STA 414/2104

Statistical Methods for Machine Learning and Data Mining

Radford M. Neal, University of Toronto, 2012

Week 8

Classification

Classification Problems

Many machine learning applications can be seen as classification problems — given a vector of p "inputs" describing an item, predict which "class" the item belongs to. Examples:

- Given anatomical measurements of an animal, predict which species the animal belongs to.
- Given information on the credit history of a customer, predict whether or not they would pay back a loan.
- Given an image of a hand-written digit, predict which digit (0-9) it is.
- Given the proportions of iron, nickel, carbon, etc. in a type of steel, predict whether the steel will rust in the presence of moisture.

We assume that the set of possible classes is known, with labels $C_1, \ldots, C_K$.

We have a training set of items in which we know both the inputs and the class, from which we will somehow learn how to do the classificaton.

Once we've learned a classifier, we use it to predict the class of future items, given only the inputs for those items.

Approaches to Classification

Classification problems can be solved in (at least) three ways:

- Learn how to directly produce a class from the inputs — that is, we learn some function that maps an input vector, x, to a class, C_k.
- Learn a "discriminative" model for the probability distribution over classes for given inputs — that is, learn $P(C_k|x)$ as a function of x. From $P(C_k|x)$ and a "loss function", we can make the best prediction for the class of an item.
- Learn a "generative" model for the probability distribution of the inputs for each class — that is, learn $P(x|C_k)$ for each class k. From this, and the class probabilities, $P(C_k)$, we can find $P(C_k|x)$ using Bayes' Rule.

Note that the last option above makes sense only if there is some well-defined distribution of items in a class. This isn't the case for the previous example of determining whether or not a type of steel will rust.

Loss functions and Classification

Learning $P(C_k|x)$ allows one to make a prediction for the class in a way that depends on a "loss function", which says how costly different kinds of errors are.

We define L_{kj} to be the loss we incur if we predict that an item is in class C_j when it is actually in class C_k. We'll assume that losses are non-negative and that $L_{kk} = 0$ for all k (ie, there's no loss when the prediction is correct). Only the relative values of losses will matter.

If all errors are equally bad, we would let L_{kj} be the same for all $k \neq j$.

Example: Giving a loan to someone who doesn't pay it back (class C_1) is much more costly than not giving a loan to someone who would pay it back (class C_0). So for this application we might define $L_{01} = 1$ and $L_{10} = 10$.

Note that in this example we should define the loss function to account both for monetary consequences (money not repaid, or interest not earned) and other effects that don't have immediate monetary consequences, such as customer dissatisfaction when their loan isn't approved.

Predicting to Minimize Expected Loss

A basic principle of decision theory is that we should take the action (here, make the prediction) that minimizes the *expected* loss, according to our probabilistic model.

If we predict that an item with inputs x is in class C_j, the expected loss is

$$\sum_{k=1}^{K} L_{kj} P(C_k|x)$$

We should predict that this item in the class, C_j, for which this expected loss is smallest. (The minimum might not be unique, in which case more than one prediction would be optimal.)

If all errors are equally bad (say loss of 1), the expected loss when predicting C_j is $1 - P(C_j|x)$, so we should predict the class which highest probability given x.

For binary classification ($K = 2$, with classes labelled by 0 and 1), minimizing expected loss is equivalent to predicting that an item is in class 1 if

$$\frac{P(C_1|x)}{P(C_0|x)} \frac{L_{10}}{L_{01}} \quad > \quad 1$$

Generative Models for Classification

Classification from Generative Models Using Bayes' Rule

In the generative model approach to classification, we learn models from the training data for the probability or probability density of the inputs, x, for items in each of the possible classes, C_k — that is, we learn models for $P(x|C_k)$ for $k = 1, \ldots, K$.

To do classification, we instead need $P(C_k|x)$. We can get these conditional class probabilities using Bayes' Rule:

$$P(C_k|x) \quad = \quad \frac{P(C_k)\, P(x|C_k)}{\sum_{j=1}^{K} P(C_j)\, P(x|C_j)}$$

Here, $P(C_k)$ is the prior probability of class C_k. We can easily estimate these probabilities by the frequencies of the classes in the training data. Alternatively, we may have good information about $P(C_k)$ from other sources (eg, census data).

