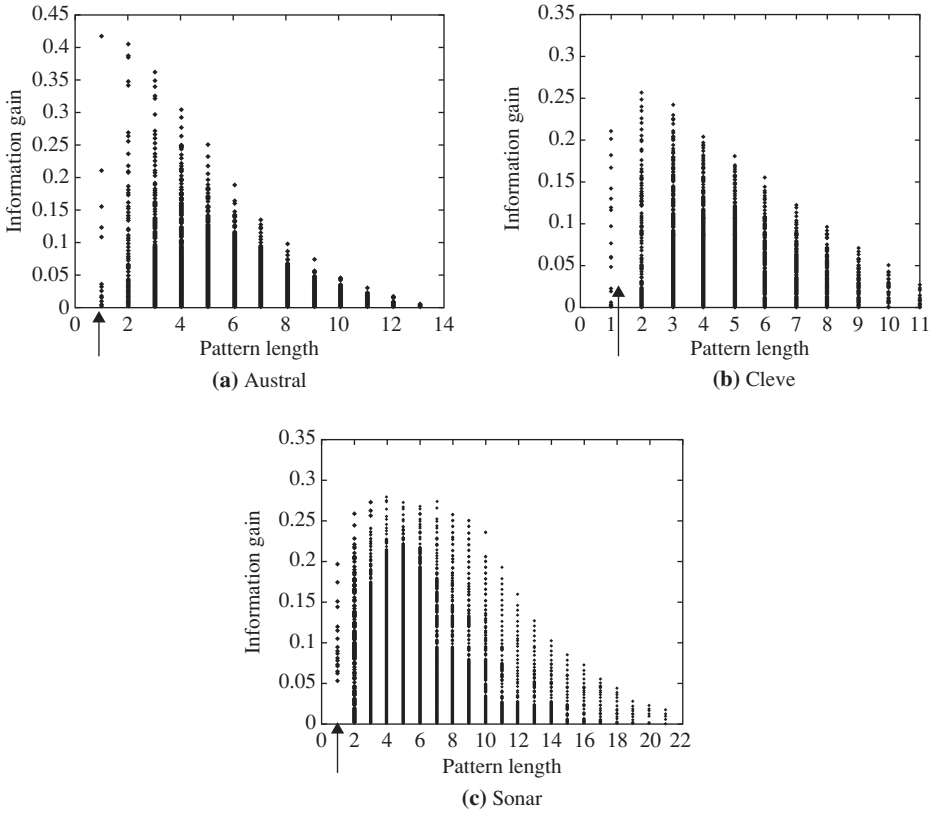


Figure 8.13: Single feature versus frequent pattern: Information gain is plotted for single features (patterns of length 1, indicated by arrows) and frequent patterns (combined features) for three UCI data sets. *Source:* Adapted from Cheng, Yan, Han, and Hsu [CYHH07].

may also lead to decreased accuracy due to a form of overfitting in which there are too many features. Many of the patterns may be redundant. Therefore, it's a good idea to apply feature selection to eliminate the less discriminative and redundant frequent patterns as features. The *general framework for discriminative frequent pattern–based classification* is as follows.

1. **Feature generation:** The data, D, are partitioned according to class label. Use frequent itemset mining to discover frequent patterns in each partition, satisfying minimum support. The collection of frequent patterns, $\mathcal{F}$, makes up the feature candidates.

2. **Feature selection:** Apply feature selection to $\mathcal{F}$, resulting in $\mathcal{F}_S$, the set of selected (more discriminating) frequent patterns. Information gain, Fisher score, or other evaluation measures can be used for this step. Relevancy checking can also be incorporated into this step to weed out redundant patterns. The data set D is transformed to D', where the feature space now includes the single features as well as the selected frequent pat-

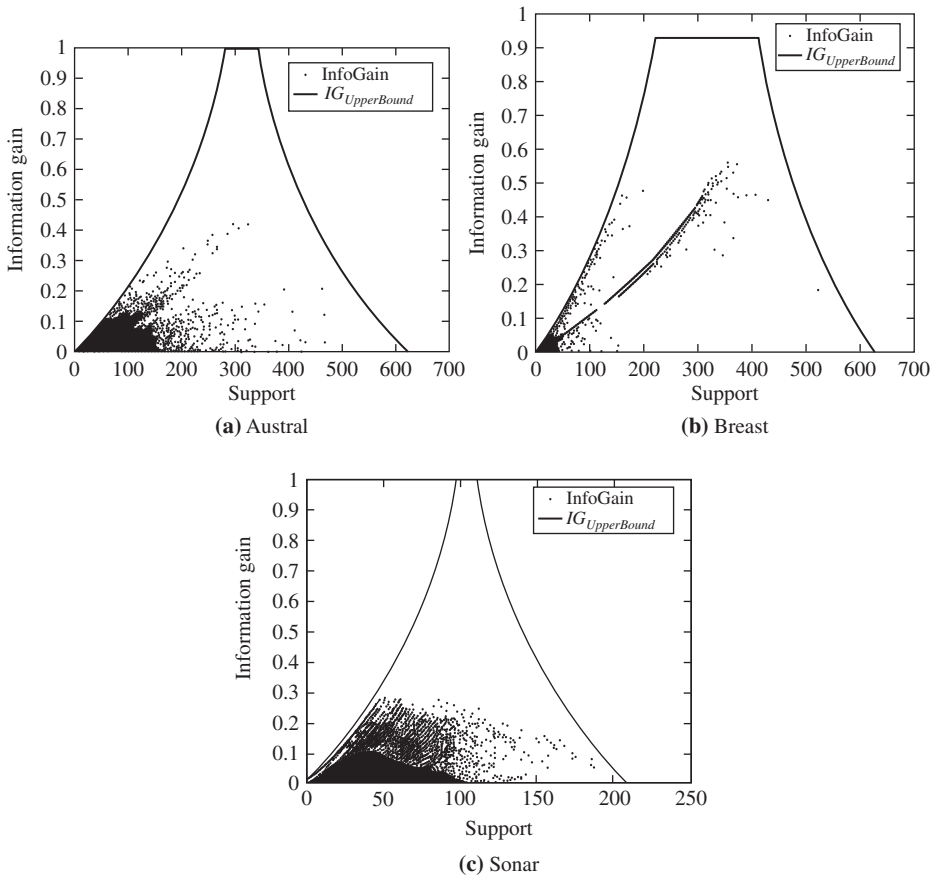


Figure 8.14: Information gain versus pattern frequency (support) for three UCI data sets. A theoretical upper bound on information gain ($IG_{UpperBound}$) is also shown. *Source:* Adapted from Cheng, Yan, Han, and Hsu [CYHH07].

terns, $\mathcal{F}_{\mathcal{S}}$. Commonly used feature selection methods were introduced in Section 8.1.

3. **Learning of classification model:** A classifier is built on the data set D'. Any learning algorithm can be used as the classification model.

The general framework is summarized in Figure 8.15(a), where the discriminative patterns are represented by dark circles. Although the approach is straightforward, we can encounter a computational bottleneck by having to first find *all* the frequent patterns, and then analyze *each one* for selection. The amount of frequent patterns found can be huge due to the explosive number of pattern combinations between items.

To improve the efficiency of the general framework, consider condensing steps 1 and 2 into just one step. That is, rather than generating the complete set of frequent patterns, it's possible to mine only the highly discriminative ones.

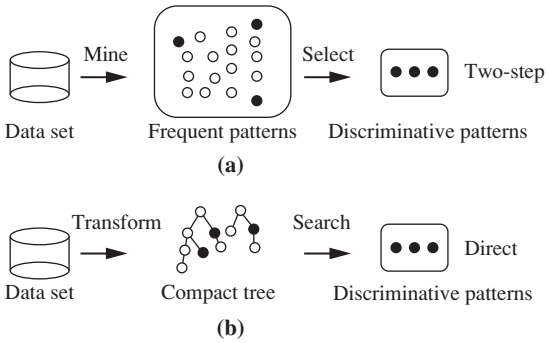


Figure 8.15: A framework for frequent pattern–based classification: (a) a two-step general approach versus (b) the direct approach of DDPMine.

This more direct approach is referred to as **DDPMine** (Direct Discriminative Pattern Mining). The DDPMine algorithm follows this approach, as illustrated in Figure 8.15(b). It first transforms the training data into a compact tree structure known as a frequent pattern tree, or FP-tree (Section 6.2.4**hh: cross check section number later**), which holds all of the attribute–value (itemset) association information. It then searches for discriminative patterns on the tree. The approach is direct in that it avoids generating a large number of indiscriminative patterns. It incrementally reduces the problem by eliminating training tuples, thereby progressively shrinking the FP-tree. This further speeds up the mining process.

By choosing to transform the original data to an FP-tree, DDPMine avoids generating redundant patterns because an FP-tree stores only the *closed* frequent patterns. By definition, any subpattern, β, of a closed pattern, α, is redundant with respect to α (Section 6.1.2**hh: cross check section number later**). DDPMine directly mines the discriminative patterns and integrates feature selection into the mining framework. The theoretical upper bound on information gain is used to facilitate a branch-and-bound search, which prunes the search space significantly. Experimental results show that DDPMine achieves orders of magnitude speedup over the two-step approach without decline in classification accuracy. DDPMine also outperforms state-of-the-art associative classification methods in terms of both accuracy and efficiency.

Compared with associative classifiers, DDPMine is able to prune a huge number of non-discriminative frequent patterns. However, DDPMine might still use hundreds or even thousands of frequent patterns in the classification model. *How can we further reduce the number of patterns to build a more compact classifier?* This will not only speed up the computation, but also make the classifier more explainable to the end users. **DPClass** (Discriminative Pattern-based Classification) addresses this issue by combining the strength of two methods, including tree-based classifiers (e.g., decision tree classifier, random forest, etc., which were introduced in Chapter 7.2) and feature selection (e.g., forward se-

lection, LASSO, etc., which were introduced in Section 8.1). DPClass works as follows. First, it uses random forest which contains multiple tree-based classifiers. Then, each prefix path from the root of a tree of random forest to its non-leaf node is treated as a discriminative pattern. Finally, it leverages feature selection, including forward feature selection and LASSO based method, to select a small subset of highly discriminative patterns to construct a linear classifier, such as logistic regression classifier or linear SVMs. The empirical evaluations on a good number of UCI datasets shows that DPClass performs similar as or better than DDPMine. On the other hand, DPClass uses a significantly less number of discriminative patterns than DDPMine. Therefore, the classifier generated by DPClass is more compact, making itself faster in test and more explainable to end users.

8.5 Classification with Weak Supervision

The effectiveness of the classifiers we have introduced so far (e.g., SVMs, logistic regression, k-NN) largely depends on 'strong supervision'. It means that in order to train a highly accurate classifier, we typically need a large number of high-quality training tuples, and the true class label for each training tuple is accurately annotated, say by the domain experts. But, what if there is only a small number of labeled training tuples? Document classification, speech recognition, computer vision, and information extraction are just a few examples of applications in which unlabeled data are abundant. Consider *document classification*, for example. Suppose we want to build a model to automatically classify text documents like articles or web pages. In particular, we want the model to distinguish between hockey and football documents. We have a vast amount of documents available, yet the documents are not class-labeled. Recall that supervised learning requires a training set, that is, a set of class-labeled data. To have a human examine and assign a class label to individual documents (to form a training set) is time consuming and expensive. *Speech recognition* requires the accurate labeling of speech utterances by trained linguists. It was reported that 1 minute of speech takes 10 minutes to label, and annotating phonemes (basic units of sound) can take 400 times as long. *Information extraction systems* are trained using labeled documents with detailed annotations. These are obtained by having human experts highlight items or relations of interest in text such as the names of companies or individuals. High-level expertise may be required for certain knowledge domains such as gene and disease mentions in biomedical information extraction. Clearly, the manual assignment of class labels to prepare a training set can be extremely costly, time consuming, and tedious. In computer vision, a fundamental task is to build a highly accurate classifier to automatically recognize various objects (i.e., class labels). But some objects (e.g., a new type of dog) might appear only *after* the classifier has been built. In other words, there are no training tuples at all for the newly appeared class label. How can the classifier still recognize the test image of such a new type of dog?

We study five approaches for classification that are suitable for situations where there is only a limited number or no labeled training tuples. Section 8.5.1 introduces *semi-supervised classification*, which builds a classifier using both labeled and unlabeled data. Section 8.5.2 presents *active learning*, where the learning algorithm carefully selects a few of the unlabeled data tuples and asks a human to label only those tuples. Section 8.5.3 presents *transfer learning*, which aims to extract the knowledge from one or more source classification tasks (e.g., classifying camera reviews) and apply the knowledge to a target classification task (e.g., classifying TV reviews). Section 8.5.4 studies *distant supervision* whose key idea is to automatically obtain a large number of inexpensive, but potentially noisy labeled training tuples. Finally, Section 8.5.5 introduces *zero-shot learning*, which deals with the case there are no training tuples for certain class labels at all. Each of these strategies can reduce the need to annotate large amounts of data, resulting in cost and time savings. In comparison to the traditional setting which requires 'strong supervision' (i.e., a large number of high-quality labeled tuples are available to train the classifier), we collectively refer to these approaches as **classification with weak supervision**.

Other forms of weak supervision exist. To name a few, *crowdsourcing learning* aims to train a classification model with a *noisy training set*. Here, the class labels are provided by workers on a crowdsourcing platform (e.g., Amazon Mechanical Turk), where we can often obtain a large amount of labeled training tuples with a relatively low cost. However, some (or many) labels provided by the crowdsourcing workers might be wrong. How to infer the true label (i.e., the ground truth) from the noisy labels is a major concern of crowdsourcing learning. Crowdsourcing learning can be viewed as a form of weakly supervised learning in that the supervision (i.e., labels) are noisy or inaccurate. In *multi-instance learning*, each training tuple (e.g., an image, a document) is called a *bag*, which consists of a set of *instances* (e.g., different regions of an image, different sentences of a document). A bag is labeled as a positive bag, as long as at least one of its instances is assigned with a positive class label. A bag is labeled as a negative bag if none of its instances has a positive class label. For example, an image is labeled as 'beach' if at least one of its regions is about beach; and it is labeled as 'non-beach' if none of its regions is about beach. A document is labeled as positive sentiment if at least one of its sentences has expressed positive sentiment, **yu: is this the convention in sentiment analysis?** and it is labeled as negative sentiment if none of its sentences expressed positive sentiment. Given a set of labeled bags, the goal of multi-instance learning is to train a classifier to predict the label of a test (previously unseen) bag. Multi-instance learning can be viewed as a form of weakly supervised learning, in that the label (i.e., supervision) is provided at a coarse granularity (i.e., at the bag level instead of instance level). The label of a bag is also called *group-level* label (e.g., a group of regions of an image, a group of sentences of a document).

8.5.1 Semi-Supervised Classification

Semi-supervised classification uses both labeled data and unlabeled data to build a classifier. Let $X_l = \{(x_1, y_1), \ldots, (x_l, y_l)\}$ be the set of labeled data and $X_u = \{x_{l+1}, \ldots, x_n\}$ be the set of unlabeled data. Here we describe a few examples of this approach for learning.

Self-training

1. Select a learning method such as, say, Bayesian classification. Build the classifier using the labeled data, X_l.
2. Use the classifier to label the unlabeled data, X_u.
3. Select the tuple $x \in X_u$ having the highest confidence (most confident prediction). Add it and its predicted label to X_l.
4. Repeat (i.e., retrain the classifier using the augmented set of labeled data).

Co-training

1. Define two separate nonoverlapping feature sets for the labeled data, X_l.
2. Train two classifiers, f_1 and f_2, on the labeled data, where f_1 is trained using one of the feature sets and f_2 is trained using the other.
3. Classify X_u with f_1 and f_2 separately.
4. Add the most confident $(x, f_1(x))$ to the set of labeled data used by f_2, where $x \in X_u$. Similarly, add the most confident $(x, f_2(x))$ to the set of labeled data used by f_1.
5. Repeat.

Figure 8.16: Self-training and co-training methods of semi-supervised classification.

Self-training is the simplest form of semi-supervised classification. It first builds a classifier using the labeled data. The classifier then tries to label the unlabeled data. The tuple with the most confident label prediction is added to the set of labeled data, and the process repeats (Figure 8.16). Although the method is easy to understand, a disadvantage is that it may reinforce errors.

Co-training is another form of semi-supervised classification, where two or more classifiers teach each other. Each learner uses a different and ideally independent set of features for each tuple. Consider web page data, for example, where attributes relating to the images on the page may be used as one set of features, while attributes relating to the corresponding text constitute another set of features for the same data. Each set of features (called 'a view') should be sufficient to train a good classifier. Suppose we split the feature set into two sets and train two classifiers, f_1 and f_2, where each classifier is trained on a different set. Then, f_1 and f_2 are used to predict the class labels for the unlabeled data, X_u. Each classifier then teaches the other in that the tuple having the most confident prediction from f_1 is added to the set of labeled data for f_2 (along with its predicted label).

