Classification

Nikunj C. Oza
NASA Ames Research Center
nikunj.c.oza@nasa.gov

A supervised learning task involves constructing a mapping from an input data space (normally described by several features) to an output space. A set of training examples—examples with known output values—is used by a learning algorithm to generate a model. This model is intended to approximate the mapping between the inputs and outputs. This model can be used to generate predicted outputs for inputs that have not been seen before. Within supervised learning, one type of task is a classification learning task, in which each output consists of one or more classes to which the corresponding input belongs. For example, we may have data consisting of observations of sunspots. In a classification learning task, our goal may be to learn to classify sunspots into one of several types. Each example may correspond to one candidate sunspot with various measurements or just an image. A learning algorithm would use the supplied examples to generate a model that approximates the mapping between each supplied set of measurements and the type of sunspot. This model can then be used to classify previously unseen sunspots based on the candidate's measurements. The generalization performance of a learned model (how closely the target outputs and the model's predicted outputs agree for examples that have not been presented to the learning algorithm) would provide an indication of how well the model has learned the desired mapping.

More formally, a classification learning algorithm L takes a training set T as its input. The training set consists of $|T|$ *examples* or *instances*. It is assumed that there is a probability distribution $\mathcal{D}$ from which all training examples are drawn independently—that is, all the training examples are *independently and identically distributed* (i.i.d). The ith training example is of the form $(\mathbf{x}_i, y_i)$, where $\mathbf{x}_i$ is a vector of values of several features and y_i represents the class to be predicted.[1] In the sunspot classification example given above, each training example would represent one sunspot's classification (y_i) and the corresponding set of measurements ($\mathbf{x}_i$). The output of a supervised learning algorithm is a model h that approximates the unknown mapping from the inputs to the outputs. In our example, h would map from the sunspot measurements to the

[1] In this chapter, we will assume that each individual example belongs to only one class. However, we allow two separate examples with the same input values to be in different classes. For example, there can be two examples having inputs $(0.8, 0.3)$, where one example is in class 1 and the other is in class 2. That is, we will allow for some noise in the input and output generation processes.

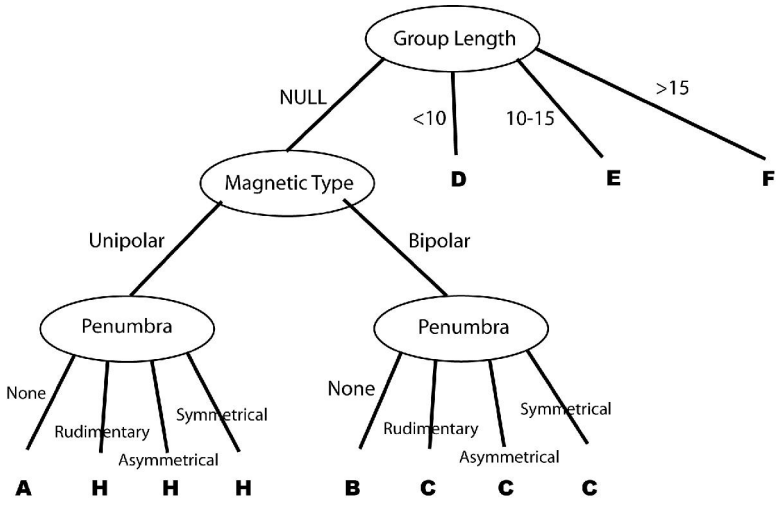

Figure 1: Example decision tree

type of sunspot.

We may have a *test set* S—a set of examples not used in training that we use to test how well the model h predicts the outputs on new examples. Just as with the examples in T, the examples in S are assumed to be independent and identically distributed (i.i.d.) draws from the distribution $\mathcal{D}$. We measure the error of h on the test set as the proportion of test cases that h misclassifies:

$$\frac{1}{|S|} \sum_{(x,y) \in S} I(h(x) \neq y)$$

where $I(v)$ is the indicator function—it returns 1 if v is true and 0 otherwise.

In our sunspot classification example, we would identify additional examples of sunspots that were not used in generating the model, and use these to determine how accurate the model is—the fraction of the test samples that the model classifies correctly.

An example of a classification model is the decision tree shown in figure 1. We will discuss the decision tree learning algorithm in more detail later—for now, we assume that, given a training set with examples of sunspots, this decision tree is derived. This can be used to classify previously unseen examples of sunpots. For example, if a new sunspot's inputs indicate that its 'Group Length' is in the range 10-15, then the decision tree would classify the sunspot as being of type 'E,' whereas if the 'Group Length' is 'NULL,' the 'Magnetic Type' is 'bipolar,' and the 'Penumbra' is 'rudimentary,' then it would be classified as type 'C.'

In this chapter, we will add to the above description of classification problems. We will discuss decision trees and several other classification models. In particular, we will discuss the learning algorithms that generate these classifica-

tion models, how to use them to classify new examples, and the strengths and weaknesses of these models. We will end with pointers to further reading on classification methods applied to astronomy data.

0.1 Features of Classification Model Learning Algorithms

As indicated above, classification learning aims to use a training set to generate a model that can be used to classify new, typically previously-unseen, examples. In some cases, the aim is to determine the most likely class given the new example's inputs. In other cases, one may measure the probabilities of each class given the example's inputs and use this as a measure of confidence—if the highest probability minus the second highest probability is high, then the model is quite confident in the highest probability class. If this probability difference is low, then the confidence is low. Regardless of the problem, one needs an estimate of the probability distribution of the class given the inputs, which is $P(Y|X)$ where X represents the new example and Y represents the vector of possible classes. Because we have a training set that is used to determine the probabilities, we are actually interested in $P(Y|X, D)$, where D represents the training data.