For binary classification, with classes C_0 and C_1, we get

$$\begin{aligned} P(C_1|x) \quad &= \quad \frac{P(C_1)\, P(x|C_1)}{P(C_0)\, P(x|C_0) \;+\; P(C_1)\, P(x|C_1)} \\ &= \quad \frac{1}{1 \;+\; P(C_0)\, P(x|C_0) \,/\, P(C_1)\, P(x|C_1)} \end{aligned}$$

Naive Bayes Models for Binary Inputs

When the inputs are binary (ie, x is a vectors of 1's and 0's) we can use the following simple generative model:

$$P(x|C_k) \quad = \quad \prod_{i=1}^{p} \theta_{ki}^{x_i} \, (1-\theta_{ki})^{1-x_i}$$

Here, θ_{ki} is the estimated probability that input i will have the value 1 in items from class k.

The maximum likelihood estimate for θ_{ki} is simply the fraction of 1's in training items that are in class k.

This is called the *naive Bayes* model — "Bayes" because we use it with Bayes's Rule to do classification, an "naive" because this model assumes that inputs are independent given the class, which is something a naive person might assume, though it's usually not true.

It's easy to generalize naive Bayes models to discrete inputs with more than two values, and further generalizations (keeping the independence assumption) are also possible.

Binary Classification using Naive Bayes Models

When there are two classes (C_0 and C_1) and binary inputs, applying Bayes' Rule with naive Bayes gives the following probability for C_1 given x:

$$\begin{aligned} P(C_1|x) &= \frac{P(C_1)\,P(x|C_1)}{P(C_0)\,P(x|C_0) \;+\; P(C_1)\,P(x|C_1)} \\ &= \frac{1}{1 \;+\; P(C_0)\,P(x|C_0)\,/\,P(C_1)\,P(x|C_1)} \\ &= \frac{1}{1 \;+\; \exp(-a(x))} \end{aligned}$$

where

$$\begin{aligned} a(x) &= \log\left(\frac{P(C_1)\,P(x|C_1)}{P(C_0)\,P(x|C_0)}\right) \\ &= \log\left(\frac{P(C_1)}{P(C_0)}\prod_{i=1}^{p}\left(\frac{\theta_{1i}}{\theta_{0i}}\right)^{x_i}\left(\frac{1-\theta_{1i}}{1-\theta_{0i}}\right)^{1-x_i}\right) \\ &= \log\left(\frac{P(C_1)}{P(C_0)}\prod_{i=1}^{p}\frac{1-\theta_{1i}}{1-\theta_{0i}}\right) \;+\; \sum_{i=1}^{p} x_i \log\left(\frac{\theta_{1i}/(1-\theta_{1i})}{\theta_{0i}/(1-\theta_{0i})}\right) \end{aligned}$$

Gaussian Generative Models

When the inputs are real-valued, a Gaussian model for the distribuiton of inputs in each class may be appropriate. If we also assume that the covariance matrix for all classes is the same, the class probabilities for binary classification turn out to depend on a linear function of the inputs.

For this model,

$$P(x|C_k) \;=\; (2\pi)^{-p/2}\,|\Sigma|^{-1/2}\,\exp\Big(-(x-\mu_k)^T\Sigma^{-1}(x-\mu_k)\,/\,2\Big)$$

where μ_k is an estimate of the mean vector for class C_k, and Σ is an estimate for the covariance matrix (same for all classes).

One can show that the maximum likelihood estimate for μ_k is the sample means of input vectors for items in class C_k, and the maximum likelihood estimate for Σ is

$$\sum_{k=1}^{K} \frac{n_k}{n} S_k$$

where n is the total number of training items, n_k is the number of training items in class C_k, and S_k is the usual maximum likelihood estimate for the covariance matrix based on items in class C_k.

