CSC4008 Data Mining

117010279 Ziren WANG

April 2020

6 Advanced Classification

6.1 Bayesian Belief Networks

6.1.1 Outlook

Recall in Naive Bayesian method, we assume the independence among attributes. However, this is not the case in reality. How can we model the dependencies among attributes? one proposal is Bayesian belief networks.
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.
Let $\boldsymbol{X} = (x_1, ..., x_n)$ be a data tuple described by the variables or attributes $Y_1, ..., Y_n$, respectively. Recall that each variable is conditionally independent of its nondescendants in the network graph, given its parents. This allows the network to provide a complete representation of the existing joint probability distribution with the following equation:

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

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

6.1.2 Training

1. The network **topology** (or "layout" of nodes and arcs) may be constructed by human experts or inferred from the data. If network structure unknown and all variables observable then we need to search through the model space to reconstruct network topology
2. The network **weights** can be searched by gradient descent strategy.

For our problem, we maximize $P_w(D) = \prod_{d=1}^{|D|} P_w(\boldsymbol{X}_d)$. This can be done by following the gradient of $\ln P_w(S)$, which makes the problem simpler. Given the network topology and initialized w_{ijk}, the algorithm proceeds as follows:

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

$$\frac{\partial \ln P_w(D)}{\partial w_{ijk}} = \sum_{d=1}^{|D|} \frac{P(Y_i = y_{ij}, U_i = u_{ik}|\boldsymbol{X}_d)}{w_{ijk}}. \quad (9.2)$$

The probability on the right side of Eq. (9.2) 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 $\boldsymbol{X}_d$, then 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

$$w_{ijk} \leftarrow w_{ijk} + (l)\frac{\partial \ln P_w(D)}{\partial w_{ijk}}, \quad (9.3)$$

where l is the **learning rate** representing the step size and $\frac{\partial \ln P_w(D)}{\partial w_{ijk}}$ is computed from Eq. (9.2). 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. (9.3).

Figure 1: gradient descent strategy

6.2 ANN: Classification by Back-propagation

6.2.1 Perceptron

$y = \sum_{j=1}^{d} w_j x_j + w_0 = W^T X \; where: W = [w_0, w_1, ...w_d] \; and \; X = [1, x_1, ...x_d]^T$

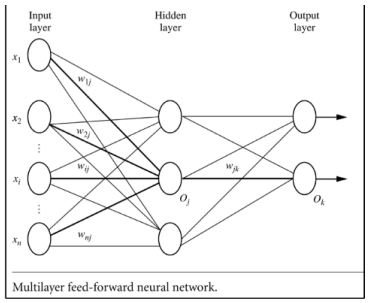
6.2.2 Algorithm

Algorithm: Backpropagation. Neural network learning for classification or numeric prediction, using the backpropagation algorithm.
Input:

- D, a data set consisting of the training tuples and their associated target values;
- l, the learning rate;
- *network*, a multilayer feed-forward network.

Output: A trained neural network.
Method:

(1) Initialize all weights and biases in *network*;
(2) **while** terminating condition is not satisfied {
(3) **for** each training tuple $\boldsymbol{X}$ in D
(4) // Propagate the inputs forward:
(5) **for** each input layer unit j {
(6) $O_j = I_j$; // output of an input unit is its actual input value
(7) **for** each hidden or output layer unit j {
(8) $I_j = \sum_i w_{ij} O_i + \theta_j$; //compute the net input of unit j with respect to the previous layer, i
(9) $O_j = \frac{1}{1+e^{-I_j}}$; } // compute the output of each unit j
(10) // Backpropagate the errors:
(11) **for** each unit j in the output layer
(12) $Err_j = O_j(1-O_j)(T_j - O_j)$; // compute the error
(13) **for** each unit j in the hidden layers, from the last to the first hidden layer
(14) $Err_j = O_j(1-O_j)\sum_k Err_k w_{jk}$; // compute the error with respect to the next higher layer, k
(15) **for** each weight w_{ij} in *network* {
(16) $\Delta w_{ij} = (l) Err_j O_i$; // weight increment
(17) $w_{ij} = w_{ij} + \Delta w_{ij}$; } // weight update
(18) **for** each bias θ_j in *network* {
(19) $\Delta\theta_j = (l) Err_j$; // bias increment
(20) $\theta_j = \theta_j + \Delta\theta_j$; } // bias update
(21) } }



Multilayer feed-forward neural network.

$$I_j = \sum_i w_{ij} O_i + \theta_j, \quad (9.4)$$

$$O_j = \frac{1}{1+e^{-I_j}}. \quad (9.5)$$

$$Err_j = O_j(1-O_j)(T_j - O_j), \quad (9.6)$$

$$Err_j = O_j(1-O_j)\sum_k Err_k w_{jk}, \quad (9.7)$$

$$\Delta w_{ij} = (l) Err_j O_i. \quad (9.8)$$

$$w_{ij} = w_{ij} + \Delta w_{ij}. \quad (9.9)$$

$$\Delta\theta_j = (l) Err_j. \quad (9.10)$$

$$\theta_j = \theta_j + \Delta\theta_j. \quad (9.11)$$

Figure 2: Back-propagation Algorithm

6.2.3 Efficiency and Interpretability

Each epoch takes $O(|D| * w)$, $with |D| tuples and w weights$. But the number of epochs can be exponential to the number of inputs in the worst case. Thus, for easier comprehension, **Rule extraction** by network pruning:

- Simplify the network structure by removing weighted links that have the least effect on the trained network;
- Perform link, unit, or activation value clustering;
- the set of input and activation values are studied to derive rules describing the relationship between the input and hidden unit layers.

One can use sensitivity analysis to access the impact that a given input variable has on a network output.

6.3 Support Vector Machines

6.3.1 Model

$min \; \frac{1}{2}||w||^2$
$s.t. \; r^t(w^t x^t + w_0) \geq +1, \forall t$ where: $r^t = +1 \; if \; x^t \in C_1$ and $r^t = -1 \; if \; x^t \in C_2$
Here, the **Margin** is $\frac{|w^t x^t + w_0|}{||w||}$. We require $\frac{r^t|w^t x^t + w_0|}{||w||} \geq \rho, \forall t$
Notice: the complexity of trained classifier is characterized by the number of support vectors rather than the dimensionality of the data. The support vectors are the essential or critical training examples, since they must lie closet to the decision boundary (MMH).

6.3.2 Non-Linear Boundary

Rather than run SVM on x_i, we run it on $\phi(x_i)$. This is so called **Kernelization**.
Define: $z = \phi(x)$, then the SVM solution becomes:
$w = \sum_t \alpha^t r^t z^t = \sum_t \alpha^t r^t \phi(x^t)$ *and* $g(x) = w^t \phi(x) = \sum_t \alpha^t r^t \phi(x^t)^T \phi(x) = \sum_t \alpha^t r^t K(x^t, x)$
Typical Kernel Functions can be: Polynomial kernel of degree h; Gaussian radial basis function kernel or Sigmoid kernel.

6.4 Lazy Learners: Learning from Your Neighbors

Lazy learning just simply stores the training data (or only minor processing) and waits until it is given a test tuple. On the other hand, eager learning give a set of training tuples, constructs a classification model before receiving new data to classify. we can see that lazy learner uses a richer hypothesis space while eager learner must commit to a single hypothesis that covers the entire instance space.

6.4.1 KNN Algorithm

All instances correspond to points in the n-D space. The nearest neighbor are defined in terms of **Euclidean distance** $dist(x_1, x_2)$.
1. For discrete-valued target function, KNN returns the most common value among the k training examples nearest to x_q. 2. For real-valued target function, KNN returns the mean values of the k nearest neighbors. When taking votes, a robust way is to consider distance-weighted so that robust enough to noisy data. **Curse of dimensionality**: distance between neighbors could be dominated by irrelevant attributes. To solve this problem, we need to eliminate the least relevant attributes.

6.4.2 Case-Based Reasoning

Case-based reasoning (CBR) classifiers use a database of problem solutions to solve new problems. CBR stores the tuples or "cases" for problem solving as complex symbolic descriptions. Applications can be customer-service (product-related diagnosis), legal ruling etc.
When given a new case to classify, a case-based reasoner will first check if an identical training case exists. If one is found, then the accompanying solution to that case is returned. If no identical case is found, then the case-based reasoner will search for training cases having components that are similar to those of the new case. Conceptually, these training cases may be considered as neighbors of the new case. If cases are represented as graphs, this involves searching for subgraphs that are similar to subgraphs within the new case.
Challenges in case-based reasoning include finding a good similarity metric (e.g., for matching subgraphs) and suitable methods for combining solutions. Other challenges include the selection of salient features for indexing training cases and the development of efficient indexing techniques.