One example of a machine learning technique that estimates $P(Y|X, D)$ is nearest-neighbor classification [19]. Given a new example X, nearest-neighbor classification identifies one or more neighbors within D and chooses the class for X based on the classes of the neighbors. One possibility is to find the nearest neighbor and assign to X the class of that nearest neighbor. Another possibility is to find K nearest neighbors for some $K \geq 2$ and assign the class most frequently seen among those neighbors or perhaps each class can get a weighted vote, with nearer neighbors getting stronger votes. One difficulty with this scheme is that the memory requirements for nearest-neighbor based techniques grow with the size of the training set. Given how large modern datasets are in many disciplines including astronomy, nearest-neighbor based techniques are mostly impractical. One needs a machine learning technique that can compress the information from the training set into a form that is of constant size regardless of the size of the training set but is nevertheless able to use information learned from the training set to classify new examples.

The decision tree example given above is a good example of such a model. Decision trees cannot grow to a depth greater than the number of input features and cannot have a total number of nodes beyond $V^{|F|}$, where V is the maximum number of values that any feature can have and F is the set of features. This is independent of the number of training examples.

As explained above, classifier learning algorithms take a training set as input and return a model as output. However, this model is not of an arbitrary form. The model is drawn from a family of possible models by finding which model is best according to some pre-defined criteria. For example, the decision tree learning algorithm that we will describe later may have returned the decision tree shown in figure 1. However, the set of possible models that the learning algorithm can return is limited. In particular, this learning algorithm will always

return a decision tree and never another type of model such as a neural network. Additionally, as mentioned above, the decision tree will have depth that is no greater than the number of input features because each input feature can be used at most once on any path from the root to a leaf in the tree. Other learning algorithms may have different restrictions on the models that they may return. For example, a decision stump learning algorithm run on sunspot data may return only the top node ('Group Length') shown in figure 1, the three leaves shown on the right with classes 'D, 'E', and 'F,' and the subtree rooted by the feature 'Magnetic Type' replaced by the class most frequently seen among the training examples that have 'Group Length' as 'NULL.' The set of possible models that a learning algorithm may return is called a *model family.*

More formally, instead of merely returning $P(Y|X, D)$, we often condition on a model family H and end up deriving the following:

$$\begin{aligned} P(Y|X,D) &= \sum_{h=1}^{|H|} P(Y|X,H_h,D)P(H_h|D) && (1)\\ &= \sum_{h=1}^{|H|} P(Y|X,H_h)P(H_h|D) && (2)\\ &= \sum_{h=1}^{|H|} \frac{P(Y|X,H_h)P(D|H_h)P(H_h)}{P(D)}. && (3) \end{aligned}$$

In equation (1), we replace $P(Y|X, H_h, D)$ with $P(Y|X, H_h)$ to get equation (2) because H_h is meant to contain all the information that we would ordinarily get from D. For example, if H_h is a decision tree learned from the training data D, we no longer need D but rather can just use H_h to return the predicted class or the probability of each possible class. Equation (2) conditions on the set of models, and returns a sum of probabilities $P(Y|X, H_h)$, each of which is weighted by $P(H_h|D)$.

Even using equation (2) seems impractical because H can be a very large family. For example, H could be the entire set of decision trees over the set of features F. If each of the features is binary, then the total number of decision trees that could be returned by the learning algorithm is

$$\sum_{f=1}^{|F|}(|F| - f + 1)^{2^{f-1}}$$

which can become large very quickly. For this reason, typical learning algorithms, such as the decision tree learning algorithm, choose one model from within H that is best according to some criterion. If the criterion is maximizing $P(H_h|D)$, then the chosen model is called the *Maximum A Posteriori* (MAP) model, because it is the model with the highest posterior probability given the training data. Equation (3) is constructed by replacing $P(H_h|D)$ using Bayes's

Rule with $P(D|H_h)P(H_h)/P(D)$. Since $P(D)$ is the same for all models and often the prior probabilities of the models $P(H_h)$ are assumed to be the same, some learning methods choose the model that maximizes $P(D|H_h)$, which is the probability that the training data was generated by model H_h and is often called the *likelihood* of the data. The model that maximizes the likelihood is called the *Maximum Likelihood* (ML) model. For definitions of MAP and ML models, as well as how they appear in machine learning methods for classification, see [26].

Clearly, using one model has risks—that model, although the single best performing model on average, may perform poorly on some parts of the input space. Also, the model chosen is the best performing model on that particular training dataset. Given a slightly different training set, a different model may be more appropriate. There is another possible method of performing *model selection* rather than either using all possible models or one model. One can use some intermediate number of models, all of which perform reasonably well (e.g., have relatively high values of $P(H_h|D)$), but which are different from one another so that they cover more of the input space or more of the space of models optimal over the possible training sets. Methods that return multiple such models are referred to as *ensembles*, and are the subject of another chapter in this book.

The alert reader may have been inspired by the use of Bayes's Rule in equation 3 and noted that

$$P(Y|X) = \frac{P(X|Y)P(Y)}{P(X)}.$$

This may make one wonder whether there are methods that, instead of attempting to learn $P(Y|X)$ or a similar function that generates the predicted class or probability distribution over classes given the input, attempt to learn $P(X|Y)$, $P(Y)$, and $P(X)$. There are methods that attempt to learn $P(X|Y)$ and $P(Y)$. Learning $P(X)$ separately is unnecessary because

$$P(X) = \sum_{y=1}^{|C|} P(X|Y=y)P(Y=y)$$

where C is the set of possible classes. Note that every factor in the above summand is either of the form $P(X|Y)$ or $P(Y)$, which are already being calculated.

Methods that learn an estimate of $P(Y|X)$ are called *discriminative* methods because they attempt to find a model that discriminates between multiple classes using the inputs. On the other hand, methods that learn an estimate of $P(X|Y)$ and $P(Y)$ are called *generative* methods because they attempt to learn a model of the process that generates the data. One assumes in this case that, in the true data generating process, a class is drawn at random from some prior distribution $P(Y)$ and then, given that class, a set of features is generated

using the distribution $P(X|Y)$. We will give examples of generative models, such as Naïve Bayes classifiers, later in this chapter. More on discriminative vs. generative methods appears in [19].