Classification using Gaussian Models for Each Class

For binary classification, we can now apply Bayes' Rule to get the probability of class 1 from a Gaussian model with the same covariance matrix in each class:

As for the naive Bayes model:

$$P(C_1|x) \;=\; \frac{1}{1 \;+\; \exp(-a(x))}$$

where

$$a(x) \;=\; \log\left(\frac{P(C_1)\,P(x|C_1)}{P(C_0)\,P(x|C_0)}\right)$$

Substituting the Gaussian densities, we get

$$\begin{aligned} a(x) \;&=\; \log\left(\frac{P(C_1)}{P(C_0)}\right) \;+\; \log\left(\frac{\exp(-(x-\mu_1)^T\Sigma^{-1}(x-\mu_1)\,/\,2)}{\exp(-(x-\mu_0)^T\Sigma^{-1}(x-\mu_0)\,/\,2)}\right) \\ &=\; \log\left(\frac{P(C_1)}{P(C_0)}\right) \;+\; \frac{1}{2}\Big(\mu_0^T\Sigma^{-1}\mu_0 - \mu_1^T\Sigma^{-1}\mu_1\Big) \;+\; x^T\Big(\Sigma^{-1}(\mu_1-\mu_0)\Big) \end{aligned}$$

The quadratic terms of the form $x^T\Sigma^{-1}x/2$ cancel, producing a linear function of the inputs, as was also the case for naive Bayes models.

Discriminative Models for Classification

Logistic Regression

We see that binary classification using either naive Bayes or Gaussian generative models leads to the probability of class C_1 given inputs x having the form

$$P(C_1|x) \quad = \quad \frac{1}{1 \; + \; \exp(-a(x))}$$

where $a(x)$ is a linear function of x, which can be written as $a(x) = \beta_0 + x^T\beta$.

Rather than start with a generative model, however, we could simply start with this formula, and estimate β_0 and β from the training data. Maximum likelihood estimation for β_0 and β is not hard, though there is no explicit formula.

This is a discriminative training procedure, that estimates $P(C_k|x)$ without estimating $P(x|C_k)$ for each class.

Which is Better — Generative or Discriminative?

Even though logistic regression uses the same formula for the probability for C_1 given x as was derived for the earlier generative models, maximum likelihood logistic regression does *not* in general give the same values for β_0 and β as would be found with maximum likelihood estimation for the generative model.

So which gives better results? It depends...

If the generative model accurately represents the distribution of inputs for each class, it should give better results than discriminative training — it effectively has more information to use when estimating parameters.

However, if the generative model is not a good match for the actual distributions, using it might produce very bad results, even when logistic regression would work well. The independence assumption for naive Bayes and the equal covariance assumption for Gaussian models are often rather dubious.

Similarly, logistic regression may be less sensitive to outliers than a Gaussian generative model.

Non-linear Logistic Models

As we saw earlier for neural networks, the form $P(C_1|x) = 1 / (1 + \exp(-a(x)))$ for the probability of class C_1 given x can be used with $a(x)$ being a non-linear function of x.

If $a(x)$ can equally well be any non-linear function, the choice of this form for $P(C_1|x)$ doesn't really matter, since *any* function $P(C_1|x)$ could be obtained with an appropriate $a(x)$.

However, in practice, non-linear models are biased towards some non-linear functions more than others, so it does matter that a logistic model is being used, though not as much as when $a(x)$ must be linear.

Probit Models

An alternative to logistic models is the *probit* model, in which we let

$$P(C_1|x) \quad = \quad \Phi(a(x))$$

where Φ is the cumulative distribution function of the standard normal distribution:

$$\Phi(a) \quad = \quad \int_{-\infty}^{a} (2\pi)^{-1/2} \exp(-x^2/2)\, dx$$

This would be the right model if the class depended on the sign of $a(x)$ plus a standard normal random variable. But this isn't a reasonable model for most applications. It might still be useful, though.

Gaussian Process Models for Classification

A Gaussian process logistic regression model for data $(x_1, y_1), \ldots, (x_n, y_n)$ with the y_i being binary can be expressed as

$$\begin{aligned} \theta &\sim \ldots \\ f &\sim GP(\theta) \\ y_i \,|\, x_i,\, f &\sim \text{Bernoulli}\left(1 \,/\, (1 + \exp(-f(x_i)))\right) \end{aligned}$$

where θ represent all the parameters of the Gaussian process's covariance function.