Similarly, the tuple having the most confident prediction from f_2 is added to the set of labeled data for f_1. The method is summarized in Figure 8.16. Co-training is less sensitive to errors than self-training. A difficulty is that the assumptions for its usage may not hold true, that is, it may not be possible to

split the features into mutually exclusive and class-conditionally independent sets.

Alternate approaches to semi-supervised learning exist. For example, we can model the joint probability distribution of the features and the labels. For the unlabeled data, the labels can then be treated as missing data. The EM algorithm (Chapter 10) can be used to maximize the likelihood of the model. Semi-supervised classification methods using support vector machines have also been proposed.

"*When does semi-supervised classification work?*" Generally speaking, there are two commonly used assumptions behind semi-supervised learning. The first assumption is *clustering assumption*, which means that data tuples from the same cluster are likely to share the same class label. The clustering algorithms will be introduced in Chapters 9-10. A representative example that utilizes the clustering assumption is semi-supervised support vector machines (S3VMs). Recall that in the standard SVMs (Section 8.3), we seek a max-margin hyperphane that correctly separates the positive training tuples from negative tuples with a large margin. In S3VMs, it considers two design objectives, including (1) seeking a max-margin hyperplane to separate positive tuples from negative ones (which is the same as standard SVMs), and (2) avoiding to disrupt the clustering structure of unlabeled tuples. For the latter, this means that we favor a classifier (e.g., a hyperplane) that goes through the low density region of the unlabeled tuples. The second commonly used assumption behind semi-supervised learning is manifold assumption. We will not go into the technical details of manifold.[8] Simply put, the manifold assumption in the contexts of classification means that a pair of close tuples are likely to share the same class label. A representative example that utilizes the manifold assumption is graph-based semi-supervised classification. It works as follows. First, we construct a graph whose nodes are input tuples, including both labeled and unlabeled tuples, and the edges indicate the local proximity. For example, we can link each data tuple to its k-nearest neighbors. In the constructed graph, only a small handful of nodes are labeled and the vast majority are unlabeled. The classification method propagates the labels of these labeled nodes (i.e., tuples) to the unlabeled nodes.

8.5.2 Active Learning

Active learning is an iterative type of supervised learning that is suitable for situations where data are abundant, yet the class labels are scarce or expensive to obtain. The learning algorithm is active in that it can purposefully query a user (e.g., a human annotator) for labels. The number of tuples used to learn a concept this way is often much smaller than the number required in typical supervised learning.

"*How does active learning work to overcome the labeling bottleneck?*" To keep costs down, the active learner aims to achieve high accuracy using as few

[8] In mathematical terms, a manifold is a topological space which approximates the Euclidean space in the vicinity of each data point.

labeled instances as possible. Let D be all of data under consideration. Various strategies exist for active learning on D. Figure 8.17 illustrates a *pool-based approach* to active learning. Suppose that a small subset of D is class-labeled. This set is denoted L. U is the set of unlabeled data in D. It is also referred to as a pool of unlabeled data. An active learner begins with L as the initial training set. It then uses a *querying function* to carefully select one or more data samples from U and requests labels for them from an oracle (e.g., a human annotator). The newly labeled samples are added to L, which the learner then uses in a standard supervised way. The process repeats. The goal of active learning is to achieve high accuracy using as few labeled tuples as possible. Active learning algorithms are typically evaluated with the use of learning curves, which plot accuracy as a function of the number of instances queried.

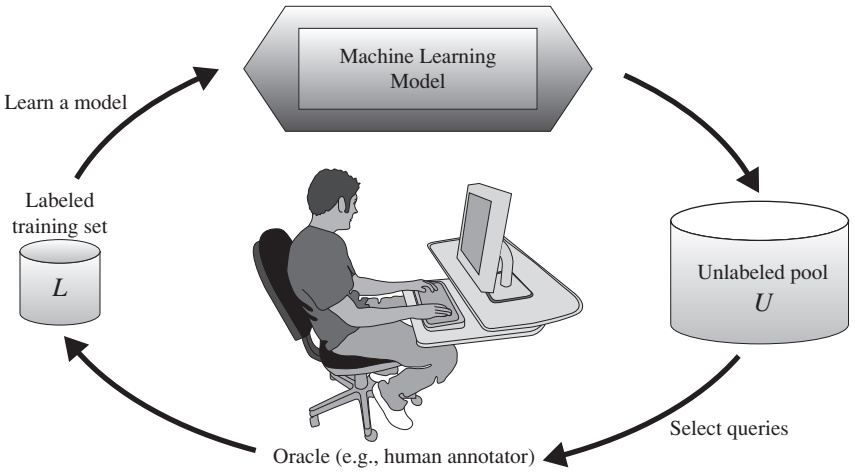


Figure 8.17: The pool-based active learning cycle. *Source:* From Settles [Set10], Burr Settles Computer Sciences Technical Report 1648, University of Wisconsin–Madison; used with permission.

Most of the active learning research focuses on how to *choose* the data tuples to be queried. Several frameworks have been proposed. *Uncertainty sampling* is the most common strategy, where the active learner chooses to query the tuples which it is the least certain how to label. *Query-by-committee* is another commonly used active learning strategy. In this method, it constructs multiple (say 5) classification models, and then selects the unlabeled tuple that constructed classification models have most disagreement in terms of its predicted class labels (say 3 classifiers predict that it belongs to positive class, whereas 2 classifiers predicts it a negative tuple). Other strategies work to reduce the *version space*, that is, the subset of all hypotheses (i.e., classifiers) that are consistent with the observed training tuples. Alternatively, we may follow a decision-theoretic approach that estimates expected error reduction. This selects tuples that would result in the greatest reduction in the total number of incorrect predictions such

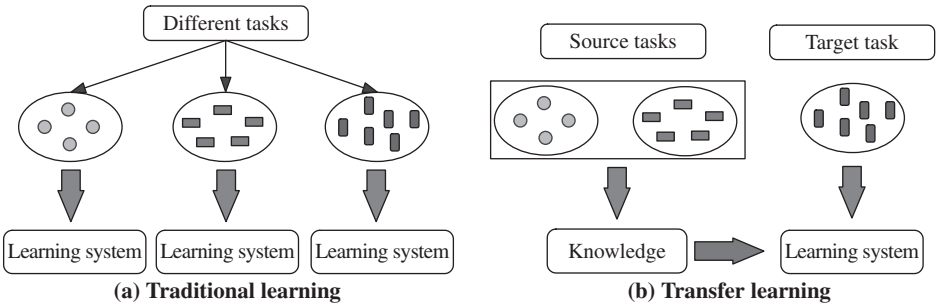


Figure 8.18: Transfer learning versus traditional learning. (a) Traditional learning methods build a new classifier from scratch for each classification task. (b) Transfer learning applies knowledge from a source classification task to simplify the construction of a classifier for a new, target classification task. *Source:* From Pan and Yang [PY10]; used with permission.

as by reducing the expected entropy over U. This latter approach tends to be more computationally expensive.

8.5.3 Transfer Learning

Suppose that *AllElectronics* has collected a number of customer reviews on a product such as a brand of camera. The classification task is to automatically label the reviews as either positive or negative. This task is known as **sentiment classification**. We could examine each review and annotate it by adding a *positive* or *negative* class label. The labeled reviews can then be used to train and test a classifier to label future reviews of the product as either positive or negative. The manual effort involved in annotating the review data can be expensive and time consuming.

Now, suppose that *AllElectronics* has customer reviews for other products as well such as TVs. The distributions of review data for different types of products can vary greatly. We cannot assume that the TV-review data will have the same distribution as the camera-review data; thus we must build a separate classification model for the TV-review data. Examining and labeling the TV-review data to form a training set will require a lot of effort. In fact, we would need to label a large amount of data to train the review-classification models for each product. It would be nice if we could adapt an existing classification model (e.g., the one we built for cameras) to help learn a classification model for TVs. Such *knowledge transfer* would reduce the need to annotate a large amount of data, resulting in cost and time savings. This is the essence behind *transfer learning*.

Transfer learning aims to extract the knowledge from one or more *source tasks* and apply the knowledge to a *target task*. In our example, the source task is the classification of camera reviews, and the target task is the classification of TV reviews. Figure 8.18 illustrates a comparison between traditional learning

methods and transfer learning. Traditional learning methods build a new classifier for each new classification task, based on available class-labeled training and test data. Transfer learning algorithms apply knowledge about source tasks when building a classifier for a new (target) task. Construction of the resulting classifier requires fewer training data and less training time. Traditional learning algorithms assume that the training data and test data are drawn from the same distribution and the same feature space. Thus, if the distribution changes, such methods need to rebuild the models from scratch.

Transfer learning allows the distributions, tasks, and even the data domains used in training and testing to be different. Transfer learning is analogous to the way humans may apply their knowledge of a task to facilitate the learning of another task. For example, if we know how to play the recorder, we may apply our knowledge of note reading and music to simplify the task of learning to play the piano. Similarly, knowing Spanish may make it easier to learn Italian.

Transfer learning is useful for common applications where the data becomes outdated or the distribution changes. Here we give two more examples. Consider *web-document classification*, where we may have trained a classifier to label, say, articles from various newsgroups according to predefined categories. The web data that were used to train the classifier can easily become outdated because the topics on the Web change frequently. Another application area for transfer learning is *email spam filtering*. We could train a classifier to label email as either "*spam*" or "*not spam*", using email from a group of users. If new users come along, the distribution of their email can be different from the original group, hence the need to adapt the learned model to incorporate the new data.

There are various approaches to transfer learning, the most common of which is the *instance-based transfer learning* approach. This approach reweights some of the data from the source task and uses it to learn the target task. The **TrAdaBoost** (Transfer AdaBoost) algorithm exemplifies this approach. Consider our previous example of web-document classification, where the distribution of the old data on which the classifier was trained (the source data) is different from the newer data (the target data). TrAdaBoost assumes that the source and target domain data are each described by the same set of attributes (i.e., they have the same "feature space") and the same set of class labels, but that the distributions of the data in the two domains are very different. It extends the AdaBoost ensemble method described in Section 7.7.3. TrAdaBoost requires the labeling of only a small amount of the target data. Rather than throwing out all the old source data, TrAdaBoost assumes that a large amount of it can be useful in training the new classification model. The idea is to filter out the influence of any old data that are very different from the new data by automatically adjusting weights assigned to the training tuples.

Recall that in boosting, an ensemble is created by learning a series of classifiers. To begin, each tuple is assigned a weight. After a classifier M_i is learned, the weights are updated to allow the subsequent classifier, M_{i+1}, to "pay more attention" to the training tuples that were misclassified by M_i. TrAdaBoost follows this strategy for the target data. However, if a source data tuple is

misclassified, TrAdaBoost reasons that the tuple is probably very different from the target data. It therefore *reduces* the weight of such tuples so that they will have less effect on the subsequent classifier. As a result, TrAdaBoost can learn an accurate classification model using only a small amount of new data and a large amount of old data, even when the new data alone are insufficient to train the model. Hence, in this way TrAdaBoost allows knowledge to be transferred from the old classifier to the new one.

A major challenge with transfer learning is **negative transfer**, which occurs when the new classifier performs worse than if there had been no transfer at all. Work on how to avoid negative transfer is an area of active research, where the key is to quantify the difference between the source task and the target task. *Heterogeneous transfer learning*, which involves transferring knowledge from different feature spaces and multiple source domains, is another active research topic. Traditionally, transfer learning has been used on small-scale applications. The use of transfer learning on larger applications, such as social network analysis and video classification, is often built upon the deep learning models with a 'pre-training' plus 'fine-tuning' strategy, which will be introduced in Chapter 11.

Transfer learning is closely related to another powerful weakly supervised learning method, namely *multi-task learning*.[9] Let us use the sentiment classification example to illustrate the difference between transfer learning and multi-task learning. In the transfer learning setting, we assume that we have a large number of manually labeled camera review data (i.e., the source task), but no or a very limited number of manually labeled TV review data (i.e., the target task). The goal of transfer learning to transfer the knowledge about the source task (camera review sentiment classification) to help build a better classifier for TV review sentiment classification (i.e., the target task). Now, suppose for both TV review and camera review, we only have a small amount of manually labeled data. How can we accurately build both classifiers—one for TV review sentiment and the other for camera review sentiment? Multi-task learning addresses this challenge by training both classifiers simultaneously so that the knowledge from one learning task (e.g., TV review sentiment) can be transferred to the other learning task (e.g., camera review sentiment), and vice versa.

8.5.4 Distant Supervision

Let us take another look at the sentiment classification example. Suppose that *AllElectronics* launches a new holiday sales campaign on social media platforms (e.g., Twitter), which goes viral with hundreds of thousands tweets. The manager at *AllElectronics* wants to figure out the sentiment of these Tweets, so that she can adjust the campaign strategy accordingly. We could manually label a large number of tweets regarding their sentiment and then train a classifier to predict the sentiment (positive vs. negative) of the remaining tweets. But that

[9]In some machine learning literature, multi-task learning is viewed as a special case of transfer learning, namely inductive transfer learning where the source and target domains share the same feature (i.e., attribute) space.

would be time consuming. The manager wonders: "*Can we train a sentiment classifier about the tweets without any manual labels?*" **Distant supervision** aims to answer this question by automatically generate a large number of labeled tuples. In particular, the manager notices that for a large subset of the tweets, its text content contains a ':)' sign or a ':(' sign, which are often associated with positive and negative sentiments respectively. Therefore, we could treat all the tweets with a ':)' sign as positive tuples and those with a ':(' sign negative tuples, and use them to train a sentiment classifier. Once the classifier is trained, we can use it to predict the sentiment for any future tweet even if it does not contain a ':)' or ':(' sign. Notice that in this case, we do not need to manually label *any* tweet in terms of its sentiment, and such labels (regarding positive or negative sentiment) are automatically generated.

In the tweet sentiment classification example above, we exploit the specific information (i.e., a ':)' or ':(' sign) in the input data to automatically generate labeled training tuples. An alternative strategy for classification with distant supervision often leverages the external knowledge base to automatically generate labels for the training tuples. For example, in order to classify tweets into different categories (e.g., news, health, science, games, etc.), we could explore the Open Directory Project (ODP, `http://odp.org`), which maintain a directory for web links by volunteers. Thus, if a tweet contains a url (e.g., `nytimes.com`), we can automatically find its ODP category (e.g., news), which is treated as the label of the corresponding tweet. In this way, we will be able to automatically generate a large labeled training set. Once the classifier is trained, we can use it to predict the class label (i.e., the category) of a test tweet, even if it does not contain a url. Another way to automatically generate labeled training examples is to leverage YouTube video that is linked to the tweet. The method is based on the following two observations. First, there are a large number of tweets, each of which contains a link to a YouTube video. Second, for each YouTube video, it is always associated with one of 18 predefined class labels. Therefore, we can treat the label of YouTube video as the label of the associated tweet.

In addition to social media post classification tasks, distant supervision is also found useful for relation extraction for natural language processing. An active research direction in distant supervision is how to effectively ask users to write a *labeling function*, instead of manually label training tuples, to automatically generate labels for a large number of unlabeled data. A major limitation of distant supervision is that the automatically generated labels are often very noisy. For example, some tweets with a ':)' sign could have neutral or even negative sentiment; the class labels of a tweet does not always align with the label or category of the url (either a web page or a YouTube video) it contains.

8.5.5 Zero-Shot Learning

Suppose that we have a collection of animal images, each of which has a unique label, including 'owl', 'dog', or 'fish'. Using this training data set, we can build a

classifier, say SVMs or logistic regression classifier.[10] Then, given a test image, we can use the trained classifier to predict its class label, i.e., which one of the three possible animals (owl, dog or fish) this image is about. *But, what if the test image is actually about a cat?* In other words, the class label of the test data *never* appears in the training data. This is what *zero-shot learning* aims to address, where the classifier needs to predict a test tuple whose class label was never observed during the training stage. In other words, there is *zero* training tuples for the novel class label (e.g., cat in our example). The term 'shot' here refers to data tuple.

At the first glance, this seems to be an impossible mission. You might wonder: '*If there is zero training tuples about the cat, how can I build a classifier to recognize an image about the cat?'* But, we might have some high-level description about the novel classes. For example, for 'cat', we can learn from the Wikipedia that a cat has retractable claws and super night vision. Zero-shot learning tries to leverage such external knowledge or side-information to build a classifier that can recognize such novel class labels.