Sometimes, instead of directly using criteria like the posterior probability or likelihood to try to identify the best model, some other, more convenient criterion is used. One example criterion is classification error on the training set—the fraction of training examples for which the decision tree returns the incorrect class even after learning. The most commonly used decision tree learning algorithms—ID3 [23, 19] and C4.5 [22]—use this criterion. There are some models that can return the posterior probability of an example being in each class. In this case, mean squared error may be a better error criterion. So, for each training example $(\mathbf{x}_i, y_i) \in T$ ($i \in \{1, 2, \ldots, |T|\}$), if $\hat{y}_{i,c}$ is the model's prediction of the posterior probability of class c ($c \in \{1, 2, \ldots, |C|\}$) for training input $\mathbf{x}_i$, while $y_{i,c}$ is 1 if $\mathbf{x}_i$ is an example of class c and 0 otherwise, then the mean squared error criterion is

$$\frac{1}{|T||C|} \sum_{i=1}^{|T|} \sum_{c=1}^{|C|} (y_{i,c} - \hat{y}_{i,c})^2.$$

Clearly, the more confident the classifier is in the correct class, the lower the value of the mean squared error, and similarly the more confident the classifier is in an incorrect class, the higher the value of the mean squared error.

Sometimes the error function is a sum of several criteria instead of just one criterion. Typically one criterion is an error term such as the mean-squared error, and the other one is a criterion designed to reduce the complexity of the model so that it does not *overfit* [26] the training data—fit the training data so well that it fits the noise in the training data in addition to the signal and then perform poorly on the test set. For example, one way to guarantee that decision tree learning will fit the training data at least as well as any other decision tree is to generate a tree that is as deep as needed to make the leaves as homogeneous as possible (maximally-skewed class distributions at each leaf). However, constructing a decision tree in this manner may reduce the feature selection benefit of decision tree learning and instead may include many irrelevant features. This may lead to lower performance on the test set.

We now describe some of the most popular classification learning algorithms.

0.2 Decision Trees and Decision Stumps

A decision tree is a tree structure consisting of nonterminal nodes, leaf nodes, and arcs. An example of a decision tree that may be produced in our example sunspot domain is depicted in figure 1. Each nonterminal node represents a test on a feature value. In the example decision tree, the top node (called the *root* node) tests the value for 'Group Length.' If the group length is either '< 10,' '$10 - 15$,' or '> 15,' then the classification returned is **D**, **E**, or **F**, respectively. If the group length is 'NULL,' then instead of returning a class, an

Decision_Tree_Learning(T,F,C)
  if $T_{i,C}$ is the same for all $i \in \{1, 2, \ldots, |T|\}$,
    return a leaf node labeled $T_{1,C}$.
  else if $|F| = 0$,
    return a leaf node labeled $\text{argmax}_c \sum_{i=1}^{|T|} I(T_{i,C} = c))$.
  else
    $f = Choose\_Best\_Feature(T, F)$
    Set *tree* to be a nonterminal node with test f.
    for each value v of feature f,
      $T_{f=v} = \emptyset$
    for each example $T_i \in T$,
      $v = T_{i,f}$
      Add example T_i to set $T_{f=v}$.
    for each value v of feature f,
      $subtree = Decision\_Tree\_Learning(T_{f=v}, F - f, C)$
      Add a branch to *tree* labeled f with subtree *subtree*.
  return *tree*.

Figure 2: Decision Tree Learning Algorithm. This algorithm takes a training set T, feature set F, and class feature C, as inputs and returns a decision tree. T_i denotes the ith training example, $T_{i,f}$ denotes example i's value for feature f, and $T_{i,C}$ denotes example i's value for the class feature C.

additional feature, 'Magnetic Type,' is checked. If the type is 'Unipolar,' then the 'Penumbra' is checked. If the 'Penumbra' is 'None,' then the class returned is **A**. If the 'Penumbra' is 'Rudimentary,' 'Asymmetrical,' or 'Symmetrical,' the class returned is **H**. If the value of 'Magnetic Type' is 'Bipolar,' then the feature 'Penumbra' is checked as well. This time though, if the value of 'Penumbra' is 'None,' then class **B** is returned, and otherwise class **C** is returned. There may have been other features measured for each example; however, none of them was used.

Decision trees are constructed in a top-down manner which we now describe. One decision tree learning algorithm (ID3) is shown in figure 2. If all the examples are of the same class, then the algorithm just returns a leaf node of that class. If there are no features left with which to construct a nonterminal node, then the algorithm has to return a leaf node. It returns a leaf node labeled with the default class (often chosen to be the class most frequently seen in the training set). If none of these conditions is true, then the algorithm finds the one feature value test that comes closest to splitting the entire training set into groups such that each group only contains examples of one class (we discuss this in more detail in the next paragraph). When such a feature is selected, the training set is split according to that feature. That is, for each value v of the feature, a training set $T_{f=v}$ is constructed such that all the examples in $T_{f=v}$ have value v for the chosen feature. The learning algorithm is called recursively on each of these training sets.

We have yet to describe the function *Choose_Best_Feature* in figure 2. The

feature with the highest *information gain* is commonly used. Information gain is defined as follows:

$$\begin{aligned} Gain(T, f) &= Entropy(T) - \sum_{v \in values(f)} \frac{|T_{f=v}|}{|T|} Entropy(T_{f=v}), \\ Entropy(T) &= \sum_{i \in values(C)} -\frac{|T_{f=v}|}{|T|} log_2 \left(\frac{|T_{f=v}|}{|T|} \right). \end{aligned}$$

Here, $values(f)$ is the set of possible values of feature f if f is a categorical feature. If f is an ordered feature (whether discrete or continuous), then different binary split points are considered (e.g., tests of the form $f < v$ that split the training set into two parts). T is the training set, $T_{f=v}$ is the subset of the training set having value v for feature f, C is the class feature, and p_i is the fraction of the training set having class C_i. The information gain is the entropy of the training set minus the sum of the entropies of the subsets of the training set that result from separating the training set using feature f. The more homogeneous the training set in terms of class values, the lower the entropy. Therefore, information gain measures how much a feature improves the separation of the training set into subsets of homogeneous classes.