Alternatively, we could use a probit model, with

$$y_i \,|\, x_i,\, f \sim \text{Bernoulli}\left(\Phi(f(x_i))\right)$$

However, with neither of these models can we use simple matrix operations to evaluate $P(y|x, \theta)$, or to predict a new y^* by $P(y^*|x^*, y, x, \theta)$.

The Latent Gaussian Process

To do computations for Gaussian process classification, we need to explicitly represent the "latent variables" $z_i = f(x_i)$.

Using matrix operations, we can compute the joint density of the latent variables and observed responses (for given θ):

$$\begin{aligned} & P(z_1, \ldots, z_n, y_1, \ldots, y_n \,|\, x_1, \ldots, x_n, \theta) \\ & \quad = \quad P(z_1, \ldots, z_n \,|\, x_1, \ldots, x_n, \theta)\, P(y_1, \ldots, y_n \,|\, z_1, \ldots, z_n) \end{aligned}$$

The first factor above (the prior for latent variables) is Gaussian; the second (the likelihood for these latent variables) is a simple product of Bernoulli probabilities (from a logit or probit model).

Implementing the model with Latent Variables

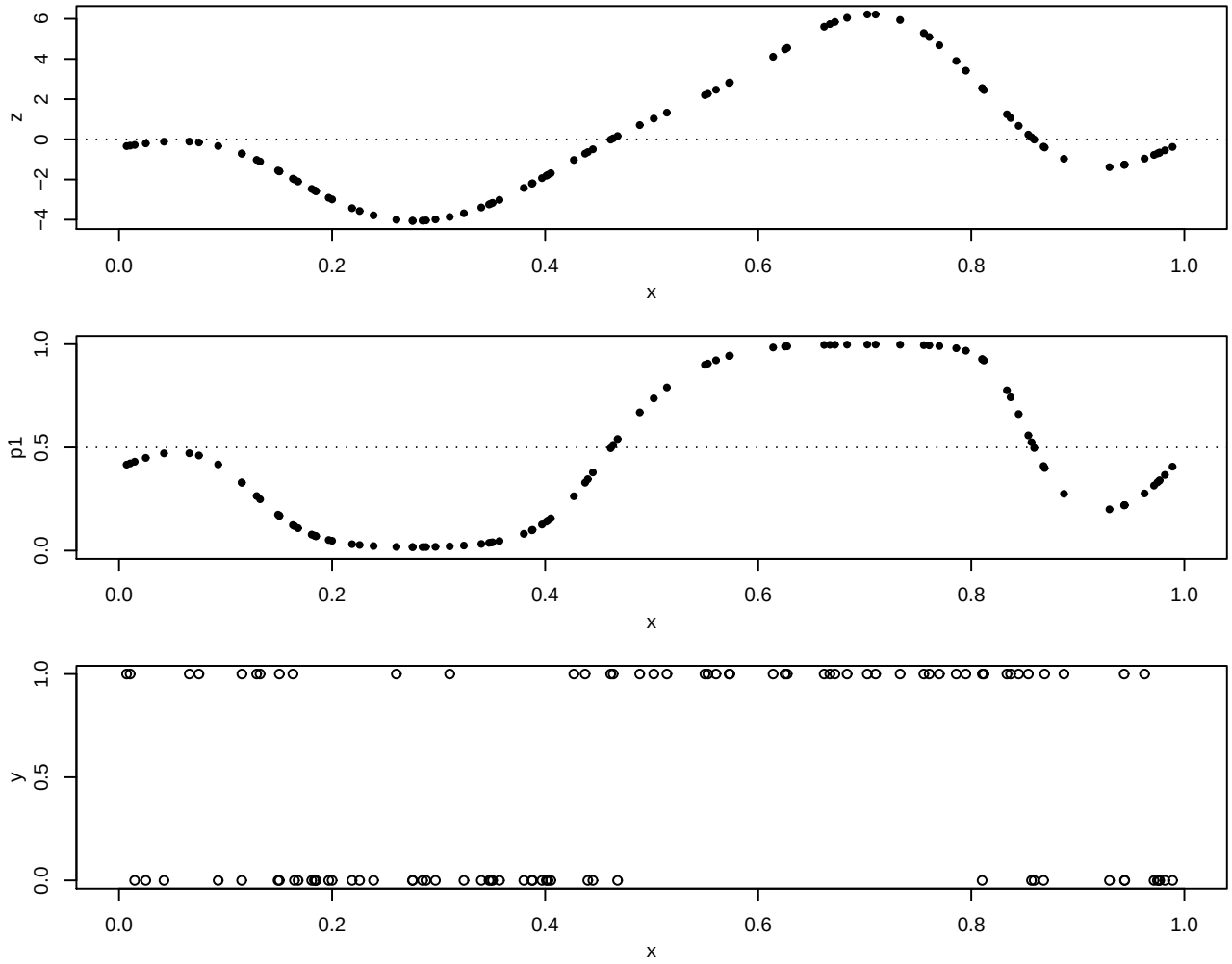
Two methods are commonly used for handling the latent variables:

