9 Classification

In classification, we are trying to learn a map from an input space to some finite output space. In the simplest case we simply detect whether or not the input has some property or not. For example, we might want to determine whether or not an email is spam, or whether an image contains a face. A task in the health care field is to determine, given a set of observed symptoms, whether or not a person has a particular disease. Such detection tasks are *binary classification* problems.

In *multi-class classification* problems we are interested in determining to which of multiple categories the input belongs. For example, given a recorded voice signal we might want to recognize the identity of a speaker (perhaps from a set of people whose voice properties are given in advance). Another well studied classification task is optical character recognition, i.e., the recognition of letters or numbers from images of handwritten or printed characters.

The input $\mathbf{x}$ might be a vector of real numbers, or a discrete feature vector. For binary classification problems the output y might be an element of the set $\{-1, 1\}$. For a multi-class problem with K categories the output might be an integer in $\{1, \ldots, K\}$.

The general goal of classification is to learn a *decision boundary*, often specified as the level set of a function, e.g., $a(\mathbf{x}) = 0$. The purpose of the decision boundary is to specify the regions of the input space that correspond to each class. For binary classification the decision boundary is the surface in the feature space that separates inputs into two classes; points $\mathbf{x}$ for which $a(\mathbf{x}) < 0$ are deemed to be in one class, while points for which $a(\mathbf{x}) > 0$ are in the other class. Points on the decision boundary, $a(\mathbf{x}) = 0$, are those inputs for which the two classes are equally probable.

In this chapter we introduce several basic methods for classification. We focus mainly on binary classification problems, for which the methods are conceptually straightforward, easy to implement, and often effective. Some of the methods generalize straightforwardly to multi-class problems. In subsequent chapters we discuss some of the more sophisticated methods that might be needed for more challenging problems.

9.1 Classification by Regression

One tempting way to perform classification is with least-squares rgression. That is, we could treat the class labels $y \in \{-1, 1\}$ as real numbers, and estimate a vector of weights, $\mathbf{w}$, by minimizing

$$E(\mathbf{w}) = \sum_i (y_i - \mathbf{x}_i^T \mathbf{w})^2 \,, \tag{9.1}$$

for a given set of labeled training data points $\{\mathbf{x}_i, y_i\}$. Given the optimal regression weights, one could then perform regression on subsequent test inputs and use the sign of the output to determine the output class, i.e., $\text{sgn}[\mathbf{w}^T \mathbf{x}]$, where

$$\text{sgn}[z] = \begin{cases} -1 & z \leq 0 \\ 1 & z > 0 \end{cases} \tag{9.2}$$

In simple cases this might perform well, but in general it does not. This is because the objective function in linear regression measures the distance from the modeled class labels (which can be any real number) to the true class labels, which may not provide an accurate measure of how well

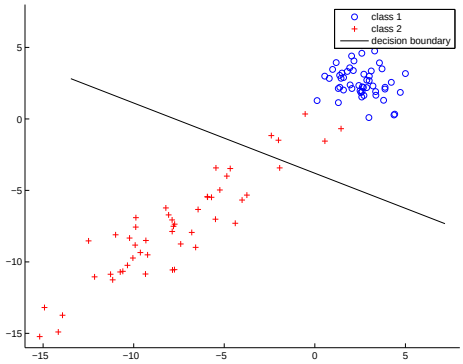

Figure 9.1: Here, one class has a small isotropic covariance. The other is large and anistropic. Although the data are linearly separable (i.e., a line exists that correctly separates the training samples), the LS regression fails to separate the classes. Rather, the LS regression decision boundary produces 5 incorrectly classified points.

the model has classified the data. For example, a linear regression model will tend to produce predicted labels that lie outside the range of the class labels for "extreme" members of a given class (e.g. 5 when the class label is 1), causing the error to be measured as high even when the classification (given, say, by the sign of the predicted label) is correct. In such a case the decision boundary may be shifted towards such an extreme case, potentially reducing the number of correct classifications made by the model. Figure 9.1 depicts a simple example of this behaviour.

The problem arises from the fact that the constraint that $y \in (-1, 1)$ is not built-in to the model (the regression algorithm knows nothing about it), and so wastes considerable representational power trying to reproduce this effect. In other words, the decision boundary for the simple LS classifier in (9.1) is the hyperplane $\mathbf{x}_i^T \mathbf{w} = 0$. The classifier only cares about the sign of $\mathbf{x}_i^T \mathbf{w}$, but not its magnitude. Nevertheless, the LS regression optimization is also attempting to find a way to keep the its magnitude of $\mathbf{x}_i^T \mathbf{w}$ close to 1, which is not important for the classification. We should instead formulate the problem so that this constraint is built into the model.

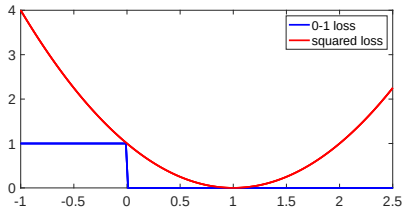

Figure 9.2: These two graphs show the 0-1 classification loss (blue) and the squared loss (red) on the y axis, as a function of $y\, f(\mathbf{x})$ on the x axis.

Another way to understand the problem with this approach is through the loss function being used. A common 'ideal' loss for classification is known as the 0-1 loss, given by $\text{sgn}[\, y\, f(\mathbf{x})\,]$ for $y \in \{-1, 1\}$ and a classifier $\text{sgn}[f(\mathbf{x})]$. In our example here, $f(\mathbf{x}) = \mathbf{x}^T \mathbf{w}$. The loss in this case is 0 when the training sample is correctly classified, and 1 otherwise. So the empirical loss over the entire training set is simply the number of incorrectly classified samples.

But the loss used to find the weights is the squared loss. With a little work you can show that the squared loss is equal to $(y\,f(\mathbf{x})-1)^2$. One can see from the plot of the 0-1 loss and the squared loss in Figure 9.2 that the squared loss penalizes the classifer for being wrong (i.e., negative $yf(\mathbf{x})$) with a large value of f, but it also penalizes large values of f even when the classifier is correct.