Let us use the animal classification example (Figure 8.19) to explain how zero-shot learning works. Formally, there are n training images each of which is represented by a d-dimensional feature vector and a 3-dimensional label vector. The label vector indicate which of the three known classes the training image belongs to. For example, for an image about a 'dog', its label vector is $[1, 0, 0]$. In addition, we have the external knowledge about the class label, where each class label (animal) can be described by four *semantic attributes*,[11] including whether the animal 'has four legs', 'has wings', 'has retractable claws, and 'has super night vision'. For example, since a dog has four legs, but no wings or retractable claws or super night vision, the class label 'dog' can be described by a 4-dimensional semantic attribute vector $[1, 0, 0, 0]$. Likewise, the class label 'cat' can be described by a 4-dimensional semantic attribute vector $[1, 0, 1, 1]$, meaning that a cat has four legs, retractable claws and super night vision but no wings. Notice that such external knowledge is available for both known class labels (e.g., 'owl', 'dog' and 'fish') and novel class labels (e.g., 'cat' and 'rooster').

Then, using the input training tuples (i.e., the $n \times d$ feature matrix X and the $n \times 3$ label matrix Y in the upper left corner of Figure 8.19) and the external knowledge about the three known class labels (i.e, the information about four semantic attributes for the three known class labels in the bottom left corner of Figure 8.19), we train a *semantic attribute classifier* $\mathcal{F}$, which predicts a 4-dimensional semantic attribute vector for an input image represented by a d-dimensional feature vector. In our example, the output of the semantic attribute classifier $\mathcal{F}$ tells whether the given image has 'four legs', 'wings',

[10] Different from the classification tasks we have seen so far which typically involve two possible class labels (e.g., positive vs. negative sentiment), in this setting, we have a multiclass classification problem since there are three possible class labels. The techniques for multi-class classification will be introduced in Section 8.7.1.

[11] In literature, the semantic attribute is also referred to as semantic feature or semantic property or just attribute.

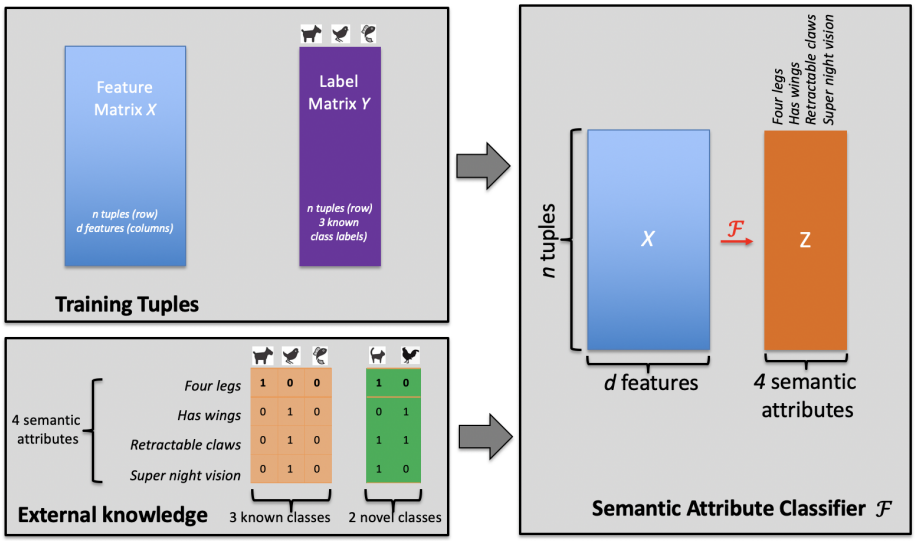


Figure 8.19: Top left: input n training tuples in d-dimensional feature space, each of which is labeled by one of the three known classes (i.e., 'dog', 'owl', and 'fish'. Bottom left: external knowledge where each known and novel class is described by four semantic attributes. Right: the trained semantic attribute classifier $\mathcal{F}$.

'retractable claws' and 'super night vision' respectively. We can use a 2-layer neural network to train such a semantic attribute classifier which will be introduced in Chapter 11.[12] Then, given a test image, we predict which of the two novel classes (i.e., 'cat' and 'rooster') it belongs to based on the following two steps. First, given the d-dimensional feature vector of the test image, we use the semantic attribute classifier $\mathcal{F}$ to output a 4-dimensional semantic attribute vector, whose elements indicate whether or not the test image has the corresponding semantic attributes. For example, if the semantic attribute classifier output a vector $[1, 0, 0, 1]$, it means that the classifier predicts that the test image (1) has four legs, (2) has no wings, (3) has no retractable claws, and (4) has super night vision. Second, we compare the predicted semantic attribute vector with the external knowledge about the two novel classes respectively (i.e., the 4×2 green table in the middle bottom of Figure 8.19). We predict that the test image belongs to the novel class whose semantic attribute vector is most similar to that of the test image. In our example, since the predicted semantic attribute vector $[1, 0, 0, 1]$ is more similar to that of 'cat' ($[1, 0, 1, 1]$) than that of 'rooster' ($[0, 1, 1, 0]$), we predict that it is an image about 'cat'.

[12] The input of this 2-layer neural network is the d-dimensional feature, the hidden layer corresponds to the 4 semantic attributes, and output layer corresponds to the 3 known class labels. Unlike a typical neural network, the model parameter for the second layer (from semantic attribute to the known class labels) can be directly obtained based on the external knowledge about 4 semantic attributes for the 3 known class labels.

The key of the method described above is that we leverage the semantic attributes as a bridge to transfer the output of the semantic classifier which was trained on the known class labels to predict the novel class labels. From this perspective, we can also view zero-shot learning as a special form of transfer learning (i.e., to transfer the knowledge about the known class labels to novel classes). In addition to the semantic attribute, there are other forms of external knowledge that can be harnessed for zero-shot learning. An example is the class-class similarity between known and novel classes. In the animal image classification application mentioned above, we can train a multi-class classifier to predict which of the three known classes an image belongs to. Now, given a test image that comes from the novel class (either 'cat' or 'rooter'), the trained classifier predicts it belongs to 'dog', and if we know that 'dog' is more similar to 'can' than 'rooster', it is safe to predict the test image is indeed a 'cat', rather than a 'rooster'. In the standard zero-shot learning setting, we always assume that the test image must come from one of the novel classes. This assumption might be too strong in reality. For example, the test image might come from either known classes (dog, owl or fish) or novel classes (cat or rooster). There have been research on *generalized zero-shot learning* to address such a more complicated setting. Other applications of zero-shot learning include neural activity recognition, where the classifier needs to recognize the word that a person is thinking about based her neural activity reflected on the fMRI image. In this application, the class labels are words. It is impossible to construct a training data set that covers all possible words that a human can think of. Zero-shot learning can effectively help extrapolate the classifier trained on a limited number of words (known class labels) to the unseen words during the training stage (i.e., the novel classes).

8.6 Classification with Rich Data Type

The classification techniques we have seen so far assume the following setting. That is, given a training set, where each training tuple is represented by a feature (or attribute) vector and a class label, we build a classifier that predicts the label of a test (unseen) tuple. Since each training tuple is represented as a feature vector, it can be viewed as a data point in the feature space (i.e., *spatial* data). Beyond spatial data, there are rich types of data from real-world applications, such as stream data, sequence (e.g., text), graph data, grid data (e.g., image), and spatial-temporal data (e.g., video). Many classification techniques have been developed for various types of data. In this section, we will see three case studies, including stream data classification (Section 8.6.1), sequence classification (Section 8.6.2) and graph data classification (Section 8.6.3). Deep learning techniques that will be introduced in Chapter 11 provide another powerful way for classification with rich data types, by automatically learning a feature representation of the input data (e.g., image, text, graph).

8.6.1 Stream Data Classification

Suppose a bank wants to develop a data mining tool to automatically detect fraudulent transactions. To start with, we could collect a large set of historical transactions which contain both legitimate and fraudulent transactions, and use them as the training tuples to construct a classifier (e.g., SVMs, logistic regression, etc.). If the performance of the trained classifier is acceptable, we integrate it into the bank's IT system to classify future transactions as fraudulent vs. legitimate. When the classifier flags a fraudulent transaction, we will ask a bank expert to manually check if it is indeed a fraudulent transaction or a false positive (i.e., the transaction is actually a legitimate one). Once we receive such manual annotation from the bank expert, naturally, we might want to use it as the new training tuples to improve the classification accuracy. This is a new classification setting, namely *stream data classification*, where the transactions arrive sequentially at different times in a stream fashion,[13] and the classifier needs to (1) classify the new transactions upon their arrival, and (2) be updated (i.e., re-trained) with the newly available labeled tuples (e.g., the new fraudulent transactions confirmed by the bank expert).

There are a number of challenges facing stream data classification. First, the new data tuples often arrive in a high speed. Therefore, simply using the newly obtained labeled tuples (e.g., newly found fraudulent transactions by the bank expert) together with the historical training tuples to re-train the underlying classifier might not be affordable, since the speed to re-train the classifier from scratch might be slower that the arrival rate of the new data tuples. Second, the length of the stream data (i.e., the total number of the transactions) is often very large, and in theory could be *infinite*. For example, a bank could constantly receive new transactions over many many years. Therefore, it is impossible to store all the data tuples, and as such in stream data classification, it is often assumed that each data tuple can only be accessed once or a few limited number of times. This is often referred to as *one-pass constraint*. Third, the characteristic of the underlying classes might change over time. For example, some fraudulent users might change their behaviors in order to bypass the current fraud detection tool. As such, some historical training tuples might become less relevant or even noisy for building an effective classifier to detect new fraudulent transactions. This is often referred to as *concept drifting*, that is, the concept (i.e., the class label that the classifier aims to learn) is evolving over time.

An effective method for stream data classification is based on ensemble, which works as follows (see Figure 8.20 for an illustration). We partition the data stream into equal-sized chunks. Each chunk contains a training set (X_i, y_i) $(i = 0, 1, ..., k)$, where X_i and y_i are the feature matrix and the label vector for the i^{th} chunk; and $i = 0, 1, ..., k$ is the index of the chunks, with $i = 0$ being the current (i.e., the most recent) chunk and $i = k$ being the oldest chunk. For each of the k historical chunks (i.e., $i = 1, ..., k$), we train a classifier f_i (e.g., Naive Bayes classifier), which outputs the posterior probability that the given tuple x belongs to the target class c (e.g., fraudulent transaction) $f_i(x) = p_i(c|x)$. Each

[13] Formally, a data stream is an ordered sequence of data tuples.

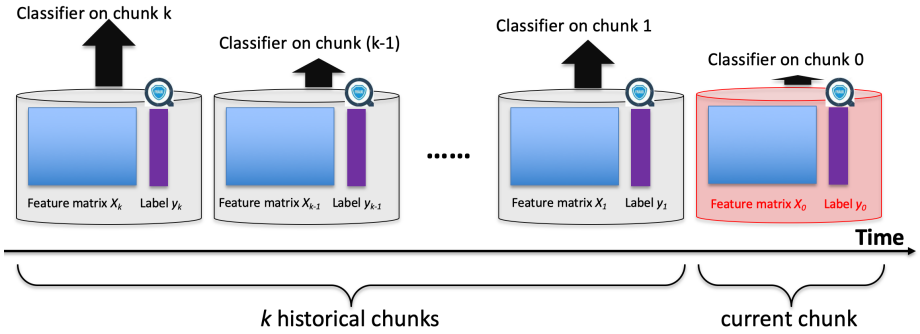


Figure 8.20: Ensemble based method for stream data classification. Each chunk trains an individual classifier, and uses the most recent chunk (chunk 0) to estimate the classification error. The higher the classification error, the lower the weight in the ensemble. In the figure, the height of the black arrow indicates the classification error. The individual classifier on chunk k will be discarded since it has the highest error rate. The individual classifier on chunk 1 has a lower weight in the ensemble, compared with chunk $(k-1)$ or chunk 0.

classifier f_i is also associated with a weight w_i. Therefore, the ensemble output of a test tuple x (i.e., the overall predicted probability that the given transaction x is fraudulent) is the weighted sum of the outputs of these k classifiers, i.e., $p(c|x) = \sum_{i=1}^{k} f_i(x)w_i$. When a new chunk (X_0, y_0) arrives, we train a new classifier f_0. In the meanwhile, we use the newly arrived chunk as the test set to estimate the classification error of all $(k+1)$ classifiers $f_i(i = 0, ..., k)$.[14] Among all $(k+1)$ classifiers, we select k classifiers with the lowest classification errors as the members of the ensemble. In other words, the classifier with the highest classification error rate will be discarded. In the meanwhile, for the k survived classifiers, we update their weights. The lower the classification error rate, the higher the weight. We can see that this method naturally addresses the three main challenges for stream data classification mentioned before. First, we only use the most recent chunk to train the new classifier f_0, which handles the high arrival rate of the stream data. Second, each incoming data tuple is accessed once to train the current classifier as well as to update the weights of individual classifiers. Third, by adjusting the weights of the individual classifiers, the ensemble pays more attention to the most relevant chunks (i.e., those individual classifiers with lower classification errors on the most recent chunk), so that it naturally captures the drifted class concept over time.

In addition to finance, stream data classification has also been applied to other application domains, such as marketing, network monitoring, and sensor networks. Many alternative learning strategies for stream data classification

[14]For the newly constructed classifier f_0, we cannot directly use the new chunk (X_0, y_0) as the test set, since it is also used to train the classifier f_0. Otherwise, it will leads to an over-optimistic error rate. Instead, we can use the cross-validation technique on chunk (X_0, y_0) to estimate the error rate of the classifier f_0.

exists. For example, the very fast decision tree (VFDT) is built upon (a) the so-called Hoeffding trees, which build the decision tree (e.g., splitting an attribute on tree nodes) by using a sampled subset of training tuples, (b) sliding window mechanism to obtain classifier that focuses on the most recent stream data. Another characteristics of stream data lies in its semi-supervised nature. This means that the vast majority of the newly arrived data are unlabeled and only a handful of them are labeled. For example, in the fraud transaction examples, the bank expert might only be able to manually label a very small percentage (say 1%) of them. There have been research that develops semi-supervised stream data classification based on ensemble methods.

8.6.2 Sequence Classification

A **sequence** (i.e. sequential data) is an *ordered* list of values $(\boldsymbol{x}^1, \boldsymbol{x}^2, ..., \boldsymbol{x}^T)$, where $\boldsymbol{x}^t$ $(t = 1, ..., T)$ is the value at a particular position or time stamp t, and T is the length of the sequence. The value $\boldsymbol{x}^t$ could be a categorical value (i.e., a symbol), or a numerical value or even a vector, or an itemset. Sequence naturally appears in many real applications. To name a few, in natural language processing, the sequence could be a sentence, where $\boldsymbol{x}^t$ is the t^{th} word of the input sentence and T is the length (number of words) of the sentence; in genomic analysis, a sequence could be a DNA segment, where each value $\boldsymbol{x}^t$ is one of the four animo acid A, C, G and T; in time series, the sequence could be a sequence of measurements at different time stamps, where $\boldsymbol{x}^t$ is one or more measurements (e.g., temperature, humidity) at the t^{th} time stamp and T is the total time stamps; in frequent pattern mining for market analysis, the sequence could be a list of transactions of a customer over time (i.e., each value $\boldsymbol{x}^t$ is the itemset the customer purchased at the corresponding time stamp t). The goal of sequence classification is to build a classifier that predicts the label of a given sequence. The label could be the positive vs. negative sentiment of the sentence in natural language processing, the gene coding area vs. non-coding area in genomic analysis, high-value vs. ordinary customer in market analysis.

One approach for sequence classification is through *feature engineering*. That is, we first convert the input sequence to a vector of features, which is in turn fed into a conventional classifier (e.g., decision tree, support vector machine). For symbolic sequence where each value $\boldsymbol{x}^t$ is a categorical value, we can use *n-gram* to construct the candidate features. Formally, an n-gram is a segment of sequence with n consecutive symbols. In our genomic analysis example, each 1-gram (also called unigram) is just one of the four animo acid (A, C, G, and T); and each 2-gram (also called bigram) is a sequence segment of two consecutive animo acid (e.g., AC, AG, GT, etc.). In natural language processing, each unigram could be a single word and a bigram is two consecutive words. For frequent sequence pattern mining, we can use pattern-based approach where each candidate feature is a frequent sequence pattern. Given a set of candidate features, a sequence can be converted to a feature vector, whose elements indicate the presence or absence of the corresponding candidate features in the input sequence. Alternatively, the elements of the feature vector could indicate

	Unigrams				Bigrams															
Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Candidate features	A	C	G	T	AC	AG	AT	CG	CT	CA	GA	GC	GT	TA	TC	TG	AA	CC	GG	TT
Feature vector (binary)	1	1	1	1	1	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0
Feature vector (frequency)	1	5	1	1	1	0	0	1	0	0	0	0	1	0	0	0	0	4	0	0

Figure 8.21: An example of using $n-$gram for sequence data feature engineering. Given a DNA segment 'ACCCCCGT', we want to convert it to a feature vector, using n-grams. There are 20 candidate features in total, including 4 unigrams and 16 bigrams. For the binary feature vector (the third row), an element indicates whether the corresponding feature appears (1) or not (0) in the input sequence. For the weighted feature vector (the fourth row), an element indicates how many times (the frequency) that the corresponding feature appears in the input sequence.

the frequency of the corresponding features (i.e., how many times the given candidate feature appears in the input sequence). See Figure 8.21 for an example. For numerical sequence, we can first use discretization to convert it into a symbolic sequence, which might cause information loss. One potential issue with the feature engineering based approach is that the number of candidate features might be large, many of which are irrelevant to the classification task. Hence, feature engineering often works hand-in-hand with feature selection for sequence classification. Feature selection was introduced in Section 8.1.

Some classifiers we have introduced before reply on certain distance measures (e.g., $k-$nearest neighbor classifier) or a kernel function (e.g., support vector machines) between different data tuples. Therefore, an alternative way for sequence classification is to define appropriate distance measures or a kernel function between different sequences. $k-$nearest neighbor classifier with commonly used distance measures (e.g., Euclidean distance) often leads to a competitive performance for sequence classification. There also exist distance measures and kernel functions that are specially designed for sequence data. For example, *dynamic time warping* distance (DTW) is a more robust distance measure than Euclidean distance with respect to the distortion in time; a commonly used kernel function for sequence data is called *string kernel*, which can be in turned fed into support vector machine for sequence classification.

In addition to predicting a class label for the entire sequence, some sequence classification task seeks to predict a label for each time stamp. For example, in natural language processing, we might want to predict whether each word of a given sentence is a specific type of named entity (e.g., location, person); in part-of-speech tagging, we need to predict whether each word is a pronoun or verb or noun. The key to sequence classification is to accurately model the *sequential dependence* among different values $\boldsymbol{x}^t(t = 1, ..., T)$. A traditional method for modeling sequential dependence is called *Hidden Markov Model* (or HMM for short), which has a fundamental limitation, in that it assumes the future values

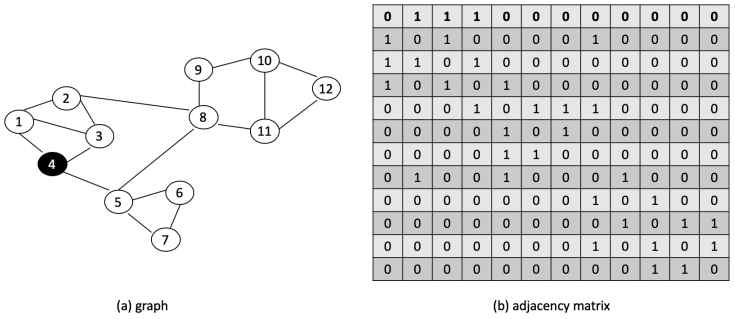
($\boldsymbol{x}^{t+1}$, $\boldsymbol{x}^{t+2}$, ...) are independent of the past values ($\boldsymbol{x}^1$, ..., $\boldsymbol{x}^{t-1}$) given the current value $\boldsymbol{x}^t$ (known as the Markov assumption). A more powerful method to handle the sequential dependence is a specific type of deep learning technique called *Recurrent Neural Networks* which will be introduced in Chapter 11.

8.6.3 Graph Data Classification

Graph data (also referred to as network data) which is essentially a collection of *nodes* (or vertices) linked with each other by *edges*, is a ubiquitous data type arising in many applications, including social networks, power-grid, transaction network, biological networks and more. The goal of graph data classification is to build a classifier to predict the label of either nodes (i.e., node level classification) or the entire graphs (i.e., graph level classification). An example of node level classification is webpage classification. Given a web graph whose nodes are webpages and edges are hyperlinks from one webpage to another, we want to determine the category (i.e., the label) of each webpage (i.e., a node in the webgraph). An example of graph-level classification is toxicity prediction. Given a collection of molecule graphs, we want to build a classifier to predict whether a given molecule graph is toxic (i.e., positive class label) or not (i.e., negative class label).

In the similar spirit as sequence classification, graph data classification can be done through feature engineering or proximity measures. For feature engineering based graph classification, we first extract a set of features describing each node or each graph, which are in turn fed into a conventional classifier (e.g., decision tree, logistic regression) to build a node-level or graph-level classifier. For node-level classification, commonly used node features include the number of neighboring nodes linked to the given node (i.e., node degree), the number of triangles the given node participates, the total edge weights of neighboring nodes (i.e., the weighted node degree), the node importance measure (e.g., eigen-centrality score, the PageRank score), the local clustering coefficient which measure to what extent the neighborhood of the given node looks like a full clique. For graph-level classification, commonly used graph features include the size of the graph (e.g., the number of nodes, edges, the total edge weights), the diameter of the graph, the total number of triangles in the graph, etc. More recent approaches often rely on a specific deep learning technique called *Graph Neural Networks* (which will be introduced in Chapter 11) to automatically extract node-level or graph-level features.

"How can we measure the proximity between two nodes or two graphs?" Let us first introduce the notation of *adjacency matrix*, which is an $n \times n$ matrix for a graph with n nodes. The rows and columns of the adjacency matrix $\boldsymbol{A}$ represent the nodes. Given two nodes i and j, if there is a connection between them, we set the corresponding entries of the adjacency matrix as 1 (i.e., $\boldsymbol{A}(i,j) = \boldsymbol{A}(j,i) = 1$); otherwise, we set $\boldsymbol{A}(i,j) = \boldsymbol{A}(j,i) = 0$. We can also set entry $\boldsymbol{A}(i,j)$ as a numerical value to indicate the weight of the corresponding edge. Figure 8.22 presents an example. An effective way to measure the node proximity is called **random walk with restart**. The algorithm is summarized in Figure 8.23 and



0	1	1	1	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	1	0	0	0	0
1	1	0	1	0	0	0	0	0	0	0	0
1	0	1	0	1	0	0	0	0	0	0	0
0	0	0	1	0	1	1	1	0	0	0	0
0	0	0	0	1	0	1	0	0	0	0	0
0	0	0	0	1	1	0	0	0	0	0	0
0	1	0	0	1	0	0	0	1	0	0	0
0	0	0	0	0	0	0	1	0	1	0	0
0	0	0	0	0	0	0	0	1	0	1	1
0	0	0	0	0	0	0	1	0	1	0	1
0	0	0	0	0	0	0	0	0	1	1	0

Figure 8.22: An example of a graph (a) and its adjacency matrix (b).

Algorithm: Random walk with restart for measuring node proximity.

Input:

- $\boldsymbol{A}$, adjacency matrix of the input graph of size $n \times n$;
- i, the query node;
- $0 < c < 1$, a damping factor.

Output: The node proximity vector $\boldsymbol{r}$ of size $n \times 1$.

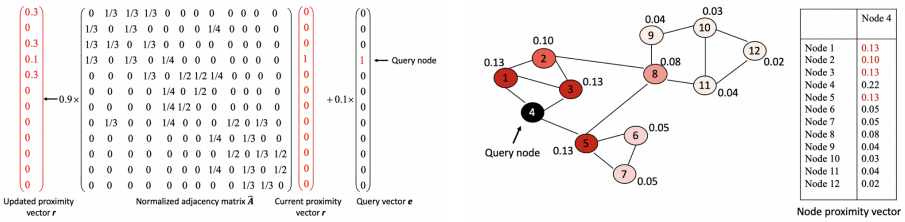
Method:

//Pre-processing
(1) Set the query vector $\boldsymbol{e}$ as an $n \times 1$ vector, where $\boldsymbol{e}(i) = 1$ and $\boldsymbol{e}(j) = 1$ $(j \neq i)$;
(2) Initialize proximity vector $\boldsymbol{r} = \boldsymbol{e}$;
(3) Calculate the degree matrix $\boldsymbol{D}$ of $\boldsymbol{A}$,
where $\boldsymbol{D}(i,i) = \sum_{j=1}^{n} \boldsymbol{A}(i,j), \boldsymbol{D}(i,j) = 0$ $(i \neq j, i, j = 1, .., n)$;
(4) Normalize $\hat{\boldsymbol{A}} = \boldsymbol{A}\boldsymbol{D}^{-1}$;
(5) **while** (termination condition is not satisfied){ // for each iteration
(6) Update $\boldsymbol{r} \leftarrow c\hat{\boldsymbol{A}}\boldsymbol{r} + (1-c)\boldsymbol{e}$;
(7) }

Figure 8.23: Random walk with restart for measuring the proximity between nodes on a graph.

the steps are described next.

Given a query node i, Figure 8.23 produces a node proximity vector $\boldsymbol{r}$ of length n, which contains the proximity measures from node i to other nodes in the graph. First (Step 1), we introduce a query vector $\boldsymbol{e}$ which is an $n \times 1$ vector, whose i^{th} entry is 1 (i.e., $\boldsymbol{e}(i) = 1$) and all other entries are zeros. We (Step 2) initialize the node proximity vector $\boldsymbol{r}$ as the query vector $\boldsymbol{e}$. Then (Steps 3-4), we normalize the adjacency matrix so that each column of $\hat{\boldsymbol{A}}$ sums



(a) First iteration of random walk with restart

(b) Final proximity scores and proximity vector

Figure 8.24: An example of applying random walk with restart to compute the proximity vector of query node 4. (a) The first iteration of the algorithm, where the damping factor $c = 0.9$. (b) The final proximity scores from node 4 to other nodes (more red mean higher proximity scores) and proximity vector.

up to 1.[15] After that, we iteratively update the node proximity vector $\boldsymbol{r}$ (Step 6) until a termination condition is met. At each iteration, we update the node proximity score for each node as follows. For the query node itself, its proximity score is updated as $\boldsymbol{r}(i) \leftarrow c\sum_{t=1}^{n} \hat{\boldsymbol{A}}(t,i)\boldsymbol{r}(t) + (1-c)$. That is, the updated proximity score for node i is a weighted sum of the proximity scores of its neighboring nodes, damped by the parameter c and then increased by an amount of $(1-c)$. For each other node j $(j \neq i)$, its proximity score is updated as $\boldsymbol{r}(j) \leftarrow c\sum_{t=1}^{n} \hat{\boldsymbol{A}}(t,j)\boldsymbol{r}(t)$. That is, the updated proximity score for node j is a weighted sum of the proximity scores of its neighboring nodes, damped by the parameter c. We repeat this process until a termination condition is met (e.g., a maximum iteration number is reached, or the difference between the node proximity vectors in two consecutive iterations is small enough).

Example 8.6.1 Let us apply random walk with restart algorithm in Figure 8.23 to the graph in Figure 8.22, and we aim to compute the proximity scores from the query node 4 to all other nodes. To this end, we set the query node $\boldsymbol{e}$ as a vector of length 12, since there are 12 nodes in total, and the 4^{th} entry of the query vector is 1 and all others are 0s. We initialize the proximity vector the same as the query vector $\boldsymbol{r} = \boldsymbol{e} = (0,0,0,1,0,0,0,0,0,0,0,0)'$, and normalize the adjacency matrix (Figure 8.24(a)). In order to update the node proximity vector, we iteratively apply Step 7 of Figure 8.23. Figure 8.24(a) demonstrates the computation results of the first iteration. The final proximity scores and proximity vector are shown in Figure 8.24(b). We can see that the results are more or less consistent with our intuition, in that, nodes that are closer to the query node 4 (e.g., nodes 1, 2, 3, 5) receive higher proximity scores than others.

"Why is random walk with restart a good node proximity measure?" Algorithm described in Figure 8.23 is equivalent to the following random walk

[15] Alternative normalization approaches exist. For example, for a directed graph, we can set $\hat{\boldsymbol{A}} = (\boldsymbol{D}^{-1}\boldsymbol{A})'$, where the prime denotes matrix transpose. We can also use a symmetric normalization $\hat{\boldsymbol{A}} = \boldsymbol{D}^{-1/2}\boldsymbol{A}\boldsymbol{D}^{-1/2}$.

process. At the beginning, there is a random particle at the query node (e.g., node 4 in our example). At each iteration, the random particle does one of the following two things. First, it randomly surfs the graph. That is, it will randomly jumps to one of its neighbors with the probability that is proportional to the edge weights between them. Second, it returns to the starting (i.e., query) node (hence the name 'with restart'). It turns out the proximity score computed from Figure 8.23 is equivalent to the *steady state probability* that the random particle will eventually stay at the corresponding node. Therefore, if a node is closer to the query node, it will have a higher chance to attract the random particle to eventually stay there, and hence receive a higher proximity score. Another reason that random walk with restart provides a good node proximity lies in its ability to summarize *multiple, weighted relationship* between nodes on graph. For the example in Figure 8.24, the proximity between node 4 and node 8 by random walk with restart summarizes all the paths between these two nodes, with higher weights to those shorter, heavily weighted paths. Therefore, if there exist multiple paths between two nodes and each path is short and heavily weighted, random walk with restart will assign a higher proximity score between.

Once we have a node proximity measure, we can apply it to *semi-supervised node classification.* Given a graph and some (often a very limited number) labeled nodes, we can predict the labels of the remaining nodes as follows. If the average node proximity between a test node and all positively labeled nodes is higher than the average node proximity between the test node and all negatively labeled nodes, we predict it a positive node. Otherwise, we predict it a negative node.

Many variants of random walk with restart exist. For example, some methods measure the node proximity based on *commute time* or *hitting time* from random walk perspective; some methods view the graph as an electric network and measure the node proximity based on *effective conductance.* Most of these methods share the similar idea as random walk with restart in that these methods all aim to summarize multiple, weighted relationship between nodes as the proximity measure. There also exist methods that generalize random walk with restart to attributed graphs, so that the node proximity considers both the topology of the graph and the attributes of nodes and edges. Random walk based approaches can also be generalized to measure the proximity between different graphs. For example, one method uses a similar mathematical formulation as random walk with restart, but defined over the *Kronecker graph* of the input graphs, to measure the similarity between two graphs. Such a similarity measure turns out to be a valid kernel function (i.e., random walk based graph kernel). Therefore, we can feed such proximity measures into support vector machines for graph level classification.

8.7 Potpourri: Other Related Techniques

Classification plays a very important role in data mining and has made tremendous progress in the past decades. As such, many techniques have been developed to address various aspects of classification. Classification is also closely related to other data mining tasks. In this section, we will learn some of these techniques. Most classification algorithms we have studied handle multiple classes, but some, such as support vector machines and logistic regression, typically assume only two classes exist in the data. What adaptations can be made to allow more than two classes? This question is addressed in Section 8.7.1 on *multiclass classification*. Some classifiers we have learned so far (e.g., k-NN) rely on a distance (or similarity) measure between different data tuples. Section 8.7.2 introduces how to automatically learn a good distance metric for the classification task. In addition to classification accuracy, an increasingly important aspect is the interpretability of classification. It is highly desirable that the trained classifier not only predicts the class label of a test tuple, but also helps the user understand why the classifier 'thinks' the test tuple should have the predicted label. Section 8.7.3 introduces techniques to render interpretability to classification. As we have seen, many classification problems can be formulated from the optimization perspective, some of which are not easy to solve due to their combinatorial, non-convex nature. Genetic algorithm is a powerful technique to handle combinatorial optimization problem, which will be introduced in Section 8.7.4. Finally, classification belongs to a specific type of supervised learning, where the classifier receives the *instructive feedback* (i.e., the true labels for the training tuples) in order to construct the best classifier. *Reinforcement learning* represents another type of supervised learning, where the learning agent receives *evaluative feedback* (e.g., the reward of an action taken at a specific time step instead of the true value of that action). Reinforcement learning is introduced in Section 8.7.5.

8.7.1 Multiclass Classification

Some classification algorithms, such as support vector machines and logistic regression, are typically designed for binary classification. How can we extend these algorithms to allow for **multiclass classification** (i.e., classification involving more than two classes)?

A simple approach is **one-versus-all** (OVA). Given m classes, we train m binary classifiers, one for each class. Classifier j is trained using tuples of class j as the positive class, and the remaining tuples as the negative class. It learns to return a positive value for class j and a negative value for the rest. To classify an unknown tuple, $\boldsymbol{X}$, the set of classifiers vote as an ensemble. For example, if classifier j predicts the positive class for $\boldsymbol{X}$, then class j gets one vote. If it predicts the negative class for $\boldsymbol{X}$, then each of the classes except j gets one vote. The class with the most votes is assigned to $\boldsymbol{X}$.

All-versus-all (AVA) is an alternative approach that learns a classifier for each pair of classes. Given m classes, we construct $\frac{m(m-1)}{2}$ binary classifiers.

Class	*Error-correcting codeword*
C_1	1 1 1 1 1 1 1
C_2	0 0 0 0 1 1 1
C_3	0 0 1 1 0 0 1
C_4	0 1 0 1 0 1 0

Figure 8.25: Error-correcting codes for a multiclass classification problem involving four classes.

A classifier is trained using tuples of the two classes it should discriminate. To classify an unknown tuple, each classifier votes. The tuple is assigned the class with the maximum number of votes. All-versus-all tends to be superior to one-versus-all.

A problem with the previous schemes is that binary classifiers are sensitive to errors. If any classifier makes an error, it can affect the vote count.

Error-correcting codes can be used to improve the accuracy of multiclass classification, not just in the previous situations, but for classification in general. Error-correcting codes were originally designed to correct errors during data transmission for communication tasks. For such tasks, the codes are used to add redundancy to the data being transmitted so that, even if some errors occur due to noise in the channel, the data can be correctly received at the other end. For multiclass classification, even if some of the individual binary classifiers make a prediction error for a given unknown tuple, we may still be able to correctly label the tuple.

An error-correcting code is assigned to each class, where each code is a bit vector. Figure 8.25 shows an example of 7-bit codewords assigned to classes C_1, C_2, C_3, and C_4. We train one classifier for each bit position.[16] Therefore, in our example we train seven classifiers. If a classifier makes an error, there is a better chance that we may still be able to predict the right class for a given unknown tuple because of the redundancy gained by having additional bits. The technique uses a distance measurement called the Hamming distance to guess the "closest" class in case of errors, and is illustrated in Example 8.7.1.

Example 8.7.1 **Multiclass classification with error-correcting codes.** Consider the 7-bit codewords associated with classes C_1 to C_4 in Figure 8.25. Suppose that, given an unknown tuple to label, the seven trained binary classifiers collectively output the codeword 0001010, which does not match a codeword for any of the four classes. A classification error has obviously occurred, but can we figure out what the classification most likely should be? We can try by using the **Hamming distance**, which is the number of different bits between two codewords. The Hamming distance between the output codeword and the codeword for C_1 is 5 because five bits—namely, the first, second, third, fifth, and seventh—differ. Similarly, the Hamming distance between the output code and the codewords for C_2 through C_4 are 3, 3, and 1, respectively. Note that

[16] Conceptually, we can think of each bit as a semantic attribute in the zero-shot learning setting which was introduced in Section 8.5.5.

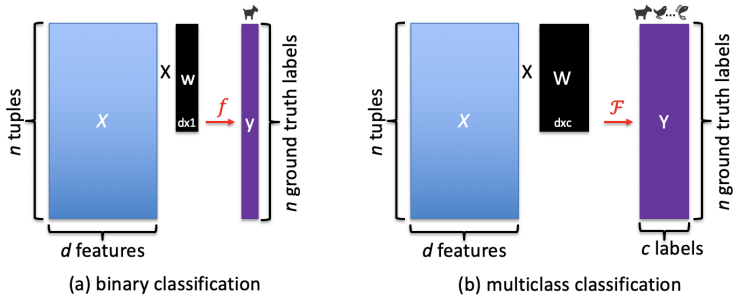


Figure 8.26: Comparison between binary vs. multiclass classification.

the output codeword is closest to the codeword for C_4. That is, the smallest Hamming distance between the output and a class codeword is for class C_4. Therefore, we assign C_4 as the class label of the given tuple.

Error-correcting codes can correct up to $\frac{h-1}{2}$ 1-bit errors, where h is the minimum Hamming distance between any two codewords. If we use one bit per class, such as for 4-bit codewords for classes C_1 through C_4, then this is equivalent to the one-versus-all approach, and the codes are not sufficient to self-correct. (Try it as an exercise.) When selecting error-correcting codes for multiclass classification, there must be good row-wise and column-wise separation between the codewords. The greater the distance, the more likely that errors will be corrected.

Binary vs. Multiclass classification. Let us use the example in Figure 8.26 to explain the relationship between binary classification and multiclass classification. For clarity, we use linear classifiers, such as logistic regression, linear SVMs. Suppose that we are given an $n \times d$ feature matrix X which represents n training images in d-dimensional feature space. For the binary classification setting (Figure 8.26(a)), we want to predict if a given image is a 'dog' or not. Therefore, the class label for each training image can be represented as a binary scalar (e.g., 1 means the image is a dog, and -1 means it is not). For the class labels for all n training tuples, we have an $n \times 1$ label vector y. In order to train a linear classifier, we seek an optimal d-dimensional weight vector w which minimizes the following loss function, $w = \text{argmin}_w \mathcal{L}(Xw, y) + \lambda\Omega(w)$, where $\mathcal{L}$ is a loss function that depends on the specific classifier,[17] $\lambda > 0$ is a regularization parameter and $\Omega()$ is a regularization term of the weight vector w. A default choice for $\Omega(w)$ could be the squared l_2 norm of the weight vector w. Now, for the multiclass classification setting (Figure 8.26(b)), we want to predict which of c classes (e.g., 'dog', 'owl', 'fish', etc.) a given image belongs to. Therefore, the class label for each training image can be represented as a c-dimensional label vector (e.g., $[-1, 1, ..., -1]$ means that the image is an owl). For the class labels for all n training tuples, we have an $n \times c$ label matrix Y. In order to train a

[17] For example, $\mathcal{L}$ is the hinge loss function for linear SVMs and it is negative log likelihood function for logistic regression classifier.

linear classifier, we seek an optimal $d \times c$ weight matrix W (instead of a vector) which minimizes the following loss function, $W = \text{argmin}_W \mathcal{L}(XW, Y) + \lambda\Omega(W)$, where the loss function $\mathcal{L}$, the regularization parameter λ and the regularization term $\Omega()$ have similar means **yu: meanings?** as in the binary setting. Notice that in the multiclass setting, the classifier is expressed in the form of a $d \times c$ weight matrix W. The predicted class label $\tilde{y}$ for a test tuple $\tilde{x}$ can be set as $\tilde{y} = \text{argmax}_i \tilde{x} \cdot W_i$, where $\cdot$ is the dot product between two vectors, W_i is the i^{th} column of W and its elements measure the weights of the corresponding features for class label i. A default choice for the regularization term $\Omega()$ could be the squared Frobenius norm of the weight matrix W.

Muilticlass classification is closely related to **multi-label classification** problem, where each data tuple could belong to one or more classes. For example, in document classification, each document can have multiple labels, each corresponding to a specific category or tag the document belongs to. Let L be the total number of classes. A natural way to handle multi-label classification is to train L independent binary classifiers, one for each class label. That is, the l^{th} classifier predicts whether or not the given data tuple has the class label l ($l = 1, ..., L$). Note that this method bears subtle difference from the one-versus-all or all-versus-all methods for multiclass classification problem introduced before. For the former (multi-label classification), a given data tuple can belong to more than one class since L classifiers are independently trained and applied. For the latter (multiclass classification), we always assign a single label (out of L possible labels) to a data tuple, by voting. Another method for multi-label classification with L labels is to convert it to a multiclass classification problem with $(2^L - 1)$ labels. This process is called *label powerset transformation*. For example, for a multi-label classification problem with three possible labels, namely A, B and C, we can convert it to a multiclass classification problem with the following 7 possible labels, namely A, B, C, AB, AC, BC and ABC. In other words, each of the 7 newly constructed labels corresponds to a subset of the original three labels that is assigned to a data tuple.

In Chapter 11, we will introduce deep learning techniques, which can naturally handle multiclass classification problem by introducing one node for each class label in the output layer.

8.7.2 Distance Metric Learning

Some classifiers (e.g., k-nearest-neighbor classifiers) rely on a distance measure. In Section 7.4, we have learned that even with the same training tuples and the same choice of k, different distance metrics (e.g., L_1 vs. L_2) might lead to quite different decision boundaries. *So, is there a way that we can automatically learn the best distance metrics for a given classification task?* Distance metric learning (or metric learning) aims to answer this question.

A commonly used distance metric is Euclidean distance (also referred to as L_2 distance), which is often the default choice for many classifiers. Given two data tuples in $p-$dimensional space $X_1 = (X_{1,1}, X_{1,2}, ..., X_{1,p})'$ and $X_2 = (X_{2,1}, X_{2,2}, ..., X_{2,p})'$, the Euclidean distance between them is defined as

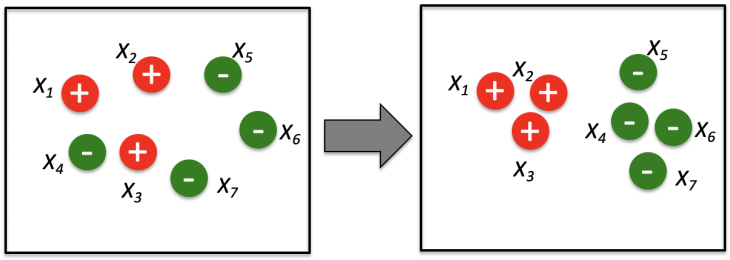
$d(X_1, X_2) = \sqrt{\sum_{i=1}^{p}(X_{1,i} - X_{2,i})^2} = \sqrt{(X_1 - X_2)'(X_1 - X_2)}$, where $'$ denotes the transpose of a vector. We compare the difference $(X_{1,i} - X_{2,i})$ between the two input data tuples along each of the p dimensions, sum the squared difference over all the p dimensions and take the square root of such a summation as the Euclidean distance. In other words, the different dimensions have the *equal* weights on the overall distance between two data tuples and their effects are considered *independently*. Therefore, if certain dimension has a larger value range than others, or if different dimensions are correlated with each other, Euclidean distance might lead to sub-optimal classification performance. To overcome the limitations of Euclidean distance, a more flexible and powerful distance is called Mahalanobis distance, which is defined as follows

$$d_M(X_1, X_2) = \sqrt{(X_1 - X_2)'M(X_1 - X_2)} = \sqrt{\sum_{i,j=1}^{p}(X_{1,i} - X_{2,i})M(i,j)(X_{1,j} - X_{2,j})} \tag{8.24}$$

where M is a $p \times p$ symmetric positive semi-definite matrix.[18] Compared with Euclidean distance, Mahalanobis distance naturally (1) assigns different weights (through the diagonal elements of matrix M) to different dimensions, and (2) incorporates the interaction effect of different dimensions (through the off-diagonal elements of matrix M) in measuring the distance between the two input tuples.

Depending on the specific choice of the M matrix, the Mahalanobis distance between two data tuples will vary. Therefore, the goal of distance metric learning becomes learning the optimal M matrix for a given classification task. Suppose there are n labeled training tuples in p-dimensional space. Let $\mathcal{S}$ contain all the training *tuple pairs* which are similar with each other (e.g., each member in $\mathcal{S}$ is a pair of training tuples which share the same class label). Let $\mathcal{D}$ contain all the training *tuple pairs* which are dissimilar with each other (e.g., each member in $\mathcal{D}$ is a pair of training tuples with different class labels). The basic idea of distance metric learning is that we want to find the optimal M matrix (hence the optimal Mahalanobis distance) so that (1) the distance between any pair of *similar* tuples in $\mathcal{S}$ is small and (2) the distance between any pair of *dissimilar* tuples in $\mathcal{D}$ is large (see Figure 8.27 for an illustration). Mathematically, we can formulate it as the following optimization problem.

[18] Mathematically, a $p \times p$ matrix M is positive semi-definite if $X'MX \geq 0$ for any $X \in R^p$. This is to ensure that $d_M(X_1, X_2)$ defined in Eq (8.24) is a valid distance metric. Since M is symmetric and positive semi-definite, we can decompose M as $M = L'L$, where L is a $r \times p$ matrix. In this way, we can re-write the Mahalanobis distance as $d_M(X_1, X_2) = \sqrt{(LX_1 - LX_2)'(LX_1 - LX_2)}$. Therefore, the Mahalanobis distance can be interpreted as the following process. That is, we first perform a linear feature transformation of the input tuples through the $r \times p$ matrix L (i.e., $X_j \rightarrow LX_j$ $(j = 1, 2)$); and then, we calculate the Euclidean distance between the two transformed data tuples in the r-dimensional space.



(a) Euclidean distance (b) Learnt Mahalanobis distance

Figure 8.27: An illustrative example of distance metric learning. Given 3 positive training tuples (X_1, X_2, X_3) and 4 negative training tuples (X_4, X_5, X_6, X_7). Left: Euclidean distance. Right: Learnt Mahalanobis distance, where the distance between tuple pairs of the same class label is smaller than that of different class labels. **yu: Can Mahalanobis distance change the linear separability of data points? Seems this is a linear transformation with the matrix L?** $\mathcal{S} = \{(X_1, X_2), (X_1, X_3), (X_3, X_2), (X_4, X_5), (X_4, X_6), (X_4, X_7), (X_5, X_6), (X_5, X_7), (X_6, X_7)\}$, and $\mathcal{D} = \{(X_1, X_4), (X_1, X_5), (X_1, X_6), (X_1, X_7), (X_2, X_4), (X_2, X_5), (X_2, X_6), (X_2, X_7), (X_3, X_4), (X_3, X_5), (X_3, X_6), (X_3, X_7)\}$.

$$\max_{M \in R^{p \times p}} \sum_{(X_i, X_j) \in \mathcal{D}} d_M^2(X_i, X_j) \tag{8.25}$$
$$\text{s.t.} \sum_{(X_i, X_j) \in \mathcal{S}} d_M^2(X_i, X_j) \leq 1, \quad M \succeq 0$$

$M \succeq 0$ means that matrix M is positive semi-definite. The optimization problem defined in Eq. (8.25) is convex. Therefore, we can resort to the off-the-shelf optimization software to solve it. We will not going into such details, which are out of the scope of this book.

The objective function as well as constraints in Eq. (8.25) are in the form of a pair of training tuples. In fact, the above formulation can be viewed as a binary classification problem, where the input is *a pair of training tuples* whose class label is $+1$ if the pair comes from $\mathcal{S}$ (i.e., they share the same class label), and -1 if the pair comes from $\mathcal{D}$ (i.e., they have different class labels). Alternative formulations for distance metric learning exist. For example, in the ranking task, the supervision might be in the form that certain tuples should be ranked higher than other tuples for a given query. The distance metric learning with such kind of supervision can be formulated with respect to the *triples* of training tuples (e.g., the query tuple, a high-ranked tuple and a low-ranked tuple). In Eq. (8.25), we work with the squared Mahalanobis distance, which makes the objective function as well as its constraints be linear with respect to matrix M. This type of methods is often referred to as linear distance metric learning. Nonlinear distance metric learning methods exist. For example, we

can *kernelizing* a linear distance metric learning method in the similar way as how we make a linear SVM classifier be a nonlinear in Section 8.3. Alternatively, some nonlinear distance metric methods learn *multiple* local linear metrics (e.g., one linear Mahalanobis distance for tuples in a given cluster). In terms of computation of Eq. (8.25), a major bottleneck lies in the constraint that the matrix M must be positive semi-definite (i.e., $M \succeq 0$). If we drop such a constraint, Eq. (8.25) becomes *similarity learning* problem, which can often be solved faster than distance metric learning. Beyond the standard classification, distance metric learning has also been applied to classification with weak supervision (e.g., semi-supervised learning, transfer learning) as well as other data mining tasks (e.g., ranking, clustering).

8.7.3 Interpretability of Classification

So far, we have primarily focused on the accuracy of the classification models. Indeed, the filed of data mining, machine learning and AI has witnessed tremendous progress on improving classification accuracy. For some application domains (e.g., computer visions, natural language processing), sophisticated classification models (such as deep learning techniques that will be introduced in Chapter 11) can now achieve an accuracy that is comparable or even surpasses humans on a variety of classification tasks. That said, the accuracy alone is often not sufficient for many application scenarios. For examples, how can the end user (e.g., the store manager in *AllElectronics*) understand the classification results by an SVM classifier? If a newly developed classification model improve the sentiment classification accuracy by 10% over the existing classifier, can we *trust* the results, that is, is the 10% improvement due to the new model's capability to capture the hidden feature in relation to the sentiment that was ignored by the existing classifier, or it is just due to the random noise? The answers to such questions lie in the *interpretability*, which describes the models' ability to explain the classification results or process in a user understandable way.

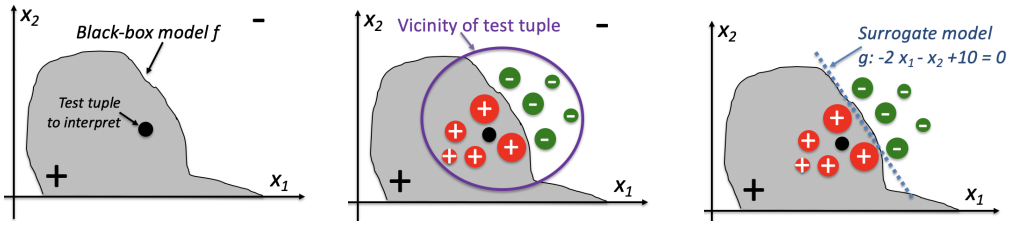
Interpretability naturally comes with some of the classification models we have learned so far. To name a few, for a decision tree classifier, the path from the root to the leaf node that the test tuple belongs to provide interpretation in terms of why the classifier predicts certain class label for the given test tuple. For the example in Chapter 7.1 (Table 7.1 and Figure 7.5), such an interpretation could be that "since the individual is *young*, with *medium income* and an *excellent credit rating*, the classifier predicts she will purchase a computer". Likewise, the rule antecedent in rule-based classification models could provide similar interpretation on why the classifier predicts a certain class label for a tuple. However, when the decision tree becomes deeper or the rule antecedent becomes longer, this type of interpretation becomes less effective. For linear classifiers (e.g., perceptron, logistic regression), the decision boundary of the classifier is in the form of $\sum_{i=1}^{p} w_i x_i = 0$, where w_i is the weight of the i^{th} attribute. Therefore, both the magnitude and the sign of the weight w_i provides an interpretation on the impact or contribution of the correspond-

ing attribute in making the class prediction. For high-dimensional data with a large number of attributes, it often leads to more effective interpretation if the linear classifier is combined with the feature selection (e.g., LASSO introduced in Section 8.1), so that we can focus on a few most important attributes to interpret the classification results. However, we cannot directly use this strategy to interpret non-linear classification models, such as Bayesian belief networks, nonlinear SVMs, deep neural networks.

The interpretation methods mentioned above (e.g., the path in a decision tree, the rule antecedent, the weights in linear classifiers) are *model-specific*, in the sense that the interpretation is designed for a specific classification model and we have the full access to the details of that model. But in some cases, the end user might only have the access to a black-box classification model f, which predicts a class label y for a given test tuple x. But the details (e.g., what kind of classification model, or its parameters) are unknown to the end user. How can we provide *model-agnostic* interpretation for such a black-box model? An effective way is to use a *proxy* or *surrogate* model g, which itself is interpretable (such as a shallow decision tree or a sparse linear classifier), to approximate the black-box classification model classification f in the local vicinity of a given test tuple which we wish to interpret.[19] LIME (Local Interpretable Model-agnostic Explanation) is such a model-agnostic method. Let us introduce its key idea using the example in Figure 8.28. In Figure 8.28(a), there is a black-box binary classification model f with a complicated decision boundary, which classifies the shaded area as positive labels and the remaining as the negative labels. We have a test tuple (the black dot) to interpret, that is, to help the end-user understand why the black-box model f predicts its class label as positive. To this end, LIME samples a few data tuples in the local vicinity of the test tuple (i.e., the purple circle in Figure 8.28(b)). For each sampled data tuple, it is assigned a binary class label, which is the prediction of the black-box model f. Each sampled tuple is also assigned a weight, which is in reverse proportion to its distance to the test tuple, the closer to the test tuple, the higher the weight (indicated by the size of sampled tuples in Figure 8.28(b)). Then, LIME trains a surrogate model g using the weighted sample tuples (e.g., a linear classifier in Figure 8.28(c)). LIME uses the surrogate model g to interpret the classification of the test tuple by the black-box model f. In this example, the test tuple is classified by the black-box model as a positive tuple and the decision boundary of the surrogate model g is $-2x_1 - x_2 + 10 = 0$. Therefore, we have the following interpretation: "the tuple is classified as positive by the black-box model f. This is because in its local vicinity, (1) both attributes (x_1 and x_2) have a negative impact on the positive class label (the larger x_1 and x_2, the less likely the test tuple belongs to the positive class; and (2) the effect of the first attribute x_1 is twice that of the second attribute x_1."

When we train the local surrogate model g, we use the *predicted label* by the

[19] There exist methods to use surrogate models to approximate f in the entire feature space (i.e., regardless of which test tuple we wish to interpret). However, these methods are less common, since it is very hard to find an interpretable surrogate model which can approximate the black-box classification model *globally*.



(a) black-box model f (b) local weighted sampling (c) training surrogate model

Figure 8.28: An example of LIME: Local Interpretable Model-agnostic Explanation

black-box model f as the labels of the sampled tuples. This is because we want to ensure the *local fidelity* of the surrogate model, which measures how well the surrogate model g approximates the black-box model in the local vicinity of the test tuple x. In the meanwhile, we want to keep the surrogate model g to be 'simple' so that it is interpretable to the end users. To this end, the surrogate model g often uses a set of interpretable features (e.g., the actual words in text classification, the region or super-pixel in image classification) which are different from the features used by the black-box model. The key in LIME is to strike a good trade-off between the *local fidelity* and the *model complexity*. The model complexity (e.g., the depth of the decision tree model, the number of selected features in the sparse linear classifiers) is in reverse proportion to the interpretability of the surrogate model g: a more complex model is less interpretable.

Alternative approaches to interpret the classification models exist. For example, *counterfactual explanation* interprets the prediction on a test tuple by identifying the optimal changes to its attributes (e.g., deleting certain words for text classification, changing the super-pixel for image classification, perturbing the continuous attribute values), which would alter the prediction results of the classification model. Another powerful interpretation technique is to use *influence function*, which is originated from the robust statistics. By influence function analysis, one could identify the most *influential* or important training tuples, which, if perturbed, would significantly alter the classification model (e.g., the weight vector in a linear classifier).

Beyond classification, interpretability also plays an important role in other data mining tasks, including clustering, outlier detection, ranking, recommendation. In addition to making the data mining models transparent so as to gain the users' trust of the model, interpretability is also intimately related to other important aspects of data mining, such as fairness, robustness, causality, privacy.

8.7.4 Genetic Algorithms

Genetic algorithms attempt to incorporate ideas of natural evolution. In general, genetic learning starts as follows. An initial **population** is created

consisting of randomly generated rules. Each rule can be represented by a string of bits. As a simple example, suppose that samples in a given training set are described by two Boolean attributes, A_1 and A_2, and that there are two classes, C_1 and C_2. The rule "*IF* A_1 *AND NOT* A_2 *THEN* C_2" can be encoded as the bit string "100", where the two leftmost bits represent attributes A_1 and A_2, respectively, and the rightmost bit represents the class. Similarly, the rule "*IF NOT* A_1 *AND NOT* A_2 *THEN* C_1" can be encoded as "001". If an attribute has k values, where $k > 2$, then k bits may be used to encode the attribute's values. Classes can be encoded in a similar fashion.

Based on the notion of survival of the fittest, a new population is formed to consist of the *fittest* rules in the current population, as well as *offspring* of these rules. Typically, the **fitness** of a rule is assessed by its classification accuracy on a set of training samples.

Offspring are created by applying genetic operators such as crossover and mutation. In **crossover**, substrings from pairs of rules are swapped to form new pairs of rules. In **mutation**, randomly selected bits in a rule's string are inverted.

The process of generating new populations based on prior populations of rules continues until a population, P, evolves where each rule in P satisfies a prespecified fitness threshold.

Genetic algorithms are easily parallelizable and have been used for classification, feature selection as well as other optimization problems. In data mining, they may be used to evaluate the fitness of other algorithms.

8.7.5 Reinforcement Learning

Suppose that the advertisement department at *AllElectronics* has a daily budget to do a single advertisement each day for one of the three products: either TV, or camera, or printer. Which product shall you choose to advertise on each day?

Let us first introduce some notation. Here, we have three possible *actions* a: $a \in \{TV, camera, printer\}$, representing which produce will be chosen for advertisement on a particular day. Let us assume that each action has a fixed *value* $q(a)$. The value $q(a)$ measures the *expected reward* if the action a is taken: $q(a) = E(R|a)$, where $R|a$ is the reward (such as the increased revenue) given that an action (an adversement on the corresponding product) is taken. Notice that the reward itself is a random variable since the actual increased revenue on different days might vary, even with the same advertisement. Therefore, we use its expectation to measure the value of the corresponding action, that is, the increased revenue *on average* that the corresponding action (advertisement) brings. If we know the value $q(a)$, we can just choose the action (advertisement) with the highest value, since it brings the highest (expected) rewards. However, in reality, such values are unknown. Now, what shall we do? It turns out we can resort to a powerful computational method called **reinforcement learning**, which learns through interaction.

In this example, we might decide to try advertisements with different products on different days, obverse the *actual* rewards on these days, and then adjust

	Day 1	Day 2	Day 3	Day 4	Day 5
Action	camera	camera	TV	camera	camera
Rewards	20	20	100	150	10

(a) actions and rewards

	TV	Camera	Printer
Value q(a)	?	?	?
Estimated value Q(a)	100	50	?

(b) estimated values

Figure 8.29: An example of reinforcement learning

the advertisement strategy for the future days accordingly. Suppose we try the camera advertisement for 4 days and TV advertisement for 1 day, and we observe the rewards in these five days, which are summarized in Figure 8.29(a). Based on these observed rewards, we can calculate an *estimate value* $Q(a)$ for each action (summarized Figure 8.29(b)). So, one possible strategy is that we just take the action with the highest estimated value $Q(a)$ for the next day (i.e., TV in this case). This strategy is *greedy* in the sense that it tries to make the best use (*exploitation*) of the information we have collected so far regarding the true value $q(a)$. But what if there is a big gap between the estimated value $Q(a)$ and the true value $q(a)$? In other words, it is quite possible that $q(camera) > q(TV)$, even though $Q(camera) < Q(TV)$, especially given that $Q(camera)$ and $Q(TV)$ are based on very limited data in this example (4 days for camera and 1 day for TV). Moreover, we have never tried printer advertisement at all and thus have zero knowledge about $q(printer)$. What if the printer advertisement actually has the highest value?

Another alternative strategy is the random strategy (*exploration*). Each day, we choose a random product to advertise, and observe the reward of that advertisement. If we keep running this for many days, the estimated value $Q(a)$ is likely to be very close to the true value $q(a)$. *After* that, we choose the advertisement with the highest (estimated) value. This strategy is optimal in the long-run. But we might spend many days *before* we figure out the optimal action, during which the received rewards might be low.

A better strategy is to combine the greedy strategy and the random strategy together. That is, at each day, with the probability $1-\epsilon$ $(0 < \epsilon < 1)$, we choose the action (advertisement) with the highest estimated value $Q(a)$; with the probability ϵ, we choose a random action; and at the end of the day, we use the observed reward to update the estimated value $Q(a)$. This strategy (called ϵ-greedy) is likely to obtain a better balance between the immediate return (by exploitation) and the long term return (by exploration).

In reinforcement learning, the learning agent tries to figure out what to do (e.g., choose the best product to advertise in order to maximize the overall increased revenue) by interacting with the environment (e.g., trying a few different advertisements, observing their rewards and adjusting the advertisement for next day accordingly). This is related to, but bears subtle difference from, classification. In classification, classifier receives the *instructive feedback* (i.e., the true labels for the training tuples) in order to construct the best classifier. In reinforcement learning, the learning agent receives *evaluative feedback* (e.g., the immediate reward of an action taken on a specific time step) in order to find

the best action.

The above example is called *multi-armed bandit problem*, in that we can imagine a slot machine with multiple arms, and each arm corresponds to an action (an advertisement on a product) with an unknown value. Beyond the ϵ-greedy algorithm, many alternative methods exist. For example, *upper-confidence-bound* method selects the action at each time step based on both the estimated value $Q(a)$ and the uncertainty (or confidence) of the estimation on $Q(a)$. Multi-armed bandit problem is a classic setting in reinforcement learning, where the learning agent tries to find out the best action to act in a single situation (e.g., each advertisement has a fixed value). In more complicated setting, the learning agent needs to choose different actions in different situations. In such settings, reinforcement learning is often formulated as a (finite) Markov decision process. Reinforcement learning has been applied in many application domains, including online advertisement, robotics, chess and many more.

8.8 Summary

- **Feature selection** aims to select a few most powerful features from a set of initial features. Typical methods including *filter methods*, *wrapper methods* and *embedded methods*. **Feature engineering** aims to construct more powerful features based on the initial features.

- Unlike naïve Bayesian classification (which assumes class conditional independence), **Bayesian belief networks** allow class conditional independencies to be defined between subsets of variables. They provide a graphical model of causal relationships, on which learning can be performed. Trained Bayesian belief networks can be used for classification.

- A **support vector machine** is an algorithm for the classification of both linear and nonlinear data. It transforms the original data into a higher dimensional space, from where it can find a hyperplane for data separation using essential training tuples called **support vectors**.

- A **rule-based classifier** uses a set of IF-THEN rules for classification. Rules can be extracted from a decision tree. Rules may also be generated directly from training data using sequential covering algorithms. *Frequent patterns* reflect strong associations between attribute–value pairs (or items) in data and are used in **classification based on frequent patterns**. Approaches to this methodology include associative classification and discriminant frequent pattern–based classification. In **associative classification**, a classifier is built from association rules generated from frequent patterns. In **discriminative frequent pattern–based classification**, frequent patterns serve as combined features, which are considered in addition to single features when building a classification model.

- **Semi-supervised classification** is useful when large amounts of unlabeled data exist. It builds a classifier using both labeled and unlabeled data. Examples of semi-supervised classification include *self-training* and *co-training*.

- **Active learning** is a form of supervised learning that is suitable for situations where data are abundant, yet the class labels are scarce or expensive to obtain. The learning algorithm can actively query a user (e.g., a human annotator) for labels. To keep the cost down, the active learner aims to achieve high accuracy using as few labeled instances as possible.

- **Transfer learning** aims to extract the knowledge from one or more *source tasks* and apply the knowledge to a *target task*. TrAdaBoost is an example of the *instance-based approach* to transfer learning, which reweights some of the data from the source task and uses it to learn the target task, thereby requiring fewer labeled target-task tuples.

- **Distant supervision** automatically generates a large number of noisy labeled tuples based on external knowledge or side informaiton.

- **Zero-shot learning** builds a classifier that predicts a test tuple whose class label was never observed during the training stage. An example of zero-shot learning is based on *semantic attribute classifier*.

- In many applications, data tuples arrive in a stream fashion. The key challenges of **stream data classification** include scalability, one-pass constraint and concept drifting.

- A **sequence** is an *ordered* list of values. The goal of **sequence classification** is to build a classifier to predict the label of the entire sequence or each time stamp of the sequence. Approaches to sequence classification include feature engineering based methods and distance measure based methods.

- The goal of **graph data classification** is to build a classifier to predict the label of either nodes or entire graphs. Similar to sequence classification, graph data classification can be done through feature engineering or proximity measures.

- Binary classification schemes, such as support vector machines, can be adapted to handle **multiclass classification**. This involves constructing an ensemble of binary classifiers. Error-correcting codes can be used to increase the accuracy of the ensemble.

- **Distance metric learning** aims to learn the best distance metrics for a given classification task. The basic idea is to find the optimal distance metrics so that the distance between similar tuples is small, whereas the distance between dissimilar tuples is large.

- In **genetic algorithms**, populations of rules "evolve" via operations of crossover and mutation until all rules within a population satisfy a specified threshold.
- **LIME** (Local Interpretable Model-agnostic Explanation) is a model-agnostic interpretation method. It finds a surrogate model in the local vicinity of the test tuple that balances the model fidelity and the model complexity.
- In classification, the learning agent (i.e., classifier) receives the *instructive feedback* in order to construct the best classifier. In **reinforcement learning**, the learning agent receives *evaluative feedback* in order to find the best action. Effective reinforcement learning methods need to strike a balance between *exploitation* and *exploration*.

8.9 Exercises

1. The *support vector machine* is a highly accurate classification method. However, SVM classifiers suffer from slow processing when training with a large set of data tuples. Discuss how to overcome this difficulty and develop a scalable SVM algorithm for efficient SVM classification in large data sets.
2. Compare and contrast *associative classification* and *discriminative frequent pattern–based classification*. Why is classification based on frequent patterns able to achieve higher classification accuracy in many cases than a classic decision tree method?
3. Example 8.7.1 showed the use of error-correcting codes for a *multiclass classification* problem having four classes.
   (a) Suppose that, given an unknown tuple to label, the seven trained binary classifiers collectively output the codeword 0101110, which does not match a codeword for any of the four classes. Using error correction, what class label should be assigned to the tuple?
   (b) Explain why using a 4-bit vector for the codewords is insufficient for error correction.
4. *Semi-supervised classification*, *active learning*, and *transfer learning* are useful for situations in which unlabeled data are abundant.
   (a) Describe *semi-supervised classification*, *active learning*, and *transfer learning*. Elaborate on applications for which they are useful, as well as the challenges of these approaches to classification.
   (b) Research and describe an approach to semi-supervised classification other than self-training and co-training.
   (c) Research and describe an approach to active learning other than pool-based learning.

(d) Research and describe an alternative approach to instance-based transfer learning.

5. **(SVM, RBF Kernel)** Given n training examples $(\mathbf{x}_i, y_i)$ $(i = 1, 2, ..., n)$ where $\mathbf{x}_i$ is the feature vector of i-th training example and y_i is its label, we training an support vector machine (SVM) with Radial Basis Function (RBF) kernel on the training data. Note that the RBF kernel is defined as $K_{\text{RBF}}(\mathbf{x}, \mathbf{y}) = \exp(-\gamma \|\mathbf{x} - \mathbf{y}\|_2^2)$.

   (a) Let $\mathbf{G}$ be the $n \times n$ kernel matrix of RBF kernel, i.e. $\mathbf{G}[i, j] = K_{\text{RBF}}(\mathbf{x}_i, \mathbf{x}_j)$. Prove that all eigenvalues of $\mathbf{G}$ are nonnegative.

   (b) Prove that RBF kernel is the sum of infinite number of polynomial kernels.

   (c) Suppose the distribution of training examples is shown in Figure 8.30, where '+' denotes positive example and '-' denotes the negative sample. If we set γ large enough (say 1000 or larger), what could possibly be the decision boundary of the SVM after training? Please draw it on Figure 8.30.

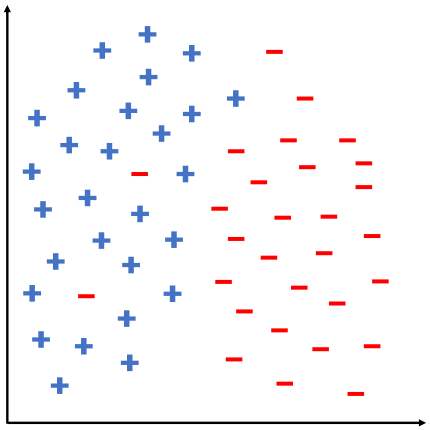


Figure 8.30: Distribution of Training Examples.

   (d) If we set γ to be infinitely large, what could possibly happen when training this SVM?

6. **(Feature selection)** Feature selection aims to select a subset of features that will be used in training. In general, there are three major types of feature selection strategy: filter method, wrapper method and embedded method.

   (a) Which type of feature selection strategy does Fisher Score belong to? Please justify your answer using 1-2 sentences.

(b) Suppose we have 6 training examples shown below. Calculate Fisher Scores for θ_0 and θ_1 and find out which one is more discriminative.

example	θ_0	θ_1	*label*
1	96	33	-
2	86	30	+
3	78	29	+
4	92	36	-
5	80	35	+
6	90	32	+

(c) Which type of feature selection strategy does LASSO belong to? Please justify your answer using 1-2 sentences.

(d) Denoting the vector of feature coefficients as $\mathbf{w}$, suppose we have a tuning parameter λ for LASSO (i.e. the LASSO penalty term is $\lambda\|\mathbf{w}\|_1$). If λ goes to infinity, what would happen to $\mathbf{w}$? Why?

7. **(Distance metric learning)** Distance metric learning aims to learn a distance metric that best describe the distance between two data points. One of the most commonly used distance metric is Mahalanobis distance. It is of the form

$$d(\mathbf{x}, \mathbf{y})^2 = (\mathbf{x} - \mathbf{y})^T \mathbf{M} (\mathbf{x} - \mathbf{y})$$

where $\mathbf{x}$ and $\mathbf{y}$ are feature vectors for two different data points. Now, supposing we have n training examples $(\mathbf{x}_i, y_i), (i = 1, 2, ..., n)$, we aim to learn the matrix $\mathbf{M}$ from the data. Research and describe one supervised method and one unsupervised method to learn the matrix $\mathbf{M}$.

8. **(Explainability)** Machine learning and data mining techniques has great potential in automatic decision making. Despite the success in deploying these techniques, many end-users do not understand how the decisions are made. Thus, it is important to study the explainability of machine learning models. Research and describe two different methods to explain the predictions of machine learning models. And justify why you think they are interpretable in your own words.

9. **(Naive Bayes and logistic regression)** Suppose that we are training a Naive Bayes classifier and a logistic regression classifier: $f : \mathbf{X} \to Y$, which maps a d-dimensional real-valued feature vector $\mathbf{X} \in \mathbb{R}^d$ to a binary class label $Y \in \{0, 1\}$. In Naive Bayes classifier, we assume that all $\mathbf{X}_i$ where $i = 1, ..., n$ are conditionally independent given the class label Y and the class prior $P(Y)$ follow the Bernoulli distribution with $P(Y = 1) = \theta$. Now, prove the equivalence of logistic regression and Naive Bayes under these two assumptions.

(a) For each $\mathbf{X}_i$, we assume it is drawn from the Gaussian distribution $P(\mathbf{X}_i | Y = k) \sim \mathcal{N}(\mu_{ik}, \sigma_{ik})$ where $k = 0, 1$. We also assume that $\sigma_{i0} = \sigma_{i1} = \sigma_i$.

(b) For each $\mathbf{X}_i$, we assume it is drawn from the Bernoulli distribution $P(\mathbf{X}_i = 1|Y = k) = p_k$ where $k = 0, 1$.

10. **(RWR and random walk graph kernel)** For graph mining using random walk with restart (RWR), the formula is $\mathbf{r}_i = c\tilde{W}\mathbf{r}_i + (1-c)\mathbf{e}_i$, where the ranking vector $\mathbf{r}_i$ will start the random walk from node i, c is the restart probability, $\tilde{W}$ is the normalized weight matrix, and $\mathbf{e}_i$ is the starting vector. Please explain:

    (a) Why RWR could capture multiple weighted relationship between nodes?

    (b) What is the similarity and the difference between random walk based graph kernel and RWR?

11. **(Distant supervision)** The traditional machine learning approach usually needs human experts to label the data examples (e.g., document, images, signals, etc.) to train a model to perform classification or regression. The human labeling process is normally expensive in terms of both time and money. Especially for the case of deep models, where the size of the training data could be extremely large.

    One alternative approach is called **distant supervision**, where the training data is generated by utilizing the existing database such as Freebase. For example, if our target is to extract the relation of friends, the item in Freebase that includes Buzz Lightyear and Woody Pride would be a positive example. By this mean, we can easily generate a large amount of labeled training data. However, for the model training, having only the positive examples are not enough. A more critical issue is how to generating the negative examples from the large-scale database. Please elaborate at least two ways to generating the negative examples in **distant supervision**.

12. **(Perceptron)** The perceptron model $y = f(\mathbf{x}) = \text{sign}(\mathbf{w}^T\mathbf{x} + b)$ can be used to learn a binary classifier from training data.

    (a) Assume there are two training samples. The positive one is $\mathbf{x}_1 = (2, 1)^T$; the negative one is $\mathbf{x}_2 = (1, 0)^T$. The learning rate $\eta = 1$. Starting from $\mathbf{w} = (1, 1)^T$ and $b = 0$, solve the parameters of the classifier.

    (b) Assume there are four training samples. The positive samples are $\mathbf{x}_1 = (1, 1)^T$ and $\mathbf{x}_2 = (0, 0)^T$; the negative samples are $\mathbf{x}_3 = (1, 0)^T$ and $\mathbf{x}_4 = (0, 1)^T$. Can we classify all training samples correctly using the perceptron model? Why?

13. **(Gradient Boosting)** XGBoost is a scalable machine learning system for tree boosting. Its objective function has a training loss and a regularization term: $\mathcal{L} = \sum_i l(y_i, \hat{y}_i) + \sum_k \Omega(f_k)$. Read the XGBoost paper and answer the following questions:

(a) What is $\hat{y}_i$? At the t-th iteration, XGBoost fixes $f_1, ..., f_{t-1}$ and trains the t-th tree model f_t. How does XGBoost approximates the training loss $l(y_i, \hat{y}_i)$ here?

(b) What is $\Omega(f_k)$? Which part in the regularization term needs to be considered at the t-th iteration?

14. **(Zero-shot Learning)** In zero-shot learning, the model observes test samples from classes that were not observed during training, and needs to predict the category they belong to. Formally, the training data has a label space Y and the testing data has a different label space Y', where $Y \cap Y' = \emptyset$. Given training data $\{(\mathbf{x}_i, y_i)|\mathbf{x}_i \in \mathbb{R}^n, i = 1, .., N\}$, zero-shot learning aims to learn a function $f : \mathbb{R}^n \to Y \cup Y'$. Suppose for each label $y \in Y \cup Y'$, a label representation vector $\mathbf{y}_i \in \mathbb{R}^m$ is also given.

(a) Suppose we aim to learn a mapping function $g : \mathbb{R}^n \to \mathbb{R}^m$ (e.g., $g(\mathbf{x}) = \mathbf{W}\mathbf{x}$) from the training data so that $\mathbf{y}_i$ is close to $g(\mathbf{x}_i)$ ($i = 1, ..., N$). Use this idea to design a zero-shot learning algorithm.

(b) Suppose we aim to learn a scoring function $g : \mathbb{R}^{n+m} \to \mathbb{R}$ (e.g., $g(\mathbf{x}, \mathbf{y}) = \mathbf{x}^T\mathbf{W}\mathbf{y}$) from the training data so that $g(\mathbf{x}_i, \mathbf{y}_i)$ is large ($i = 1, ..., N$). Use this idea to design a zero-shot learning algorithm.

15. **(Stream Data Classification)** In streaming data classification, there is an infinite sequence of the form $(\mathbf{x}, y)$. The goal is to find a function $y = f(\mathbf{x})$ that can predict the label y for an unseen instance $\mathbf{x}$. Due to evolving nature of streaming data, the dataset size is not known. In this problem, we assume $\mathbf{x} \in [-1, 1] \times [-1, 1]$ and $y \in \{-1, 1\}$, and the first eight training samples are shown below.

training sample	x_1	x_2	*label*
1	0.5	-0.5	1
2	-0.5	0.5	-1
3	0.5	0.25	1
4	0.8	0.25	1
5	0.25	0.5	-1
6	-0.5	-0.25	-1
7	-0.8	-0.25	-1
8	-0.25	-0.5	1

(a) Suppose we have stored the first eight training samples. Now a test data point $\mathbf{x}_{test} = (0.7, 0.25)^T$ comes. Use the k-nearest neighbors algorithm (k-NN) to predict the label of $\mathbf{x}_{test}$. You can set k=1.

(b) Due to the unknown size of streaming data, it could be infeasible to store all the training samples. However, if we divide the feature space into several subareas (e.g., $A_1 = [0, 1] \times [0, 1]$, $A_2 = [-1, 0) \times [0, 1]$, $A_3 = [-1, 0) \times [-1, 0)$ and $A_4 = [0, 1] \times [-1, 0)$), it is efficient to

store and update the number of positive/negative training samples in a subarea. Based on this intuition, devise an algorithm to classify stream data. What is the prediction of your algorithm on the test point $\mathbf{x}_{test} = (0.7, 0.25)^T$ given the first eight training samples?

(c) Give an example on which the k-NN algorithm and your algorithm give different predictions. To let your algorithm mimic the idea of k-NN, how to improve it given a sufficient number of training data?

16. **(Genetic Algorithm)** Briefly describe the (a) classification and (b) feature selection steps in the genetic algorithm.

17. **(Linear Regression)** Multiple linear regression is often employed for modeling the the relationship between multiple observed variables $x_i, i = 1, \ldots, p$ and a target variable y via a linear combination $y = \alpha_0 + \alpha_1 x_i + \ldots + \alpha_p x_p$. Let's consider the following example. Cereals[20] is a dataset, which contains the costumers' rating and ingredients of a variety of popular cereal products in the market.

(a) Please download the Cereals dataset.

(b) Could you implement a linear regression model, by only considering "fat" and "sugars" as observed variables and "rating" as the target variable? Please provide the obtained regression equation.

(c) t-test and p-test are two popular tools for significance testing. What are the t-test and p-test for the observed variables "fat" and "sugars"? And what do the results tell you?

18. **(Logistic Regression)** Suppose we have three positive examples $x_1 = (1, 0, 0)$, $x_2 = (0, 0, 1)$ and $x_3 = (0, 1, 0)$ and three negative examples $x_4 = (-1, 0, 0)$, $x_5 = (0, -1, 0)$ and $x_6 = (0, 0, -1)$. Apply standard gradient ascent method to train a logistic regression classifier (without regularization terms). Initialize the weight vector with two different values and set $w_0^0 = 0$ (e.g. $w_0 = (0, 0, 0, 0)', w_0 = (0, 0, 1, 0)'$). Would the final weight vector (w^*) be the same for the two different initial values? What are the values? Please explain your answer in detail. You may assume the learning rate to be a positive real constant η.

19. **(Sequence Classification)** Suppose we are given M temporal sequences $S = \{\boldsymbol{x^{(1)}}, \ldots, \boldsymbol{x^{(M)}}\}$, where each temporal sequence $\boldsymbol{x^{(m)}}$, $m = 1, \ldots, M$ consists $n^{(m)}$ temporal segments, *i.e.*, $\boldsymbol{x^{(m)}} = \{x_1^{(m)}, \ldots, x_{n^{(m)}}^{(m)}\}$. Note that the length of temporal sequences could be different. There exist both normal sequence (labeled as $Y^{(m)} = 0$) and abnormal sequence (labeled as $Y^{(m)} = 1$) in S.

(a) In the unsupervised setting, we do not have any labels for either the abnormal sequences and normal sequences. And, we observe that (1)

[20] https://drive.google.com/file/d/14TP8RCM1fr1QyYhMAWC0IjxT3CjHwCmG/view?usp=sharing

the majority of temporal sequences are normal, while only a small portion of temporal sequences in S correspond to abnormal sequences; (2) the abnormal sequences often deviate a lot from the normal sequences. Can you propose your own solution to identify the abnormal sequences out of S?

(b) In the supervised setting, we are given a training set with labeled abnormal sequences and normal sequences. Could you name one popular supervised sequence classification model to identify the abnormal sequences? What are the pros and cons of the supervised method, compared to your proposed unsupervised solution in (a)?

20. **(Reinforcement Learning)** Both reinforcement learning (RL) and the multi-armed bandit (MAB) are well known for modeling the interactions between agents and outside environments in order to achieve the maximum rewards. Interestingly, MAB is often referred to as the one-state RL problem. Could you explain why and compare the differences between these two problems?

8.10 Bibliographic Notes

hh: need to update For an introduction to Bayesian belief networks, see Darwiche [Dar10] and Heckerman [Hec96]. For a thorough presentation of probabilistic networks, see Pearl [Pea88] and Koller and Friedman [KF09]. Solutions for learning the belief network structure from training data given observable variables are proposed in Cooper and Herskovits [CH92]; Buntine [Bun94]; and Heckerman, Geiger, and Chickering [HGC95]. Algorithms for inference on belief networks can be found in Russell and Norvig [RN95] and Jensen [Jen96]. The method of gradient descent, described in sec:train-bbnSection , for training Bayesian belief networks, is given in Russell, Binder, Koller, and Kanazawa [RBKK95]. The example given in Figure 8.4 is adapted from Russell et al. [RBKK95].

Alternative strategies for learning belief networks with hidden variables include application of Dempster, Laird, and Rubin's [DLR77] EM (Expectation Maximization) algorithm (Lauritzen [Lau95]) and methods based on the minimum description length principle (Lam [Lam98]). Cooper [Coo90] showed that the general problem of inference in unconstrained belief networks is NP-hard. Limitations of belief networks, such as their large computational complexity (Laskey and Mahoney [LM97]), have prompted the exploration of hierarchical and composable Bayesian models (Pfeffer, Koller, Milch, and Takusagawa [PKMT99] and Xiang, Olesen, and Jensen [XOJ00]). These follow an object-oriented approach to knowledge representation. Fishelson and Geiger [FG02] present a Bayesian network for genetic linkage analysis.

Support Vector Machines (SVMs) grew out of early work by Vapnik and Chervonenkis on statistical learning theory [VC71]. The first paper on SVMs was presented by Boser, Guyon, and Vapnik [BGV92]. More detailed accounts

can be found in books by Vapnik [Vap95, Vap98]. Good starting points include the tutorial on SVMs by Burges [Bur98], as well as textbook coverage by Haykin [Hay08], Kecman [Kec01], and Cristianini and Shawe-Taylor [CST00]. For methods for solving optimization problems, see Fletcher [Fle87] and Nocedal and Wright [NW99]. These references give additional details alluded to as "fancy math tricks" in our text, such as transformation of the problem to a Lagrangian formulation and subsequent solving using Karush-Kuhn-Tucker (KKT) conditions.

For the application of SVMs to regression, see Schölkopf, Bartlett, Smola, and Williamson [SBSW99] and Drucker, Burges, Kaufman, Smola, and Vapnik [DBK+97]. Approaches to SVM for large data include the sequential minimal optimization algorithm by Platt [Pla98], decomposition approaches such as in Osuna, Freund, and Girosi [OFG97], and CB-SVM, a microclustering-based SVM algorithm for large data sets, by Yu, Yang, and Han [YYH03]. A library of software for support vector machines is provided by Chang and Lin at *www.csie.ntu.edu.tw/∼cjlin/libsvm/*, which supports multiclass classification.

There are several examples of rule-based classifiers. These include AQ15 (Hong, Mozetic, and Michalski [HMM86]), CN2 (Clark and Niblett [CN89]), ITRULE (Smyth and Goodman [SG92]), RISE (Domingos [Dom94]), IREP (Furnkranz and Widmer [FW94]), RIPPER (Cohen [Coh95]), FOIL (Quinlan and Cameron-Jones [Qui90, QCJ93]), and Swap-1 (Weiss and Indurkhya [WI98]). For the extraction of rules from decision trees, see Quinlan [Qui87, Qui93]. Rule refinement strategies that identify the most interesting rules among a given rule set can be found in Major and Mangano [MM95].

Many algorithms have been proposed that adapt frequent pattern mining to the task of classification. Early studies on associative classification include the CBA algorithm, proposed in Liu, Hsu, and Ma [LHM98]. A classifier that uses *emerging patterns* (itemsets with support that varies significantly from one data set to another) is proposed in Dong and Li [DL99] and Li, Dong, and Ramamohanarao [LDR00]. CMAR is presented in Li, Han, and Pei [LHP01]. CPAR is presented in Yin and Han [YH03]. Cong, Tan, Tung, and Xu describe RCBT, a method for mining top-k covering rule groups for classifying high-dimensional gene expression data with high accuracy [CTTX05].

Wang and Karypis [WK05] present HARMONY (Highest confidence classificAtion Rule Mining fOr iNstance-centric classifYing), which directly mines the final classification rule set with the aid of pruning strategies. Lent, Swami, and Widom [LSW97] propose the ARCS system regarding mining multidimensional association rules. It combines ideas from association rule mining, clustering, and image processing, and applies them to classification. Meretakis and Wüthrich [MW99] propose constructing a naïve Bayesian classifier by mining long itemsets. Veloso, Meira, and Zaki [VMZ06] propose an association rule–based classification method based on a lazy (noneager) learning approach, in which the computation is performed on a demand-driven basis.

Studies on discriminative frequent pattern–based classification were conducted by Cheng, Yan, Han, and Hsu [CYHH07] and Cheng, Yan, Han, and Yu [CYHY08]. The former work establishes a theoretical upper bound on the

discriminative power of frequent patterns (based on either information gain [Qui86] or Fisher score [DHS01]), which can be used as a strategy for setting minimum support. The latter work describes the DDPMine algorithm, which is a direct approach to mining discriminative frequent patterns for classification in that it avoids generating the complete frequent pattern set. H. Kim, S. Kim, Weninger, et al. proposed an NDPMine algorithm that performs frequent and discriminative pattern–based classification by taking *repetitive* features into consideration [KKW+10].

For texts on genetic algorithms, see Goldberg [Gol89], Michalewicz [Mic92], and Mitchell [Mit96].

Work on multiclass classification is described in Hastie and Tibshirani [HT98], Tax and Duin [TD02], and Allwein, Shapire, and Singer [ASS00]. Zhu [Zhu05] presents a comprehensive survey on semi-supervised classification. For additional references, see the book edited by Chapelle, Schölkopf, and Zien [ClZ06]. Dietterich and Bakiri [DB95] propose the use of error-correcting codes for multiclass classification. For a survey on active learning, see Settles [Set10]. Pan and Yang present a survey on transfer learning [PY10]. The TrAdaBoost boosting algorithm for transfer learning is given in Dai, Yang, Xue, and Yu [DYXY07].

Bibliography

[ASS00] E. L. Allwein, R. E. Shapire, and Y. Singer. Reducing multiclass to binary: A unifying approach for margin classifiers. *J. Machine Learning Research*, 1:113–141, 2000.

[BGV92] , B. Boser, I. Guyon, and V. N. Vapnik. A training algorithm for optimal margin classifiers. In *Proc. Fifth Annual Workshop on Computational Learning Theory*, pages 144–152, ACM Press: San Mateo, CA, 1992.

[Bun94] W. L. Buntine. Operations for learning with graphical models. *J. Artificial Intelligence Research*, 2:159–225, 1994.

[Bur98] C. J. C. Burges. A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery*, 2:121–168, 1998.

[CH92] G. Cooper and E. Herskovits. A Bayesian method for the induction of probabilistic networks from data. *Machine Learning*, 9:309–347, 1992.

[ClZ06] O. Chapelle, B. Schölkopf, and A. Zien. *Semi-supervised Learning*. MIT Press, 2006.

[CN89] P. Clark and T. Niblett. The CN2 induction algorithm. *Machine Learning*, 3:261–283, 1989.

[Coh95] W. Cohen. Fast effective rule induction. In *Proc. 1995 Int. Conf. Machine Learning (ICML'95)*, pages 115–123, Tahoe City, CA, July 1995.

[Coo90] G. F. Cooper. The computational complexity of probabilistic inference using Bayesian belief networks. *Artificial Intelligence*, 42:393–405, 1990.

[CST00] N. Cristianini and J. Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-Based Learning Methods*. Cambridge Univ. Press, 2000.

[CTTX05] G. Cong, K.-Lee Tan, A.K.H. Tung, and X. Xu. Mining top-k covering rule groups for gene expression data. In *Proc. 2005 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'05)*, pages 670–681, Baltimore, MD, June 2005.

[CYHH07] H. Cheng, X. Yan, J. Han, and C.-W. Hsu. Discriminative frequent pattern analysis for effective classification. In *Proc. 2007 Int. Conf. Data Engineering (ICDE'07)*, pages 716–725, Istanbul, Turkey, April 2007.

[CYHY08] H. Cheng, X. Yan, J. Han, and P. S. Yu. Direct discriminative pattern mining for effective classification. In *Proc. 2008 Int. Conf. Data Engineering (ICDE'08)*, Cancun, Mexico, April 2008.

[Dar10] A. Darwiche. Bayesian networks. *Comm. ACM*, 53:80–90, 2010.

[DB95] T. G. Dietterich and G. Bakiri. Solving multiclass learning problems via error-correcting output codes. *J. Artificial Intelligence Research*, 2:263–286, 1995.

[DBK+97] H. Drucker, C. J. C. Burges, L. Kaufman, A. Smola, and V. N. Vapnik. Support vector regression machines. In M. Mozer, M. Jordan, and T. Petsche, editors, *Advances in Neural Information Processing Systems 9*, pages 155–161. MIT Press, 1997.

[DHS01] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification* (2nd ed.). John Wiley & Sons, 2001.

[DL99] G. Dong and J. Li. Efficient mining of emerging patterns: Discovering trends and differences. In *Proc. 1999 Int. Conf. Knowledge Discovery and Data Mining (KDD'99)*, pages 43–52, San Diego, CA, Aug. 1999.

[DLR77] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *J. Royal Statistical Society, Series B*, 39:1–38, 1977.

[Dom94] P. Domingos. The RISE system: Conquering without separating. In *Proc. 1994 IEEE Int. Conf. Tools with Artificial Intelligence (TAI'94)*, pages 704–707, New Orleans, LA, 1994.

[DYXY07] W. Dai, Q. Yang, G. Xue, and Y. Yu. Boosting for transfer learning. In *Proc. 24th Intl. Conf. Machine Learning*, pages 193–200, Jun. 2007.

[FG02] M. Fishelson and D. Geiger. Exact genetic linkage computations for general pedigrees. *Disinformation*, 18:189–198, 2002.

[Fle87] R. Fletcher. *Practical Methods of Optimization.* John Wiley & Sons, 1987.

[FW94] J. Furnkranz and G. Widmer. Incremental reduced error pruning. In *Proc. 1994 Int. Conf. Machine Learning (ICML'94)*, pages 70–77, New Brunswick, NJ, 1994.

[Gol89] D. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

[Hay08] S. Haykin. *Neural Networks and Learning Machines*. Prentice Hall, Saddle River, NJ, 2008.

[Hec96] D. Heckerman. Bayesian networks for knowledge discovery. In U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 273–305. MIT Press, 1996.

[HGC95] D. Heckerman, D. Geiger, and D. M. Chickering. Learning Bayesian networks: The combination of knowledge and statistical data. *Machine Learning*, 20:197–243, 1995.

[HMM86] J. Hong, I. Mozetic, and R. S. Michalski. AQ15: Incremental learning of attribute-based descriptions from examples, the method and user's guide. In *Report ISG 85-5, UIUCDCS-F-86-949,*, Department of Comp. Science, University of Illinois at Urbana-Champaign, 1986.

[HT98] T. Hastie and R. Tibshirani. Classification by pairwise coupling. *Ann. Statistics*, 26:451–471, 1998.

[Jen96] F. V. Jensen. *An Introduction to Bayesian Networks*. Springer Verlag, 1996.

[Kec01] V. Kecman. *Learning and Soft Computing*. MIT Press, 2001.

[KF09] D. Koller and N. Friedman. *Probabilistic Graphical Models: Principles and Techniques*. The MIT Press, 2009.

[KKW+10] H. S. Kim, S. Kim, T. Weninger, J. Han, and T. Abdelzaher. NDPMine: Efficiently mining discriminative numerical features for pattern-based classification. In *Proc. 2010 European Conf. Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECMLPKDD'10)*, Barcelona, Spain, Sept. 2010.

[Lam98] W. Lam. Bayesian network refinement via machine learning approach. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 20:240–252, 1998.

[Lau95] S. L. Lauritzen. The EM algorithm for graphical association models with missing data. *Computational Statistics and Data Analysis*, 19:191–201, 1995.

[LDR00] J. Li, G. Dong, and K. Ramamohanrarao. Making use of the most expressive jumping emerging patterns for classification. In *Proc. 2000 Pacific-Asia Conf. Knowledge Discovery and Data Mining (PAKDD'00)*, pages 220–232, Kyoto, Japan, April 2000.

[LHM98] B. Liu, W. Hsu, and Y. Ma. Integrating classification and association rule mining. In *Proc. 1998 Int. Conf. Knowledge Discovery and Data Mining (KDD'98)*, pages 80–86, New York, NY, Aug. 1998.

[LHP01] W. Li, J. Han, and J. Pei. CMAR: Accurate and efficient classification based on multiple class-association rules. In *Proc. 2001 Int. Conf. Data Mining (ICDM'01)*, pages 369–376, San Jose, CA, Nov. 2001.

[LM97] K. Laskey and S. Mahoney. Network fragments: Representing knowledge for constructing probabilistic models. In *Proc. 13th Annual Conf. Uncertainty in Artificial Intelligence*, pages 334–341, Morgan Kaufmann: San Francisco, CA, Aug. 1997.

[LSW97] B. Lent, A. Swami, and J. Widom. Clustering association rules. In *Proc. 1997 Int. Conf. Data Engineering (ICDE'97)*, pages 220–231, Birmingham, England, April 1997.

[Mic92] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer Verlag, 1992.

[Mit96] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1996.

[MM95] J. Major and J. Mangano. Selecting among rules induced from a hurricane database. *J. Intelligent Information Systems*, 4:39–52, 1995.

[NW99] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer Verlag, 1999.

[OFG97] E. Osuna, R. Freund, and F. Girosi. An improved training algorithm for support vector machines. In *Proc. 1997 IEEE Workshop on Neural Networks for Signal Processing (NNSP'97)*, pages 276–285, Amelia Island, FL, Sept. 1997.

[Pea88] J. Pearl. *Probabilistic Reasoning in Intelligent Systems*. Morgan Kauffman, 1988.

[PKMT99] A. Pfeffer, D. Koller, B. Milch, and K. Takusagawa. SPOOK: A system for probabilistic object-oriented knowledge representation. In *Proc. 15th Annual Conf. Uncertainty in Artificial Intelligence (UAI'99)*, pages 541–550, Stockholm, Sweden, 1999.

[Pla98] J. C. Platt. Fast training of support vector machines using sequential minimal optimization. In B. Schoelkopf, C. J. C. Burges, and A. Smola, editors, *Advances in Kernel Methods—Support Vector Learning*, pages 185–208. MIT Press, 1998.

[PY10] S. J. Pan and Q. Yang. A survey on transfer learning. *IEEE Trans. on Knowledge and Data Engineering*, 22:1345–1359, 2010.

[QCJ93] J. R. Quinlan and R. M. Cameron-Jones. FOIL: A midterm report. In *Proc. 1993 European Conf. Machine Learning (ECML'93)*, pages 3–20, Vienna, Austria, 1993.

[Qui86] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[Qui87] J. R. Quinlan. Simplifying decision trees. *Int. J. Man-Machine Studies*, 27:221–234, 1987.

[Qui90] J. R. Quinlan. Learning logic definitions from relations. *Machine Learning*, 5:139–166, 1990.

[Qui93] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[RBKK95] S. Russell, J. Binder, D. Koller, and K. Kanazawa. Local learning in probabilistic networks with hidden variables. In *Proc. 1995 Joint Int. Conf. Artificial Intelligence (IJCAI'95)*, pages 1146–1152, Montreal, Canada, Aug. 1995.

[RN95] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, 1995.

[SBSW99] B. Schlökopf, P. L. Bartlett, A. Smola, and R. Williamson. Shrinking the tube: A new support vector regression algorithm. In M. S. Kearns, S. A. Solla, and D. A. Cohn, editors, *Advances in Neural Information Processing Systems 11*, pages 330–336. MIT Press, 1999.

[Set10] B. Settles. Active learning literature survey. In *Computer Sciences Technical Report 1648*, University of Wisconsin-Madison, 2010.

[SG92] P. Smyth and R. M. Goodman. An information theoretic approach to rule induction. *IEEE Trans. Knowledge and Data Engineering*, 4:301–316, 1992.

[TD02] D. M. J. Tax and R. P. W. Duin. Using two-class classifiers for multiclass classification. In *Proc. 16th Intl. Conf. Pattern Recognition (ICPR'2002)*, pages 124–127, 2002.

[Vap95] V. N. Vapnik. *The Nature of Statistical Learning Theory*. Springer Verlag, 1995.

[Vap98] V. N. Vapnik. *Statistical Learning Theory.* John Wiley & Sons, 1998.

[VC71] V. N. Vapnik and A. Y. Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability and its Applications*, 16:264–280, 1971.

[VMZ06] A. Veloso, W. Meira, and M. Zaki. Lazy associative classificaiton. In *Proc. 2006 Int. Conf. Data Mining (ICDM'06)*, pages 645–654, 2006.

[WI98] S. M. Weiss and N. Indurkhya. *Predictive Data Mining.* Morgan Kaufmann, 1998.

[WK05] J. Wang and G. Karypis. HARMONY: Efficiently mining the best rules for classification. In *Proc. 2005 SIAM Conf. Data Mining (SDM'05)*, pages 205–216, Newport Beach, CA, April 2005.

[XOJ00] Y. Xiang, K. G. Olesen, and F. V. Jensen. Practical issues in modeling large diagnostic systems with multiply sectioned Bayesian networks. *Intl. J. Pattern Recognition and Artificial Intelligence (IJPRAI)*, 14:59–71, 2000.

[YH03] X. Yin and J. Han. CPAR: Classification based on predictive association rules. In *Proc. 2003 SIAM Int. Conf. Data Mining (SDM'03)*, pages 331–335, San Fransisco, CA, May 2003.

[YYH03] H. Yu, J. Yang, and J. Han. Classifying large data sets using SVM with hierarchical clusters. In *Proc. 2003 ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining (KDD'03)*, pages 306–315, Washington, DC, Aug. 2003.

[Zhu05] X. Zhu. Semi-supervised learning literature survey. In *Computer Sciences Technical Report 1530*, University of Wisconsin-Madison, 2005.