Approximate their posterior distribution by a Gaussian: The prior for $z_1, \ldots, z_n$ given θ is Gaussian. The likelihood, $P(y_1, \ldots, y_n \mid z_1, \ldots, z_n)$, is not, but as $n \to \infty$ it will approach a Gaussian form. Maybe a Gaussian approximation of the posterior for $z_1, \ldots, z_n$ will be adequate for finite n.

Sample for the latent variables using Markov chain Monte Carlo: We use a Markov chain to sample $z_1, \ldots, z_n$, conditional on θ and $x_i, \ldots, x_n$. We also sample for θ (unless it is fixed). We them make a prediction for a test case at x^* by averaging $P(y^*|x^*, z, x, \theta)$ over the sample of values we obtain for z and θ.

Illustration of a Gaussian Process Classification Model

Consider a Gaussian process classification model with one input, with covariance function $K(x, x') = 0.5^2 + 3^2 \exp(-5^2(x - x')^2)$. Below is a random sample from the latent Gaussian process at x values drawn uniformly from $(0, 1)$, the resulting probabilities that $y = 1$, and values for y drawn according to these probabilities:



Non-probabilistic Models for Classification

The Large Margin Hard Classifier

Some classification methods produce *only* a predicted class for a test case, with no probability distribution for that class.

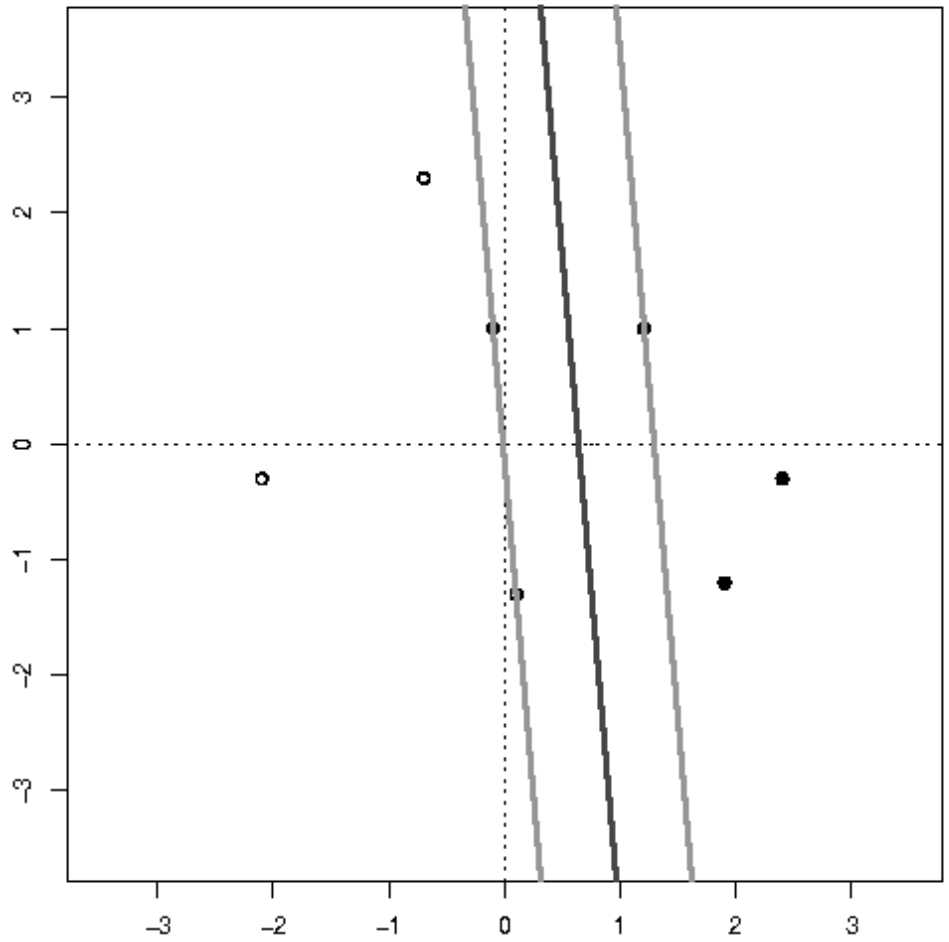
Large margin classifiers are in this category. They are the basis for the popular *Support Vector Machine (SVM)* classifiers.