A decision stump is a decision tree that is restricted to having only one nonterminal node. That is, the decision tree algorithm is used to find the one feature test that comes closest to splitting all the training data into groups of examples of the same class. One branch is constructed for each value of the feature, and a leaf node is constructed at the end of that branch. The training set is split into groups according to the value of that feature and each group is attached to the appropriate leaf. The class label most frequently seen in each group is the class label assigned to the corresponding leaf. The reader would be justified in wondering when a model so seemingly simplistic as the decision stump would be useful. Decision stumps are useful as part of certain ensemble learning algorithms in which multiple decision stumps are constructed to form a model that is more accurate than any single decision stump and often more accurate than much more complicated models [17].

We chose decision trees as the first machine learning model to describe in this chapter because the decision tree is one of the simplest and most intuitive models. It is rather similar to a set of if-then statements. In many application domains, domain experts need to understand how the model makes its classification decisions to justify its use. Decision trees are among the most transparent and easy-to-understand models, and are able to handle continuous and discrete data. Decision tree learning also includes feature selection—that is, it only uses the features that it needs to separate the training set into groups that are homogeneous or nearly-homogeneous in terms of the class variable. Decision tree learning is relatively fast even on large datasets and can also be altered to run even faster on parallel computers quite easily [27].

Decision trees have drawbacks that lead us to describe other machine learning models in this chapter. The tests within decision trees are based on single

attributes only. Decision tree learning algorithms are also unable to learn arithmetic combinations of features that are effective for classifications. Additionally, they are not designed to represent probabilistic information. The fraction of examples of different classes at each leaf is often used as a representation of the probability, but this is a heuristic measure. The number of examples at a leaf node, or even at nonterminal nodes deep in the tree, may be relatively low sometimes. Therefore, *pruning* is often used to remove such parts of the decision tree. This often helps to alleviate *overfitting*. However, pruning also leads to relatively few features being considered in cases where there is a small amount of available training data. Additionally, decision trees are sensitive to small changes in the training set. This makes them very useful as part of ensemble learning methods where the diversity in decision trees enables their combination to perform better, but may lead to difficulties when used by themselves.

0.3 Naïve Bayes

We earlier discussed Bayes's Rule as a way of motivating generative classifiers. In particular,

$$P(Y = y|X = \mathbf{x}) = \frac{P(Y = y)P(X = \mathbf{x}|Y = y)}{P(X = \mathbf{x})}.$$

Bayes's theorem tells us that to optimally predict the class of an example $\mathbf{x}$, we should predict the class y that maximizes the two expressions in the above equation.

Define F to be the set of features. If all the features are independent given the class, then we can rewrite $P(X = \mathbf{x}|Y = y)$ as $\prod_{f=1}^{|F|} P(X_f = \mathbf{x}_{(f)}|Y = y)$, where $\mathbf{x}_{(f)}$ is the fth feature value of example $\mathbf{x}$. The probabilities $P(Y = y)$ and $P(X_f = \mathbf{x}_{(f)}|Y = y)$ for all classes Y and all possible values of all features X_f are estimated from a training set. For example, $P(X_f = \mathbf{x}_{(f)}|Y = y)$ is the fraction of class-y training examples that have $\mathbf{x}_{(f)}$ as their fth feature value. Estimating $P(X = \mathbf{x})$ is unnecessary because it is the same for all classes; therefore, we ignore it. To classify a new example, we can return the class that maximizes

$$P(X = \mathbf{x}|Y = y) = \prod_{f=1}^{|F|} P(X_1 = \mathbf{x}_{(1)}, \ldots, X_f = \mathbf{x}_{(f)}|Y = y).$$

The Naïve Bayes classifier operates under the naïve assumption that the features are independent given the class, which yields

$$P(X = \mathbf{x}|Y = y) = P(Y = y) \prod_{f=1}^{|F|} P(X_f = \mathbf{x}_{(f)}|Y = y). \tag{4}$$

One simple algorithm for learning Naïve Bayes classifiers is shown in figure 3. For each training example, we just increment the appropriate counts: N is the

Naïve-Bayes-Learning(T,F)
  for each training example $(\mathbf{x}, y) \in T$,
    Increment N
    Increment N_y
    for $f \in \{1, 2, \ldots, |F|\}$
      Increment $N_{y,\mathbf{x}_{(f)}}$
  return $h(\mathbf{x}) = \text{argmax}_{y \in Y} \frac{N_y}{N} \prod_{f=1}^{|F|} \frac{N_{y,\mathbf{x}_{(f)}}}{N_y}$

Figure 3: Naïve Bayes Learning Algorithm. This algorithm takes a training set T and feature set F as inputs and returns a Naïve Bayes classifier h. N is the number of training examples seen so far, N_y is the number of examples in class y, and $N_{y,\mathbf{x}_{(f)}}$ is the number of examples in class y that have $\mathbf{x}_{(f)}$ as their value for feature f.

number of training examples seen so far, N_y is the number of examples in class y, and $N_{y,\mathbf{x}_{(f)}}$ is the number of examples in class y having $\mathbf{x}_{(f)}$ as their value for feature f. $P(Y = y)$ is estimated by $\frac{N_y}{N}$ and, for all classes y and feature values $\mathbf{x}_{(f)}$, $P(X_f = \mathbf{x}_{(f)}|Y = y)$ is estimated by $\frac{N_{y,\mathbf{x}_{(f)}}}{N_y}$. The algorithm returns a classification function that returns, for an example $\mathbf{x}$

$$\underset{y \in C}{\text{argmax}} \frac{N_y}{N} \prod_{f=1}^{|F|} \frac{N_{y,\mathbf{x}_{(f)}}}{N_y}.$$

Every factor in this equation estimates its corresponding factor in equation (4). In spite of the naïvety of Naïve Bayes classifiers, they have performed quite well in many experiments [18].

Besides the naïve assumption of Naïve Bayes classifiers, one can introduce other less restrictive assumptions of conditional independence, such as two groups of features being conditionally independent, to derive other types of Bayesian classifiers. Probabilistic networks [26, 16] include Bayesian classifiers of this type. Noisy-OR classifiers can be seen as a discriminative version of Naïve Bayes classifiers, which are generative. They are equally expressive mathematically—that is, when all the variables are binary, a Naïve Bayes classifier can be transformed into a Noisy-OR classifier, and a Noisy-OR classifier can be transformed into a Naïve Bayes classifier such that, for a given setting of input variables, they give the same class probabilities. Surprisingly, even though the Noisy-OR classifier is simpler, being a discriminative model, it is more difficult to learn (that is, more training examples are needed to learn the parameters accurately) than the Naïve Bayes classifier, which explains why Noisy-OR classifiers have received little attention by themselves. See [9] for a detailed comparison of Naïve Bayes and Noisy-OR classifiers.

Naïve Bayes classifiers have several advantages that make them quite popular. They are very easy and fast to train, are robust to missing values, and have a clear probabilistic semantics. In spite of their rather strong assumption

that the input features are independent given the class, they perform quite well in situations where this assumption is not true, and there is some theoretical evidence to justify this [14]. Another advantage is that the probability distributions that are estimated involve only one variable each. This alleviates the *curse of dimensionality*, which is the problem that, as the dimensionality of the relevant space increases, the training set required to fill the space densely enough to derive accurate models grows exponentially. Naïve Bayes classifiers are also relatively insensitive to small variations in the training set.

The simplicity of Naïve Bayes models also has the drawback that it cannot represent situations in which features interact to classify data. For example, with decision trees, we saw that certain features are only used when other features have particular values. Such interactions cannot be represented in a Naïve Bayes classifier. Additionally, the strong performance of Naïve Bayes even with violations of the assumption of feature independence comes about because correct classification only requires the correct class's posterior probability to be higher than the others—the probabilities do not actually have to be correct, and the rank order of the probabilities of the incorrect classes does not have to be correct. Naïve Bayes classifiers may not necessarily give the correct probabilities.

0.4 Neural Networks

Artificial neural networks have a structure and function that is inspired by biological neural networks [5]. The multilayer perceptron is the most common neural network representation. It is often depicted as a directed graph consisting of nodes and arcs—an example is shown in figure 4. Each column of nodes is a *layer*. The leftmost layer is the *input layer*. The inputs or features of an example to be classified are entered into the input layer. The second layer is the *hidden layer*. The third layer is the *output layer*, which, in classification problems, typically consists of as many outputs as classes. Information flows from the input layer to the hidden layer to the output layer via a set of arcs. Note that the nodes within a layer are not directly connected. In our example, every node in one layer is connected to every node in the next layer, but this is not required in general. Also, a neural network can have more or less than one hidden layer and can have any number of nodes in each hidden layer.

Each non-input node, its incoming arcs, and its single outgoing arc constitute a *neuron*, which is the basic computational element of a neural network. Each incoming arc multiplies the value coming from its origin node by the weight assigned to that arc and sends the result to the destination node. The destination node adds the values presented to it by all the incoming arcs, transforms it with a nonlinear activation function (to be described later), and then sends the result along the outgoing arc. For example, the output of a hidden node z_j

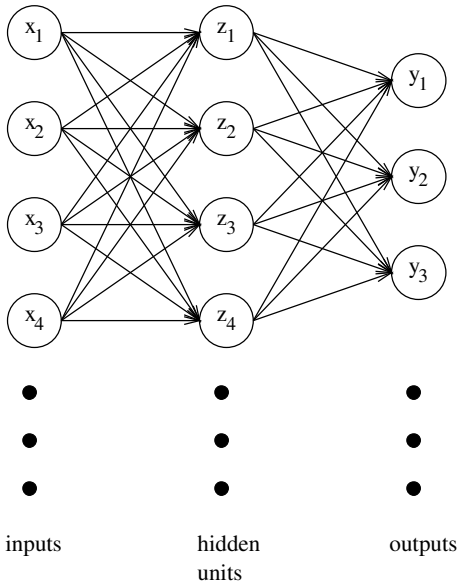

Figure 4: An example of a multilayer feedforward perceptron.

in our example neural network is

$$z_j = g\left(\sum_{i=1}^{|F|} w_{i,j}^{(1)} \mathbf{x}_{(f)}\right)$$

where $|F|$ is the number of input units, which is the same as the number of input features; $w_{i,j}^{(k)}$ is the weight on the arc in the kth layer of arcs that goes from unit i in the kth layer of nodes to unit j in the next layer (so $w_{i,j}^{(1)}$ is the weight on the arc that goes from input unit i to hidden unit j); and g is a nonlinear activation function. A commonly used activation function, inspired by observations of biological neural networks, is the sigmoid function:

$$g(a) \equiv \frac{1}{1 + exp(-a)}.$$

The output of an output node y_j is

$$y_j = g\left(\sum_{i=1}^{Z} w_{i,j}^{(2)} z_i\right)$$

where Z is the number of hidden units. The outputs are clearly nonlinear functions of the inputs. Neural networks used for classification problems typically have one output per class. The example neural network depicted in figure 4 is of this type. The outputs lie in the range $[0, 1]$. Each output value is a measure

of the network's confidence that the example presented to it is a member of that output's corresponding class. Therefore, the class corresponding to the highest output value is returned as the prediction.

The most widely used method for setting the weights in a neural network is the backpropagation algorithm [7, 19, 25]. For each of the training examples in the training set T, its inputs are presented to the input layer of the network and the predicted outputs are calculated. The difference between each predicted output and the corresponding target output is calculated. The total error of the network is

$$E = \frac{1}{2}\sum_{t=1}^{|T|}(y_i - \hat{y}_i)^2 \tag{5}$$

where y_i and $\hat{y}_i$ are the true and predicted outputs, respectively, for the ith training example. In classification, neural networks are normally set up to have one output per class, so that y_i and $\hat{y}_i$ become vectors $\mathbf{y}_i$ and $\hat{\mathbf{y}}_i$. In the training set, $\mathbf{y}_{i,c} = 1$ if the ith training example is of class c and $\mathbf{y}_{i,c} = 0$ otherwise. By training the neural network with the mean squared error criterion (equation 5), the network attempts to estimate the posterior probability $P(Y|X)$ for every class Y. A separate value of E can be calculated for each class output. We can write E in terms of the parameters in the network as follows:

$$\begin{aligned} E &= \frac{1}{2}\sum_{n=1}^{N}\left(y_n - g\left(\sum_{j=1}^{Z} w_{i,j}^{(2)} z_i\right)\right)^2, \\ &= \frac{1}{2}\sum_{n=1}^{N}\left(y_n - g\left(\sum_{j=1}^{Z} w_{i,j}^{(2)} g\left(\sum_{i=1}^{|A|} w_{i,j}^{(1)}\mathbf{x}_i\right)\right)\right)^2. \end{aligned}$$

In order to adjust the weights to reduce the error, we calculate the derivative of E with respect to each weight and change the weight accordingly. The derivatives are

$$\begin{aligned} \frac{\partial E}{\partial w_{i,j}^{(2)}} &= -\sum_{n=1}^{N}\left(y_n - g\left(\sum_{j=1}^{Z} w_{i,j}^{(2)} z_i\right)\right) g'\left(\sum_{j=1}^{Z} w_{i,j}^{(2)} z_i\right) z_i \\ \frac{\partial E}{\partial w_{i,j}^{(1)}} &= \sum_{n=1}^{N}\left(y_n - g\left(\sum_{j=1}^{Z} w_{i,j}^{(2)} g\left(\sum_{i=1}^{|A|} w_{i,j}^{(1)}\mathbf{x}_i\right)\right)\right) \\ &\quad g'\left(\sum_{j=1}^{Z} w_{i,j}^{(2)} g\left(\sum_{i=1}^{|A|} w_{i,j}^{(1)}\mathbf{x}_i\right)\right) g'\left(\sum_{i=1}^{|A|} w_{i,j}^{(1)}\mathbf{x}_i\right)\mathbf{x}_i \end{aligned}$$

Note that many factors in the derivatives appear more than once, so they can be computed just once and re-used. Also, if g is a sigmoid function, then

$g' = g(1-g)$, so these derivatives can be calculated quickly. The weights on the arcs of the networks are adjusted according to these derivatives so that if the training example is presented to the network again, then the error would be less. The learning algorithm typically cycles through the training set many times—each cycle is called an *epoch* in the neural network literature.

Neural networks have performed well in a variety of domains for nearly fifty years. They are able to represent complicated interactions among features when necessary to derive a classification. They are also *universal approximators*—given a single hidden layer and an arbitrary number of hidden units, they can approximate any continuous function, and given two hidden layers and an arbitrary number of hidden units, they can approximate any function to arbitrarily high accuracy. The nonlinear activation function enables the entire network to represent a function that is more expressive than a linear function, and gives us this universal approximator property [5].

The universal approximator property of neural networks is nice in theory; however, in practice, because neural network learning is computationally intensive, reaching a small error can take excessive time and can lead to overfitting [5]. Determining the best number of hidden units is more art than science. Neural networks are learned using gradient descent methods, which use partial derivatives of the error with respect to model parameters as shown above to adjust the parameters to improve the accuracy. However, gradient descent methods applied to nonlinear models such as neural networks are guaranteed to reach solutions that are locally optimal but not necessarily globally optimal [5]. Additionally, the error as a function of the neural network's weights seems to be relatively complicated, because starting the learning from different initial weights tends to lead to very different final weights and the resulting networks have different errors in spite of learning with the same training set. This indicates that there are many locally optimal solutions. Neural networks are also nearly impossible for domain experts and even machine learning experts to interpret for the purpose of understanding how they arrive at their classifications.

0.5 Support Vector Machine (SVM)

Support Vector Machines were developed by Vapnik in 1979 (see [8] for a tutorial). SVMs are learned using a statistical learning method based on Structural Risk Minimization (SRM). In SRM, a set of classification functions that classify a set of data are chosen in such a way that minimizing the training error (what Vapnik refers to as the empirical risk) yields the minimum upper bound on the test error (what Vapnik refers to as the actual risk). The upper bound on the test error is a guarantee that the test error will be equal to or lower than the bound; therefore, minimizing the upper bound is the next best thing to minimizing the test error itself.

The simplest example of a Support Vector Machine is a linear hyperplane trained on data that is perfectly separable as shown on the left side of figure 5. Given a set of input vectors, $\mathbf{x}_i \in \mathbf{R^d}$, and labels, $y_i \in \{-1, 1\}$, SVM finds a hyperplane described by its normal vector, $\mathbf{w}$, and distance from the origin, b,

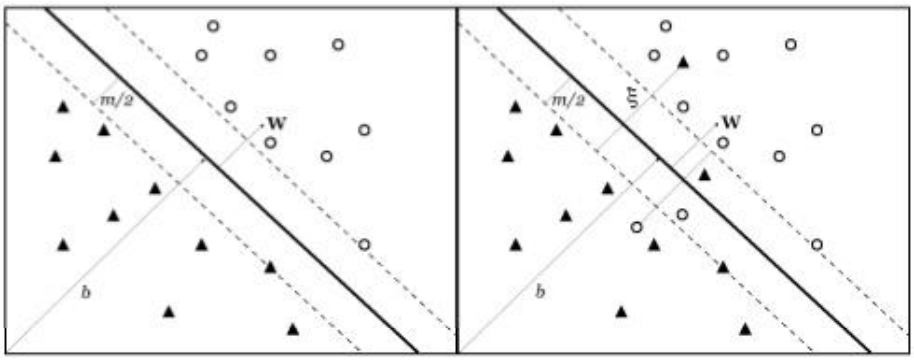

Figure 5: Optimum Hyperplane for Separable and Non-Separable Case

that divides the data perfectly and is equidistant from at least one point in each class that is closest to the hyperplane. The shortest distance between any point and the hyperplane is called the *margin*, and the algorithm works to maximize this margin in order to minimize the chance that newer data points of a given class, which are likely to be close to the existing points of the same class, will end up on the wrong side of the hyperplane [8]. In figure 5, $m/2$ is the margin.[2] This hyperplane is a decision boundary and the classification of an unknown input vector is determined by the sign of the vector operation

$$\mathbf{x}_i \cdot \mathbf{w} - b = d \tag{6}$$

If $d \geq 0$ ($d < 0$) then the input is predicted to be in the class $y = +1$ ($y = -1$).

Learning an SVM requires using the training data to find the best value of $\mathbf{w}$. The SVM is constructed by solving the optimization problem

$$\begin{aligned} & min \parallel \mathbf{w} \parallel \\ \text{subject to} \quad & \mathbf{x}_i \cdot \mathbf{w} + b \geq 1 \quad \text{for } y_i = +1 \\ & \mathbf{x}_i \cdot \mathbf{w} + b \leq -1 \quad \text{for } y_i = -1, \end{aligned}$$

where the last two equations can be combined into one as follows:

$$y_i(\mathbf{x}_i \cdot \mathbf{w} + b) - 1 \geq 0 \; \forall i \in \{1, 2, \ldots, |T|\}.$$

The objective function is chosen because we would like to maximize the margin, which turns out to be $2/ \parallel \mathbf{w} \parallel$, so this is equivalent to minimizing $\parallel \mathbf{w} \parallel$.

If the data are not perfectly separable this method can be adapted to compensate for instances that occur on the wrong side of the hyperplane. In that case slack variables, ξ_i (one for each training example), are introduced that measure the error of misclassified instances of the training data—in particular,

[2]For the case where the number of classes C is greater than two, typically the problem is split into $|C|$ two-class problems where, for each class $c \in C$, the corresponding class-c SVM learns to predict whether the example is in class c or not.

the slacks measure how far on the wrong side of the hyperplane these misclassified instances are (see the right half of figure 5). SVMs find a hyperplane that best separates the data and minimizes the sum of the errors ξ_i by changing the optimization problem to the following.

$$min \parallel \mathbf{w} \parallel + B \sum_{i=1}^{|T|} \xi_i \tag{7}$$

$$\text{subject to} \quad \mathbf{x}_i \cdot \mathbf{w} + b \geq 1 - \xi_i \quad \text{for } y_i = +1, \tag{8}$$

$$\mathbf{x}_i \cdot \mathbf{w} + b \leq -1 + \xi_i \quad \text{for } y_i = -1, \tag{9}$$

$$\xi_i \geq 0 \quad \forall i \in \{1, 2, \ldots, |T|\}, \tag{10}$$

where B is a user-defined weight on the slack variables. If B is large, the learning algorithm puts a large penalty on errors and will devise a more complicated model. If B is small, then the classifier is simpler but may have more errors on the training set. If the datasets are not balanced it is sometimes necessary to give the errors of one class more weight than the other. Additional parameters, μ_y (one for each class), can be added to weigh one class error over the other by replacing the objective in equation (7) with

$$min \parallel \mathbf{w} \parallel + B \sum_{i=1}^{|T|} \mu_{y_i} \xi_i$$

We can write the Wolfe dual [8] of the convex optimization problem shown in equation (7) as follows:

$$max \sum_{i=1}^{|T|} \alpha_i - \frac{1}{2} \sum_{i=1}^{|T|} \sum_{j=1}^{|T|} \alpha_i \alpha_j y_i y_j (\mathbf{x}_i \cdot \mathbf{x}_j) \tag{11}$$

$$\text{subject to } 0 \leq \alpha_i \leq B, \tag{12}$$

$$\sum_{i=1}^{|T|} \alpha_i y_i = 0. \tag{13}$$

This decision hyperplane can only be linear which is not suitable for many types of data. To overcome this problem a function can be used to map the data into a higher or infinite dimensional space and run SVM in this new space.

$$\Phi : \mathbf{R}^d \mapsto \mathcal{H}$$

Because of the "kernel trick," $\mathcal{H}$ may have infinite dimension while still making the learning algorithm practical. That is, one never works directly with Φ but rather with a kernel function K such that

$$K(\mathbf{x}_i, \mathbf{x}_j) = \Phi(\mathbf{x}_i) \cdot \Phi(\mathbf{x}_j),$$

so that the dual formulation shown in equation (11) becomes

$$max \sum_{i=1}^{|T|} \alpha_i - \frac{1}{2} \sum_{i=1}^{|T|} \sum_{j=1}^{|T|} \alpha_i \alpha_j y_i y_j K(\mathbf{x}_i, \mathbf{x}_j)$$

$$\text{subject to } 0 \leq \alpha_i \leq B,$$

$$\sum_{i=1}^{|T|} \alpha_i y_i = 0.$$

Even though Φ may map to a very high or infinite dimensional space, it only shows up in the form of dot products, which are always scalars. This allows us to utilize the expressive power of a high or infinite dimensional space while not paying a very high (or infinite!) price for it computationally. There are many possible kernel functions available and new kernels can be built from the data itself. The kernels only need to meet Mercer's Conditions [10] to be used in SVM. One example kernel function is a radial basis function,

$$K(\mathbf{x}_i, \mathbf{x}_j) = e^{-\gamma \|\mathbf{x}_i - \mathbf{x}_j\|^2}$$

where γ is a parameter given by the user.

For a new example $\mathbf{x}$, the predicted class is the sign of

$$f(\mathbf{x}) = \sum_{i=1}^{|T|} \alpha_i y_i K(\mathbf{x}_i, \mathbf{x}) + b.$$

However, this can be simplified. The summation actually does not need to be calculated over all the training examples, but rather only those training examples whose distance from the hyperplane is exactly half the margin and those which are on the wrong side of the hyperplane. These are referred to as *support vectors*, and are the only training points having $\alpha_i \neq 0$. If V is the subset of T containing the support vectors, then the above classification function can be simplified to

$$f(\mathbf{x}) = \sum_{(\mathbf{s},y) \in V} \alpha_i y K(\mathbf{s}, \mathbf{x}) + b.$$

As mentioned earlier, the sign of $f(\mathbf{x})$ indicates the predicted class. $|f(\mathbf{x})|$ indicates how confident the SVM is in its class. $|f(\mathbf{x})|$ does not correspond directly to a probability, but can nevertheless can be used as a measure of confidence.

SVMs are generated by solving a convex optimization problem, which means that the solution is guaranteed to be optimal for that training set, even though SVMs can be nonlinear in the original data space. They are also usable in problems such as text mining where a very large number of input features is present [21]. They have been shown to perform very well experimentally in many domains. SVMs are applicable to many domains because kernels over many different types of inputs—discrete, continuous, graphs, sequences, etc.,—have been derived. Multiple kernels can even be used at the same time [1, 12, 13]. These properties have led SVMs to be one of the most popular machine learning models in use today among machine learning researchers and practitioners, as can be seen in the many papers on SVMs in machine learning conferences over the last ten years.

SVMs have the disadvantage, just like neural networks, of being impenetrable by domain experts wishing to understand how they arrive at their classifications. Learning SVMs is also computationally intensive, although there

have been substantial efforts to reduce their running time (such as [11]). They are also designed to solve two-class problems. As mentioned earlier, they can be set up to solve $|C|$-class problems by setting up $|C|$ SVMs, each of which solves a two-class problem. However, this means $|C|$ times the computation of a two-class problem. Also, if the set of classes is mutually exclusive and if more than one SVM predicts that the example is in its corresponding class, then it is unclear how to determine which class is the correct one.

0.6 Further Reading

We gave several references throughout the chapter for more information on the machine learning algorithms discussed here. Another reference for classification algorithms in general is [15]. There have also been several attempts at applying classification methods to astronomy data. Classification and other forms of analysis on data representing different solar phenomena are described in [4]. Classification of stars and galaxies in various sky surveys such as the Sloan Digital Sky Survey are described in [3, 6]. See [24] for a description of the use of classification methods for matching objects recorded in different catalogues. Classification of sunspot images obtained by processing some NASA satellite images is described in [20]. A recent, general survey of the use of data mining and machine learning in astronomy is [2].

References

[1] F.R. Bach, G.R.G. Lanckriet, and M.I. Jordan. Multiple kernel learning, conic duality, and the smo algorithm. In *International Conference on Machine Learning*, 2004.

[2] Nicholas M. Ball and Robert J. Brunner. Data mining and machine learning in astronomy. *International Journal of Modern Physics D*, 19(7):1049–1106, 2009.

[3] N.M. Ball, R.J. Brunner, and A.D. Myers. Robust machine learning applied to terascale astronomical datasets. In *Astronomical Data Analysis Software Systems (ADASS) XVII, ASP Confererence Series, eds. Argyle R.W., Bunclark P.S., Lewis J.R*, pages 201–204, 2008.

[4] Juan M. Banda and Rafal Anrgyk. Usage of dissimilarity measures and multidimensional scaling for large scale solar data analysis. In *Proceedings of the 2010 Conference on Intelligent Data Understanding*. National Aeronautics and Space Administration, 2010.

[5] C. M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, New York, 1995.

[6] Kirk D. Borne. A machine learning classification broker for petascale mining of large-scale astronomy sky survey databases. In *National Science*

Foundation Symposium on Next Generation of Data Mining and Cyber-Enabled Discovery for Innovation (NGDM-07). National Science Foundation, 2007.

[7] A.E. Bryson and Y.-C. Ho. *Applied Optimal Control*. Blaisdell Publishing Co., 1969.

[8] C. J. C. Burges. A tutorial on support vector machines for pattern recognition. *DMKD*, 2:121–167, 1998.

[9] David L. Chen. Sisterhood of classifiers: A comparative study of naive bayes and noisy-or networks. Master's thesis, Department of Computer Science, University of California, Los Angeles, 2006.

[10] N. Cristianini and J. Shawe-Taylor. *An Introduction to Support Vector Machines and other Kernel-based learning methods*. Cambridge University Press, 2000.

[11] Santanu Das, Kanishka Bhaduri, Nikunj Oza, and Ashok Srivastava. nuanomica: A fast support vector based anomaly detection technique. In *Proceedings of the IEEE International Conference on Data Mining (ICDM)*. Institute for Electrical and Electronics Engineers, 2009.

[12] Santanu Das, Bryan Matthews, Kanishka Bhaduri, Nikunj Oza, and Ashok Srivastava. Detecting anomalies in multivariate data sets with switching sequences and continuous streams. In *NIPS 2009 Workshop: Understanding Multiple Kernel Learning Methods*, 2009.

[13] Santanu Das, Bryan Matthews, Ashok Srivastava, and Nikunj C. Oza. Multiple kernel learning for heterogeneous anomaly detection: Algorithm and aviation safety case study. In *The Sixteenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM Press, 2010.

[14] Pedro Domingos and Michael Pazzani. On the optimality of the simple bayesian classifier under zero-one loss. *Machine Learning*, 29:103–130, 1997.

[15] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification*. Wiley, New York, NY, second edition, 2001.

[16] Daphne Koller and Nir Friedman. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009.

[17] Ludmila I. Kuncheva. *Combining Pattern Classifiers*. Wiley, 2004.

[18] D. Michie, D.J. Spiegelhalter, and C.C. Taylor. *Machine Learning, Neural and Statistical Classification (edited collection)*. Ellis Horwood, New York, 1994.

[19] Tom Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[20] Trung Thanh Nguyen, Claire P. Willis, Derek J. Paddon, Sinh Hoa Nguyen, and Hung Son Nguyen. Learning sunspot classification. *Fundamenta Informaticae*, 72(1):295–309, 2006.

[21] Nikunj C. Oza, J. Patrick Castle, and John Stutz. Classification of aeronautics system health and safety documents. *IEEE Transactions on Systems, Man, and Cybernetics, Part C*, 39(6):670–680, 2009.

[22] J.R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufman, San Mateo, California, 1992.

[23] Ross Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

[24] David Rohde, Michael Drinkwater, Marcus Gallagher, Tom Downs, and Marianne Doyle. Machine learning for matching astronomy catalogues. In *Intelligent Data Engineering and Automated Learning (IDEAL)*, pages 702–707. Springer-Verlag, 2004.

[25] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. Bradford Books/MIT Press, Cambridge, Mass, 1986.

[26] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (Third Edition)*. Pearson Education, 2010.

[27] A. Srivastava, E. Han, V. Kumar, and V. Singh. Parallel formulations of decision-tree classification algorithms. *High Performance Data Mining*, pages 237–261, 2002.