9.2 k-Nearest Neighbors Classification

We can apply the k-NN idea used for regression several chapters earlier (see Sec. **??**) to classification as well. For class labels $\{-1, 1\}$, the classifier is:

$$y_{new} = \text{sgn}\left[\sum_{i \in N_k(\mathbf{x})} y_i\right] , \tag{9.3}$$

where $N_k(\mathbf{x})$ is the set containing the k nearest neighbors of $\mathbf{x}$, usually determined by Euclidean distance. Rather than a simple sum of nearest neighbors we might also take a weighted average of the k nearest neighbors:

$$y = \text{sgn}\left[\sum_{i \in N_K(\mathbf{x})} w(\mathbf{x}_i)\, y_i\right] , \quad w(\mathbf{x}_i) = e^{-||\mathbf{x}_i - \mathbf{x}||^2/2\sigma^2} \tag{9.4}$$

where σ^2 is an additional parameter to the algorithm. An alternative to the expoential weight function is simply inverse Euclidean distance.

For k-NN the decision boundary will be a collections of hyperplane patches that are perpendicular bisectors of pairs of points drawn from the two classes. As illustrated in Figure 9.3, for 2D inputs this is a set of bisecting line segments. Figure 9.3, shows a simple case but it is not hard to imagine that the decision surfaces can get very complex, e.g., if a point from class 1 lies somewhere in the middle of the points from class 2. By increasing the number of nearest neighbours (i.e., k) one is effectively smoothing the decision boundary, hopefully thereby improving generalization.

Although k-NN classifiers are easy to understand and use they have significant limitations. Among them, one must retain all training data, so storage and search for NNs can be very expensive, especially with high-dimensional features.

9.3 Decision Trees

Decision trees (and random forests) are computationally straightforward and scale well to very large datasets, and hence often used in data mining. (In the Microsoft Kinect, for example, multiple decision trees were learned on millions of training exemplars to enable real-time human pose estimation from an RGB-D camera.) Because their decision boundaries are specified in terms of relatively simple tests on the input features, decision trees also provide some explanatory power, which is often desirable. They can be applied to both classification and regression tasks, and to real-valued or discrete features. That said, in what follows we focus on binary decision trees and real-valued features for classification.

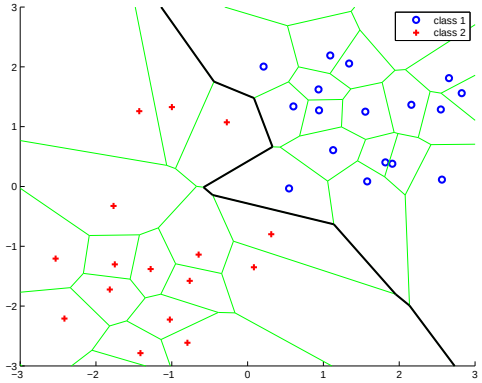

Figure 9.3: For two classes and planar inputs, the decision boundary for a 1-NN classifiifier (the bold black curve) is a subset of the perpendicular bisecting line segments (green) between pairs of neighbouring points (obtained with a Voronoi tesselation).

A decision tree comprises internal (split) nodes and leaf (terminal) nodes. For a binary tree the number of leaf nodes is always one more than the number of internal nodes. Given a d-dimensional feature vector, $\mathbf{x}$, the nodes of the tree apply a sequence of tests that determine the leaf node to which the point is assigned (by traversing the tree from the root). Each internal node, indexed by $j \in \{1, \ldots, m\}$, performs a binary test, denoted $t_j(\mathbf{x}) : \mathbb{R}^d \to \{-1, +1\}$. If $t_j(\mathbf{x})$ evaluates to -1, then $\mathbf{x}$ is directed to the left child of node j. Otherwise, $\mathbf{x}$ is directed to the right child. And so on down the tree.

Figure 9.3 shows a set of 2D features points, each of which is the measured height and width of an orange or a lemon. Suppose we want to classify whether something is a lemon or an orange based on these two measurements. The decision tree in Fig. 9.3 (left) uses three tests (internal nodes) to determine which of four leaf nodes a given 2D input point is assigned to. As such, one can partition the feature space according to which leaf node each point is assigned to. In effect, these partitions define the decision boundaries used by the tree (depicted in Fig. 9.3 (right)).

The class associated with a given leaf node is determined by the labels associated with the training samples assigned to that leaf node. One could let them vote, so the class having the largest number of training points in a given leaf is the class predicted for any test point that reaches that leaf.

Alternatively, one can estimate a (multinomial) probability distribution over the classes for each leaf, i.e., a conditional distribution over the classes. The class with the greatest number of training points will of course have the highest predictive probability. If another class also has a large number of points, this will be reflected in the predicted distribution over classes. More precisely, each leaf node, indexed by $j \in \{0, \ldots, m\}$, specifies a conditional probability distribution over class labels, $y \in \{1, \ldots, K\}$, denoted $P(y = c \mid j)$. With K classes we need to estimate the probability of each class (and of course they must sum to one). We can, for example, let $P(y = c \mid j) = N_{j,c}/N_j$ where $N_{j,c}$ is the number of training points in leaf j with label c, and N_j is the total number of training points assigned to leaf j. In total, the parameters of a decision tree include the tests at the internal nodes, and the probabilities at each of the leaf nodes.

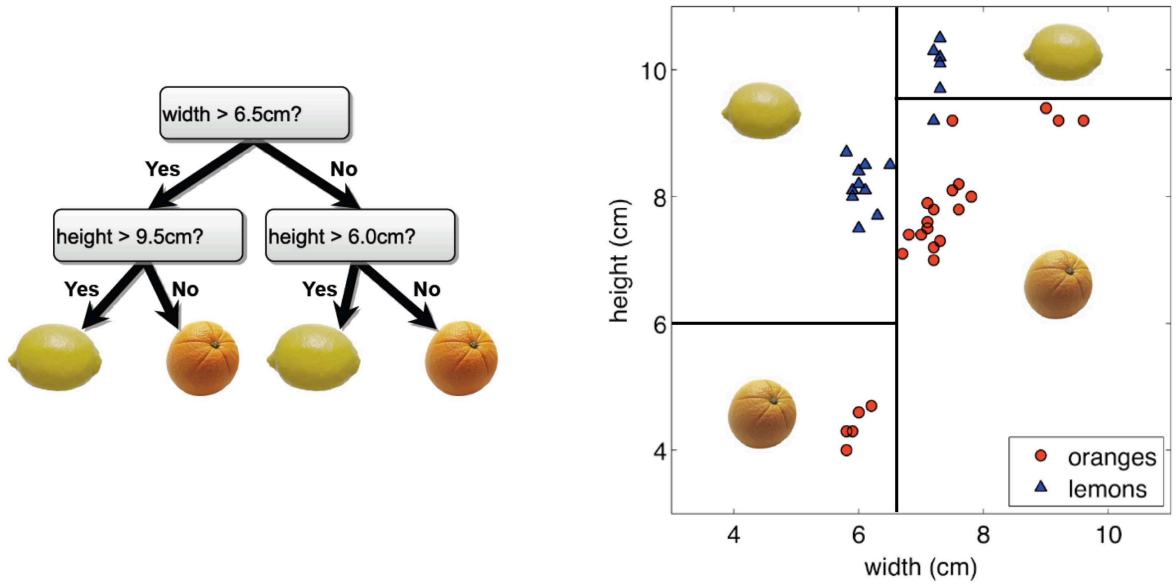

Figure 9.4: Each 2D data point is the measured height and width of a lemon (triangle) or orange (circle).

9.3.1 Learning

Learning the optimal decision tree is known to be NP-hard. As a consequence, learning is often accomplished with a greedy algorithm, in which the tree is grown from the root, recursively, one node at a time. In each step of learning, one is given a set of data points. At the root this will be the entire training set. The data received by the left and right children of the root depend on the split test learned at the root, and so on down the tree.

Given a set of data points assigned to node j, denoted $\mathcal{D}_j$, learning requires that we first decide whether the node should be a leaf (terminal) node, or an internal (split) node. If all points in $\mathcal{D}_j$ have the same label, or there are very few points, it is reasonable to terminate the recursive learning along that branch of the tree. Otherwise the node should be internal. In that case, node j gets two children and we have to determine a test that partitions the data into left and right subsets, $\mathcal{D}_L$ and $\mathcal{D}_R$, which are of course passed to the children for subsequent learning.

The eventual goal is (ideally) to find test conditions so that the conditional probability distributions of the child nodes are simpler, in some sense, than the parent. In essence, these split tests are simple classifiers; one would like to partition the data for the children such that each child only has non-zero probability for a single class. Such split tests at internal nodes are often called *decision stumps*.

A common measure of the quality of a split test, called *information gain*, is based on the information theoretic notion of entropy developed in communications theory. Often denoted H, the entropy of a distribution $p(y)$ over K discrete events (e.g., class labels) is defined by

$$H \;=\; -\sum_{c=1}^{K} p_c \log p_c \tag{9.5}$$

where $p_c \equiv P(y = c)$. Entropy is a measure of the uncertainty in a random variable, and plays a central role in optimal coding theory. Notice that H is maximal when all classes have the same probability. Ideally the leaf nodes very few or only one class one class with non-zero probability, in which case the entropy (i.e., uncertainty) is very small or zero.

Information gain is the reduction in entropy produced by partitioning the data according to a given split test. Given a hypothetical test, t_j, for the N_j training points at node j into child subsets, $\mathcal{D}_L$ and $\mathcal{D}_R$, with N_L and N_R points respectively, information gain is defined as

$$IG(\mathcal{D}_j, t_j) \;=\; H(\mathcal{D}_j) - \frac{N_L}{N_j} H(\mathcal{D}_L) - \frac{N_R}{N_j} H(\mathcal{D}_R) \tag{9.6}$$

where $H(\mathcal{D}$ is the entropy of the conditional probability distribution over classes associated with data $\mathcal{D}$. This is the reduction of uncertainty, or equivalently a measure of how uniformly distributed the data in the leaves are after the split. (This measure is often biased toward nodes with more data, and as such it is also common to use a normalized measure of information gain where IG in (9.6) is divided by $\frac{N_L}{N_j} H(\mathcal{D}_L) + \frac{N_R}{N_j} H(\mathcal{D}_R)$.)

You may have also realized that information gain is simply mutual information. IG in (9.6) is the entropy over the class label for data at node j, minus the conditional entropy over the class labels given the split. To see this, note that $\frac{N_L}{N_j}$ and $\frac{N_R}{N_j}$ are simply the probabilities of taking the left and right branches, respectively. $H(\mathcal{D}_L)$ is the entropy over the class labels conditioned on taking the left branch; and similarly for $H(\mathcal{D}_R)$. The sum of these two branch entropies, weighted by the probability of taking one branch of the other, is the conditional entropy for the class labels given the split (with some abuse of notation):

$$IG(\mathcal{D}_j, t_j) \;=\; H(\mathcal{D}_j) - H(\mathcal{D}_j \,|\, t_j)\,. \tag{9.7}$$

Finding a good split test can be expensive, because the number of possible split tests to explore is usually prohibitive. Most learning algorithms only consider univariate (axis-aligned) split functions. Univariate split tests compare just one feature dimension with a threshold. For instance if there are N_j points at node j, each of which is a d-dimensional feature vector, and one considers only a single feature dimension (e.g., the height measurement in Fig. 9.3), then there are only $N_j - 1$ unique partitions of the N_1 points (so each partition has at least one point). If one orders the N_j values, denoted a_n, $n \in 1...N_j$, then the $N_j - 1$ unique partitions correspond to tests $t_j < \tau_n$ where τ_n is the midpoint between a_n and a_{n+1}, i.e., $\tau_n = 0.5(a_n + a_{n+1})$. We can conclude that if we only consider univariate splits, then, given d feature dimensions, there are $d\,(N_j - 1)$ possible tests to consider at node j.

If we consider split tests that involve more than one feature dimension then the number of hypothetical tests increases exponentially. For example, if tests involve two feature values at a time, one must consider d choose 2 times $(N_j - 1)^2$ possible tests. It is to avoid search over exponentially many hypothetical split tests that most simple decision tree learning algorithms only consider univariate splits.

9.3.2 Decision Forests

When decision tree learning is restricted to univariate tests, the learned classifiers are often relatively weak classifiers (i.e., having much less discriminative power than the optimal classifier).

One simple way to improve decision trees is to learn many trees stochastically. As such, each tree tree provides a different conditional distribution. The different trees can then be combined to form one much more powerful classifier. One very simple way to do this is by averaging the conditional distributions (a process known as Bagging). The advantages from combing many different trees relies on the trees being as uncorrelated as possible. That's why it is common to randomly choose different subsets of features to consider for the split function search at each node.

9.4 Class Conditionals

A somewhat different approach to classification arises through modeling the distribution over the features themselves for each class. Such models are usually called "generative" models.

For instance, in the case of binary classification, suppose we have two mutually-exclusive classes c_1 and c_2 (e.g., c_1 and c_2 might be -1 and 1). The prior probability of a data vector coming from class c_1 is $P(c_1) \equiv P(y = c_1)$, and $P(c_2) \equiv P(y = c_2) = 1 - P(c_1)$. Each class has its own distribution for the feature vectors, specifically, $p(\mathbf{x}|c_1)$, and $p(\mathbf{x}|c_2)$; these are the data likelihood distributions for the two classes. The probability of a data point can then be written as (why?):

$$p(\mathbf{x}) \;=\; p(\mathbf{x}, c_1) + p(\mathbf{x}, c_2) \;=\; p(\mathbf{x}|c_1)\, P(c_1) + p(\mathbf{x}|c_2)\, P(c_2)\;. \tag{9.8}$$

If one had such a model, one could draw data samples from the model in the following way: First, one would randomly choose a class according to the probabilities $P(c_1)$ and $P(c_2)$. Then conditioned on the class, one can sample a data point, $\mathbf{x}$, from associated likelihood distribution.

For the learning problem we are given a set of labeled training data $\{(\mathbf{x}_i, y_i)\}$, and our goal is to learn the parameters of the generative model. That is, we want to 1) estimate the conditional likelihood distribution for each class, and 2) estimate $P(c_1)$ by computing the ratio of the number of elements of class 1 to the total number of elements.

Once we have learned the parameters of our *generative* model, we perform classification by comparing the posterior class probabillities:

$$P(c_1|\mathbf{x}) > P(c_2|\mathbf{x})\;? \tag{9.9}$$

That is, if the posterior probability of c_1 is larger than the probability of c_2, then we might classify the input as belonging to class 1. Equivalently, we can compare their ratio to 1:

$$\frac{P(c_1|\mathbf{x})}{P(c_2|\mathbf{x})} > 1\;? \tag{9.10}$$

If this ratio is greater than 1 (i.e. $P(c_1|\mathbf{x}) > P(c_2|\mathbf{x})$) then we classify $\mathbf{x}$ as belonging to class 1, and class 2 otherwise.

The quantities $P(c_i|\mathbf{x})$ can by computed using Bayes' Rule as:

$$P(c_i|\mathbf{x}) \;=\; \frac{p(\mathbf{x}|c_i)\, P(c_i)}{p(\mathbf{x})} \tag{9.11}$$

so that the ratio is:

$$\frac{p(\mathbf{x}|c_1)\, P(c_1)}{p(\mathbf{x}|c_2)\, P(c_2)} \tag{9.12}$$

Note that the $p(\mathbf{x})$ terms cancel and so do not need to be computed. Also, note that these computations are typically done in the logarithmic domain as this is often faster and more numerically stable. In particular, rather than comparing the ratio of the posteriors in (9.12) against 1, it is common to compare the logarithm often the posterior ratio to 0:

$$a(\mathbf{x}) \;=\; \log\left(\frac{p(\mathbf{x}|c_1)\,P(c_1)}{p(\mathbf{x}|c_2)\,P(c_2)}\right) \;>\; 0\,? \tag{9.13}$$

where $a(\mathbf{x})$ defined in this way is often referred to as the decision function, in terms of which the classifier is given by $\text{sgn}(a(\mathbf{x}))$ and the decision boundary is $a(\mathbf{x}) = 0$.

Gaussian class conditionals. As a concrete example, consider a generative model in which the inputs associated with the i^{th} class (for $i = 1, 2$) are modeled with a multi-dimensional Gaussian distribution, i.e.,

$$p(\mathbf{x}|c_i) \;=\; G(\mathbf{x}; \mu_i, \boldsymbol{\Sigma}_i)\,. \tag{9.14}$$

Also, let's assume that the prior class probabilities are equal:

$$P(c_i) \;=\; \frac{1}{2}\,. \tag{9.15}$$

The values of μ_i and $\boldsymbol{\Sigma}_i$ can be estimated by maximum likelihood on the individual classes in the training data.

Given this model, one can show that the log of the posterior ratio (9.12) is given by

$$a(\mathbf{x}) = -\frac{1}{2}(\mathbf{x}-\mu_1)^T\boldsymbol{\Sigma}_1^{-1}(\mathbf{x}-\mu_1) - \frac{1}{2}\ln|\boldsymbol{\Sigma}_1| + \frac{1}{2}(\mathbf{x}-\mu_2)^T\boldsymbol{\Sigma}_2^{-1}(\mathbf{x}-\mu_2) + \frac{1}{2}\ln|\boldsymbol{\Sigma}_2| \tag{9.16}$$

The sign of this function determines the class of $\mathbf{x}$, since the ratio of posterior class probabilities is greater than 1 when this log is greater than zero. Since $a(\mathbf{x})$ is quadratic in $\mathbf{x}$, the decision boundary (i.e., the set of points satisfying $a(\mathbf{x}) = 0$) is a conic section (e.g., a parabola, an ellipse, a line, etc.). Furthermore, in the special case where $\boldsymbol{\Sigma}_1 = \boldsymbol{\Sigma}_2$, the decision boundary is linear (why?).

9.5 Naïve Bayes

One problem with the class conditional models above concerns the large number of parameters required to learn the likelihood distribution, i.e., the input data distribution conditioned on the class. For Gaussian Class-Conditional models, with d-dimensional input vectors, we need to estimate the class mean and the class covariance matrix for each class. The mean is a d-dimensional vector. That is, the covariance is a $d \times d$ matrix (although because it is symmetric we do not need to estimate all d^2 elements). Nevertheless, the number of unknowns in the covariance matrix grows quadratically with d.

Naïve Bayes aims to simplify the estimation problem by assuming that the different input features (e.g., the different elements of the input vector), are conditionally independent. That is,

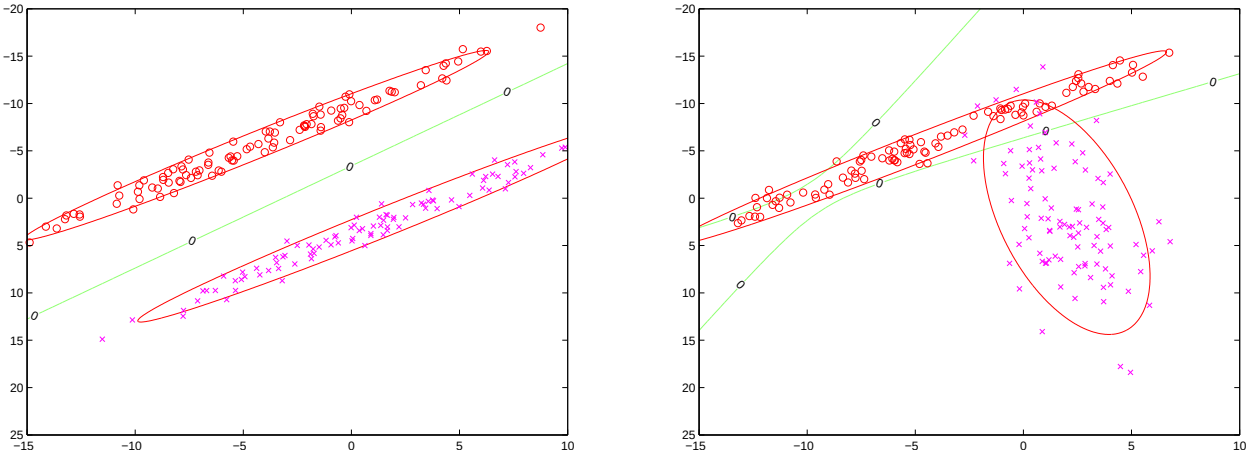

Figure 9.5: GCC classification boundaries for two cases. Note that the decision boundary is linear when both classes have the same covariance.

they are assumed to be independent when conditioned on the class. Mathematically, for inputs $\mathbf{x} \in \mathbb{R}^d$, we express this independence as

$$p(\mathbf{x} \,|\, c) \;=\; \prod_{i=1}^{d} p(x_i \,|\, c) \;. \tag{9.17}$$

With this assumption, rather than estimating one d-dimensional density, we instead estimate d 1-dimensional densities. This is important because each 1D Gaussian only has two parameters, its mean and variance, both of which are scalars. So the model has $2d$ unknowns. In the Gaussian case, the Naïve Bayes model effectively replaces the general $d \times d$ covariance matrix by a diagonal matrix. There are d entries along the diagonal of the covariance matrix; the i^{th} entry is the variance of $x_i|C$. This model is not as expressive, but it's much easier to estimate.

9.5.1 Discrete Input Features

For most of the algorithms thus far we've focused on real-valued inputs. In what follows we consider the Naïve Bayes classification algorithm for discrete inputs. In discrete Naïve Bayes, the inputs are a discrete set of "features", and we'll assume that each input either has or does not have each feature. For example, in document classification (including spam filtering), a feature might be the presence or absence of a particular word, and the feature vector for a document would be a list of which words the document does or doesn't have.

Each data vector is described by a list of discrete features $F_{1:d} = [F_1, ..., F_d]$. For simplicity we'll assume that each feature is binary, so $F_i \in \{0, 1\}$. In the case of document classification, each feature might correspond to the presence of a particular word in the email (e.g., $F_3 = 1$ could indicate that the email contains the word "business"), or another attribute (e.g., $F_4 = 1$ could mean that the mail headers appear forged). And a classifier to distinguish news stories between sports and financial news might be based on particular words or phrases such as "team," "baseball," and "mutual funds."

To understand the complexity of discrete class conditional models in general (i.e., without using the Naïve Bayes model), consider the distribution over 3 input features. For class $c = 1$ this is denoted $P(F_{1:3} \,|\, c = 1)$. (There will be models for other classes, but for our little thought experiment we'll just consider one class). With three binary features there are eight possible inputs, and the likelihood distribution requires a probability for each. The problem is that the number of parameters grows exponentially with the number of features.

Let's have a look at this in more detail. Using the basic rules of probability we find

$$P(F_{1:3} \,|\, c = 1) \;=\; P(F_1 \,|\, F_2, F_3, c = 1)\; P(F_2 \,|\, F_3, c = 1)\; P(F_3 \,|\, c = 1)\;. \tag{9.18}$$

Now, we know F_3 is either 0 or 1 (a coin toss), so we model $P(F_3 \,|\, c = 1)$ with just one number, i.e., the probability $P(F_3 = 1 \,|\, c = 1)$ (as the probability for $F_3 = 0$ is just $1 - P(F_3 = 1 \,|\, c = 1)$). Now consider the second factor $P(F_2 \,|\, F_3, c = 1)$. Because F_2 here depends on F_3, and there are two possible states of F_3, we need to model two distributions, i.e., $P(F_2 \,|\, F_3 = 0, c = 1)$ and $P(F_2 \,|\, F_3 = 1, c = 1)$. We therefore need one more parameter for each. Using the same logic, to model $P(F_1 \,|\, c = 1, F_2, F_3)$ will require one model parameter for each of the 4 possible settings of (F_2, F_3). For d-dimensional binary inputs, it is easy to see that there are $2^d - 1$ parameters that one needs to learn for each class. The number of required parameters grows prohibitively large as d increases.

The Naïve Bayes model, by comparison, only has d parameters to be learned. The assumption of Naïve Bayes is that the feature vectors are all conditionally independent given the class. The independence assumption is often very naïve, but yet the algorithm often works well nonetheless. This means that the likelihood of a feature vector for a particular class j is given by

$$P(F_{1:d} \,|\, c = j) \;=\; \prod_i P(F_i \,|\, c = j) \tag{9.19}$$

where c denotes a class $c \in \{1, 2, ...K\}$. The probabilities $P(F_i|c)$ are parameters of the model, denoted

$$a_{i,j} \;\equiv\; P(F_i = 1 \,|\, c = j) \tag{9.20}$$

We must also define class probability $b_j \equiv P(c = j)$, sometimes called a class prior.

To classify a new feature vector using this model, we choose the class with maximum probability given the features. By Bayes' Rule this is:

$$\begin{aligned} P(c = j \,|\, F_{1:d}) \;&=\; \frac{P(F_{1:d} \,|\, c = j) P(c = j)}{P(F_{1:d})} && (9.21)\\ &=\; \frac{\left(\prod_i P(F_i \,|\, c = j)\right) P(c = j)}{\sum_{\ell=1}^K P(F_{1:d}, c = \ell)} && (9.22)\\ &=\; \frac{\left(\prod_{i:F_i=1} a_{i,j} \prod_{i:F_i=0} (1 - a_{i,j})\right) b_j}{\sum_{\ell=1}^K \left(\prod_{i:F_i=1} a_{i,\ell} \prod_{i:F_i=0} (1 - a_{i,\ell})\right) b_\ell} && (9.23) \end{aligned}$$

If we wish to find the class with maximum posterior probability, we need only compute the numerator. The denominator in (9.23) is of course the same for all classes j. To compute the denominator one simply divides the numerators for each class by their sum.

The above computation involves the product of many numbers, some of which might be quite small. This can lead to underflow. For example, if you take the product $a_1 a_2 ... a_N$, and all $a_i << 1$, then the computation may evaluate to zero in floating point, even though the final computation after normalization should not be zero. If this happens for all classes, then the denominator will be zero, and you get a divide-by-zero error, even though, mathematically, the denominator cannot be zero. To avoid these problems, it is safer to perform the computations in the log-domain:

$$\alpha_j = \left(\sum_{i:F_i=1} \ln a_{i,j} + \sum_{i:F_i=0} \ln(1 - a_{i,j}) \right) + \ln b_j \tag{9.24}$$

$$\gamma = \min_j \alpha_j \tag{9.25}$$

$$P(c = j|F_{1:d}) = \frac{\exp(\alpha_j - \gamma)}{\sum_\ell \exp(\alpha_\ell - \gamma)} \tag{9.26}$$

which, as you can see by inspection, is mathematically equivalent to the original form, but will not evaluate to zero for at least one class.

9.5.2 Learning

For a collection of N training vectors F_k, each with an associated class label c_k, we can learn the parameters by maximizing the data likelihood (i.e., the probability of the data given the model). This is equivalent to estimating multinomial distributions (in the case of binary features, binomial distributions), and reduces to simple counting of features.

Suppose there are N_j training examples of class j, and N examples total. Then the estimate of the class probability is simply:

$$b_j = \frac{N_j}{N} \tag{9.27}$$

Similarly, if class j has $N_{i,j}$ examples for which the ith feature is 1 (ie $F_i = 1$), then

$$a_{i,j} = \frac{N_{i,j}}{N_j} \tag{9.28}$$

With large numbers of features and small datasets, it is likely that some features will never be seen for some classes, giving a class likelihood of zero for that feature (i.e., $a_{i,j}$ is zero when $N_{i,j}$ is zero). We can use regularization to prevent this problem from occurring. We can modify the learning rule as follows:

$$a_{i,j} = \frac{N_{i,j} + \alpha}{N_j + 2\alpha} \tag{9.29}$$

for some small value α. In the extreme case where there are no examples for which feature i is seen for class j, the probability $a_{i,j}$ will be set to $1/2$, corresponding to no knowledge. As the number of examples N_j becomes large, the role of α will become smaller and smaller.

In general, given in a multinomial distribution with a large number of classes and a small training set, we might end up with estimates of prior probability b_j being zero for some class j. This might be undesirable for various reasons, or be inconsistent with our prior beliefs. Again, to

avoid this situation, we can regularize the maximum likelihood estimator with our prior believe that all classes should have a nonzero probability. In doing so we can estimate the class prior probabilities as

$$b_j \;=\; \frac{N_j + \beta}{N + K\beta} \tag{9.30}$$

for some small value of β. When there are no observations whatsoever, all classes are given probability $1/K$. When there are observations the estimated probabilities will lie between N_j/N and $1/K$ (converging to N_j/N as $N \to \infty$).

Derivation. Here we derive just the per-class probability assuming two classes, ignoring the feature vectors; this case reduces to estimating a binomial distribution. The full estimation can easily be derived in the same way.

Suppose we observe N examples of class 0, and M examples of class 1; what is b_0, the probability of observing class 0? Using maximum likelihood estimation, we maximize:

$$\prod_i P(c_i = j) \;=\; \left(\prod_{i:c_i=0} P(c_i = 0)\right)\left(\prod_{i:c_i=1} P(c_i = 1)\right) \tag{9.31}$$

$$\;=\; b_0^N b_1^M \tag{9.32}$$

Furthermore, in order for the class probabilities to be a valid distribution, it is required that $b_0 + b_1 = 1$, and that $b_j \geq 0$. In order to enforce the first constraint, we set $b_1 = 1 - b_0$:

$$\prod_i P(c_i = j) \;=\; b_0^N (1 - b_0)^M \;. \tag{9.33}$$

The log of this is

$$L(b_0) \;=\; N \ln b_0 + M \ln(1 - b_0) \;. \tag{9.34}$$

To maximize, we compute the derivative and set it to zero:

$$\frac{dL}{db_0} \;=\; \frac{N}{b_0} - \frac{M}{1 - b_0} \;=\; 0 \tag{9.35}$$

Multiplying both sides by $b_0(1 - b_0)$ and solving gives:

$$b_0^* \;=\; \frac{N}{N + M} \tag{9.36}$$

which, fortunately, is guaranteed to satisfy the constraint that probability must be non-negative.

9.6 Logistic Regression

Despite the name, logistic regression is a probabilistic form of classification (not regression per se). That is, it is a simple method for computing the posterior probability of the class conditioned on the input, using a simple parametric model. It is used very widely.

The formulation begins with the expression for $p(\mathbf{x})$ used above for class conditional models, i.e.,

$$p(\mathbf{x}) \;=\; p(\mathbf{x}, c_1) + p(\mathbf{x}, c_2) \;=\; p(\mathbf{x}|c_1)\, P(c_1) + p(\mathbf{x}|c_2)\, P(c_2)\;, \tag{9.37}$$

From this it is straightforward to show that the posterior class probability can be expressed as

$$P(c_1|\mathbf{x}) = \frac{p(\mathbf{x}|c_1)P(c_1)}{p(\mathbf{x}|c_1)P(c_1) + p(\mathbf{x}|c_2)P(c_2)}\;. \tag{9.38}$$

Dividing the numerator and denominator by $p(\mathbf{x}|c_1)P(c_1)$ we obtain:

$$P(c_1|\mathbf{x}) \;=\; \frac{1}{1+e^{-a(\mathbf{x})}} \tag{9.39}$$

$$\;=\; g(a(\mathbf{x})) \tag{9.40}$$

where $a(\mathbf{x}) = \ln \frac{p(\mathbf{x}|c_1)P(c_1)}{p(\mathbf{x}|c_2)P(c_2)}$ and $g(a)$ is called the Sigmoid function (because it is shaped like an S). Importantly, note that $g(a)$ is monotonic, so that the probability of class c_1 grows as a grows, and it is precisely $\frac{1}{2}$ when $a(\mathbf{x}) = 0$. Since $P(c_1|\mathbf{x}) = \frac{1}{2}$ represents equal probability for both classes, this is the boundary along which we wish to make decisions about class membership; i.e., the decision boundary is $a(\mathbf{x}) = 0$.

For the case of Gaussian class conditionals where the Gaussian likelihoods for both classes have the same covariance, one can show that the function a becomes a linear function of $\mathbf{x}$. In this case, the classification probability reduces to

$$P(c_1|\mathbf{x}) \;=\; \frac{1}{1+e^{-\mathbf{w}^T\mathbf{x}-b}} \;=\; g(\mathbf{w}^T\mathbf{x}+b)\;, \tag{9.41}$$

or, if we augment the data vector with a 1 and the weight vector with b,

$$P(c_1|\mathbf{x}) \;=\; \frac{1}{1+e^{-\mathbf{w}^T\mathbf{x}}}\;. \tag{9.42}$$

To show that this is true requires some algebraic manipulation, in which the quadratic terms of the class-conditional likelihoods cancel because the covariance matrices are identical.

When the feature vectors do not have Gaussian class conditionals with identical covariances, then the form of the posterior is not so simple. Nevertheless, we are free to reetain the form of Eqn. (9.42) anyway. In other words, whether or not the likelihoods are Gaussian with simmilar covariances or not, by using the form of Eqn. (9.42) we are going to fit this model to the data as well as possible. In essence, even if the underlying likelihoods are not Gaussian the linear decision function may still be a good approximation that is sufficient for classification. We are deciding to sidee-step the problem of learning the distribution over the measurements or observations for each class, and instead we simply estimate the parameters of Eqn. (9.42).

The result is a model with relatively few parameters, since the number of parameters in logistic regression is linear in the dimension of the input vector, while learning a Gaussian covariance requires a quadratic number of parameters. With fewer parameters we can learn models more effectively with less data.[1] The decision boundary for logistic regression, i.e., $a(\mathbf{x}) = 0$, is a linear

[1] On the other hand, we cannot perform other tasks that we could with the generative model (e.g., sampling from the model; classify data with noisy or missing measurements).

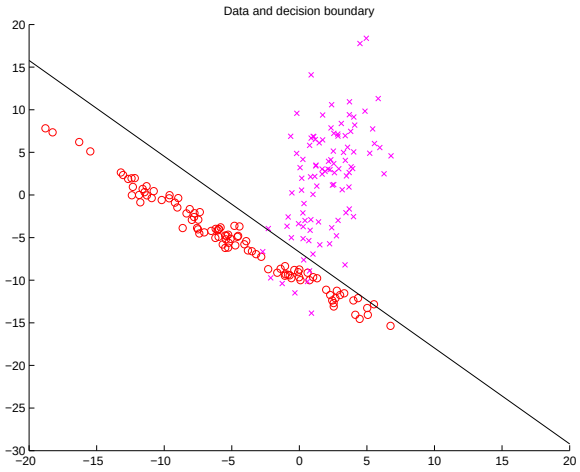

Figure 9.6: For 2D inputs, the decision boundary for logistic regression is just a line. If the bias offset is zero the line will go through the origin. With higher dimensional inputs $\mathbf{x}$ the decision boundary is a hyperplane.

function of the measurements $\mathbf{x}$. In 2D, it's a line. To see this, recall that the decision boundary is the set of points $P(c_1|\mathbf{x}) = 1/2$. Solving for $\mathbf{x}$ yields the points $\mathbf{w}^T\mathbf{x} + b = 0$, which is a line in 2D, and for higher dimensional inputs it becomes a hyperplane. If the bias offset is zero, then the hyperplane goes through the origin.

Multi-class classification. One can also extend this formulation of logistic regression beyond two classes to multi-class problems with K classes in general. To that end, the key is the formulation of a parametric expression for the posterior probability of each of K classes, conditioned on the input measurements. In this case, unlike the two-class case above, here we'll give each class its own weight vector $\mathbf{w}_k$ for $1 \leq k \leq K$. With that, we write the posterior probability of class c_k as follows:

$$P(c_k|\mathbf{x}) \;=\; \frac{e^{-\mathbf{w}_k^T\mathbf{x}}}{\sum_{\ell=1}^{K} e^{-\mathbf{w}_\ell^T\mathbf{x}}} \tag{9.43}$$

(With a little thought you will be able to see that this reduces to the two-class formulation above, in which case the weight vector is equal to the difference in the two class-specific weight vectors here.) This expression is easily shown to represent a sensible choice as it satisfies the basis rules of probability: $0 \leq P(c_k|\mathbf{x})$, and $\sum_\ell P(c_\ell|\mathbf{x}) = 1$ (verify these for yourself). In the deep learning literature this expression for the posterior class probability is often called a softmax function. The softmax functions maps the inner products $\mathbf{w}_k^T\mathbf{x}_i$ into probabilities, in effect treating the inner products as log probabilities (up to an additive constant that is the log denominator that is needed to ensure the probabilities sum to 1). The inner products are often called logits.

Rather than go through more advanced details about the multi-class formulation, we instead turn to the task of learning the logistic regression model in context of binary classification.

9.6.1 Learning

Learning for logistic regression entails the estimation of the model parameters, namely the weight vector $\mathbf{w}$ and the bias offset b. To simplify notation in this section, let's assume, as we have above, that the weight vector $\mathbf{w}$ already incorporates the bias offset, and that the measurement observation vectors have been correspondingly augmented with a 1.

We'll do this using maximum likelihood estimation. In particular, given data $\{\mathbf{x}_i, y_i\}$, we minimize the negative log of

$$\begin{aligned} p(\{\mathbf{x}_i, y_i\} \,|\, \mathbf{w}) &\propto p(\{y_i\} \,|\, \{\mathbf{x}_i\}, \mathbf{w}) \\ &= \prod_i p(y_i|\mathbf{x}_i, \mathbf{w}) \\ &= \prod_{i:y_i=c_1} P(c_1|\mathbf{x}_i) \prod_{i:y_i=c_2} (1 - P(c_1|\mathbf{x}_i)) \end{aligned} \tag{9.44}$$

In the first step above we assumed that the input features are independent of the weights in the logistic regressor, i.e., $p(\{\mathbf{x}_i\}) = p(\{\mathbf{x}_i\}|\mathbf{w})$. So this term can be ignored in the likelihood because it is constant with respect to the unknowns $\mathbf{w}$. The second step assumes that the input-output pairs are independent, so the joint likelihood is the product of the likelihoods for each input-output pair.

Finally, if we assume that c_1 and c_2 are represented as 1 and 0 respectively, then the likelihood over the N data points can be expressed as

$$p(\{\mathbf{x}_i, y_i\}|\mathbf{w}) \;\propto\; \prod_{i=1}^{N} P(c_1|\mathbf{x}_i)^{y_i} \, (1 - P(c_1|\mathbf{x}_i))^{(1-y_i)} \; . \tag{9.45}$$

Accordingly, taking the negative log, we obtain a relatively simple expression for the negative log likelihood. Up to an additive constant, it is given by

$$L(\mathbf{w}) \;=\; -\sum_{i=1}^{N} y_i \, \log P(c_1|\mathbf{x}_i) \;+\; (1 - y_i) \, \log(1 - P(c_1|\mathbf{x}_i)) \; . \tag{9.46}$$

The maximum likelihood weights $\mathbf{w}_{ML}$ are found by minimizing $L(\mathbf{w})$.

Because L is smooth we can use gradient-based optimization. We are looking to find the parameters for which the gradient of L with respect to $\mathbf{w}$ is zero. Finding the gradient L requires a little work, and the use of one very helpful identity. That is, given the sigmoidal function used above, i.e., $g(a) = 1/(1 + e^{-a})$, one can show that

$$\frac{dg}{da} \;=\; g(a) \, (1 - g(a)) \; . \tag{9.47}$$

By defining $p_i \equiv P(c_1|\mathbf{x}_i)$ to simplify notation, we're ready to derive a relatively simply expression for the derivative of the negative log likelihood:

$$\begin{aligned} \frac{\partial}{\partial \mathbf{w}} L(\mathbf{w}) &= -\sum_{i=1}^{N} y_i \, \frac{\partial}{\partial \mathbf{w}} \log p_i \;+\; (1 - y_i) \, \frac{\partial}{\partial \mathbf{w}} \log(1 - p_i) \\ &= -\sum_{i=1}^{N} y_i \, \frac{1}{p_i} \frac{\partial}{\partial \mathbf{w}} p_i \;+\; (1 - y_i) \, \frac{1}{1 - p_i} \frac{\partial}{\partial \mathbf{w}} (1 - p_i) \end{aligned}$$

And because $p_i = g(a_i)$, where $a_i \equiv \mathbf{w}^T \mathbf{x}_i$, it follows that

$$\frac{\partial p_i}{\partial \mathbf{w}} \;=\; \frac{\partial g}{\partial a_i}\,\frac{\partial a_i}{\partial \mathbf{w}} \;=\; g(a_i)\,(1 - g(a_i))\,\mathbf{x}_i\;. \tag{9.48}$$

And hence the gradient simplifies to

$$\begin{aligned}\frac{\partial}{\partial \mathbf{w}} L(\mathbf{w}) \;&=\; -\sum_{i=1}^{N} y_i\,(1 - p_i)\,\mathbf{x}_i \;-\; (1 - y_i)\,p_i\,\mathbf{x}_i \\ &=\; -\sum_{i=1}^{N} (y_i - p_i)\,\mathbf{x}_i\;. \end{aligned} \tag{9.49}$$

Unfortunately, unlike some of the earlier estimation problems we've seen thus far, there is not simple closed form expression for the parameters we wish to estimate. That is there is no simple solution to $\frac{\partial}{\partial \mathbf{w}} L(\mathbf{w}) = 0$. Nevertheless, the good news is that this function is convex, so there is a single minimum. Therefore, we can optimize it with gradient descent (or any other gradient-based search technique), which will be guaranteed to find the global minimum. To learn more about iterative gradient descent as a way to solve more complex optimization problems like this, see Chapter 10.

Regularized logistic regression: Finally, it is important to note that if the data are linearly separable, this approach leads to very large values of the weights $\mathbf{w}$. And as the as the magnitude of $\mathbf{w}$ grows large, the function $g(a(x))$ behaves more and more like a step function and thus assigns higher likelihood to the data. This can make estimation unstable, but can be easily prevented by placing a weight-decay prior on $\mathbf{w}$; i.e., assuming a prior, $p(\mathbf{w}) = G(\mathbf{w};\, 0, \sigma^2)$. This amounts to added a term like $\mathbf{w}^T\mathbf{w}/(2\sigma^2)$ to the negative log likelihood above, and hence we just add a term $\mathbf{w}/\sigma^2$ to the gradient.

9.7 Generative vs. Discriminative Models

The different classifiers described above help to illustrate a distinction between two general types of models in machine learning:

1. **Discriminative models**, such as logistic regression, decision trees and random decision forests, and k-NN classification, attempt to model the conditional probability of the target output given the input, i.e., $p(y \,|\, \mathbf{x})$;

2. **Generative models**, such as the class conditional approach and naive Bayes model in this chapter, aim to model the complete probability of the training data, i.e., $p(\mathbf{x}, y)$. Since $p(\mathbf{x}, y) \;=\; p(y \,|\, \mathbf{x})\,p(\mathbf{x})$, generative models model both the conditional distribution of the target given the input, as well as the distribution of the input features. Often generative models are specified in terms of a likelihood function $p(\mathbf{x} \,|\, y)$ and a prior over possible target outcomes $p(y)$.

The same distinction occurs in both regression and classification, e.g., k-NN is a discriminative method that can be used for either classification or regression.

The distinction is clearest when comparing LR with GCC with equal covariances, since they are both linear classifiers, but the training algorithms are quite different. This is because they have different goals; LR is optimized for classification performance, where as the GCC is a "complete" model of the probability of the data that is then pressed into service for classification. As a consequence, GCC may perform poorly with non-Gaussian data. Conversely, LR is not premised on any particular form of distribution for the two class distributions. On the other hand, LR can **only** be used for classification, whereas the GCC can be used for other tasks, e.g., to sample new $\mathbf{x}$ data, to classify noisy inputs or inputs with outliers, and so on.

The distinctions between generative and discriminative models become more significant in more complex problems. Generative models allow us to put more prior knowledge into how we build the model, but classification may often involve difficult optimization of $p(y|\mathbf{x})$; discriminative methods are typically more efficient and generic, but are harder to specialize to particular problems.