In their simplest form, large margin classifiers apply only to perfectly separable binary classification problems, in which there is a hyperplane that separates the training cases in one class from the training cases in the other class.

There will usually be many hyperplanes separating the classes. The idea is to pick the separating hyperplane that has the largest *margin* — the minimum distance of a training case from the line.

An Illustration with Two Inputs

Here is a large margin classifier in two dimensions (where a hyperplane is a straight line). There are four training cases in Class -1 (white) on the left, and three in Class $+1$ (black) on the right. The dark line is the separating hyperplane, used to predict the class of a test case. The lighter lines show the margin.



Finding the Separating Hyperplane with Largest Margin

We can define a hyperplane by the equation $w^T x + b = 0$. We can use w and b to classify test cases to the class $\text{sign}(w^T x + b)$. (We'll use -1 and $+1$ as class labels.)

Note that negating w and b swapw which side of the hyperplane has which class, and multiplying w and b by any positive constant doesn't change classifications.

When finding w and b from the training cases, we will impose the constraint that all training cases are classified correctly — that is,

$$y_i(w^T x_i + b) > 0, \quad \text{for } i = 1, \ldots, n$$

But we want to also maximize the margin, which is

$$\min_{i=1,\ldots,n} \frac{y_i}{||w||}(w^T x_i + b)$$

The is equivalent to the following optimization problem:

$$\text{minimize } ||w||^2, \quad \text{subject to } y_i(w^T x_i + b) \geq 1 \text{ for } i = 1, \ldots, n$$

(The minimizaton will shrink w to where at least one inequality above is an equality, at which point the margin will be $1/||w||$, so maximizing the margin is the same as minimizing $||w||^2$.)

Characteristics of the Maximum Margin Separating Hyperplane

The previous slide characterizes the maximum margin hyperplane as minimizing a quadratic function, subject to linear inequality constraints.

This is a convex optimization problem. It has a unique solution, which can be found reasonably efficiently by standard methods (or more efficiently using specialized methods).

The solution is locally sensitive to a subset of the training cases, called the *support vectors* — typically, but not always, less (often much less) than the full set of training cases.

Of course, all training cases have to be looked at before the support vectors can be identified. But when there are only a few support vectors, the computations do go faster.

Why Might a Large Margin Classifier be Good?

The maximum margin separating hyperplane seems intutively like it should be better than some other separating hyperplanes, such as one that goes very close to a training point.

It's also the same as you get from logistic regression with coefficients that maximize the log likelihood minus an infinitesimal quadratic penalty.

Some theorists have attempted to justify large margin classifiers using "VC-dimension" arguments — that relate to how much potential there is for overfitting — but it's not clear these arguments actually succeed.

Perhaps one can see the large margin classifier as approximating Bayesian predictions (based on a vague prior distribution)...

Comparison with a Bayesian Hard Linear Classifier

The data set in the earler slide illustrating a large margin classifier is the same as the one I used in the introduction to Bayesian inference. Here from that demonstration is the curve where the classes have equal predictive probability, together with the maximum margin classifie: