An Overview of Statistical Machine Learning Part II

Babak Shahbaba, Ph.D.

Associate Professor, Department of Statistics
University of California, Irvine

Irvine, CA
July 11, 2017

Classification Models

Logistic regression model

- When dealing with binary outcome variables, we assume the response variable, y_i, has a Bernoulli distribution (or Binomial if $n_i > 1$),

$$y_i|\mu_i \sim \mathrm{Bernoulli}(\mu_i)$$

- In this case, a common link function to connect the mean of the response variable to a set of predictors, x_i, is the *logit* function defined as follows:

$$g(\mu_i) = \log(\frac{\mu_i}{1-\mu_i}) = \log[\frac{P(y_i = 1|x_i, \beta)}{1 - P(y_i = 1, \beta|x_i)}] = x_i\beta$$

where $\beta = (\beta_0, \beta_1, ..., \beta_p)$.

- Note that

$$\mu_i \quad = \quad P(y_i = 1|x_i, \beta) = \frac{\exp(x_i\beta)}{1 + \exp(x_i\beta)}$$

Interpretation

- To interpret β, notice that $\log[\frac{P(y_i=1|x_i,\beta)}{1-P(y_i=1,\beta|x_i)}]$ is the log of odds for the outcome of interest, $y_i = 1$.
- The intercept β_0 is therefore the log of odds when all predictors are set to zero (note that this might not make sense in some cases).
- Or we can say, $\exp(\beta_0)$ is the odds when all predictors are set to zero.
- $\exp(\beta_j)$ on the other hand is how much the odds multiplicatively increases for one unit increase in x_j when all other predictors are fixed.
- Or we can say, $\exp(\beta_j)$ is the odds ratio for subjects with $X_j = x_j + 1$ compared to subjects with $X_j = x_j$ when all other predictors are fixed.
- Positive β_j indicates that the odds increases as x_j increases (everything else fixed), where is for negative estimate of β_j the odds decreases as x_j increases (everything else fixed).

Linear discriminant analysis

- When the set of p predictors, x, are continuos random variables, we can assume that their joint distribution is multivariate normal for each class,

$$f_k(x) = (2\pi)^{-p/2}|\Sigma|^{-1/2} \exp[-\frac{1}{2}(x-\mu_k)^T\Sigma^{-1}(x-\mu_k)]$$

- Note that in this setting, only the mean of the distributions, μ_k, changes from one class to another. The covariance matrix Σ remains the same for all classes.
- This assumption is of course not realistic and is made only for simplicity. We will relax it later.

Linear discriminant analysis

- Using Bayes theorem, we have

$$P(y = k|x) = \frac{\pi_k f_k(x)}{\sum_{k'=1}^{K} \pi_{k'} f_{k'}(x)}$$

  where $\pi_k = P(y = k)$.

- For a given value of x, the denominator remains the same for all classes. Therefore, we can define the discriminant function based on the numerator, $\pi_k f_k(x)$, or more commonly based on its log,

$$\begin{aligned} \delta_k(x) &= \log \pi_k + log[f_k(x)] \\ &= \log \pi_k - \frac{1}{2}\log(|\Sigma|) - \frac{1}{2}(x - \mu_k)^T \Sigma^{-1}(x - \mu_k) \end{aligned}$$

Linear discriminant analysis

- With further simplification (and removing the constant parts), we have

$$\delta_k(x) \quad = \quad \log \pi_k - \frac{1}{2}\mu_k^T \Sigma^{-1} \mu_k + x^T \Sigma^{-1} \mu_k$$

- Note that the above functions are linear in x.
- Therefore, we refer to them as *linear discriminant functions*.
- Classifying cases according to these functions is called *linear discriminant analysis* (LDA).

Linear discriminant analysis

- We can estimate π_k and μ_k for $k = 1, \ldots, K$, and Σ as follows:

$$
\begin{aligned}
\hat{\pi}_k &= \frac{n_k}{n} \\
\hat{\mu}_k &= \frac{1}{n_k} \sum_{i:y_i=k}^{n_k} x_i \\
\hat{\Sigma} &= \frac{1}{n-k} \sum_{k=1}^{K} \sum_{i:y_i=k}^{n_k} (x_i - \hat{\mu}_k)(x_i - \hat{\mu}_k)^T
\end{aligned}
$$

where n_k is the number of observed cases (training cases) that belong to class k.

Linear discriminant analysis

- After estimating the model parameters, we assign each case, i, to the class whose value of the discriminant function, $\delta_k(x_i)$, is the highest.
- Cases for which $\delta_k(x) = \delta_l(x)$ fall on the decision boundary between the two classes k and l.
- For these cases, $\delta_k(x) - \delta_l(x) = 0$, which means

$$\log \frac{\pi_k}{\pi_l} - \frac{1}{2}(\mu_k - \mu_l)^T \Sigma^{-1}(\mu_k - \mu_l) + x^T \Sigma^{-1}(\mu_k - \mu_l) = 0$$

- Note that the above equation, which specifies the decision boundary, is linear in x. As the result, the decision boundaries are *hyperplanes* in the p-dimensional space. (The decision boundary is straight line if we have two predictors only.)

Linear discriminant analysis

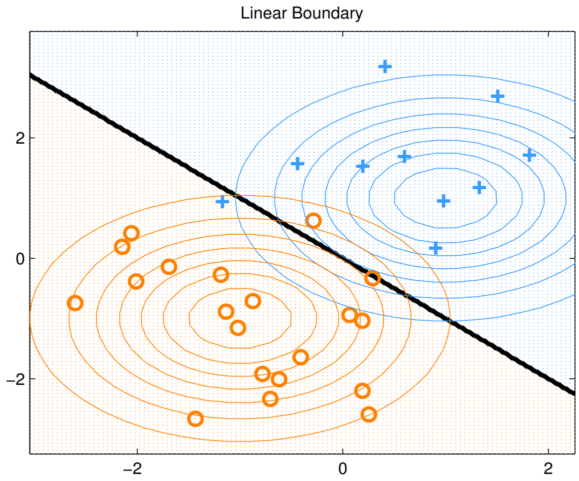


Figure: Figure 4.5a in Murphy (2012)

Quadratic discriminant analysis

- As mentioned above, the equal-covariance assumption is restrictive and is only made for convenience.
- By relaxing this assumption, the discriminant function becomes

$$\delta_k(x) \quad = \quad \log \pi_k - \frac{1}{2}\log(|\Sigma_k|) - \frac{1}{2}(x - \mu_k)^T \Sigma_k^{-1}(x - \mu_k)$$

  which are quadratic functions of x; hence, they are called *quadratic discriminant functions*.
- Classifying cases according to these functions is called *quadratic discriminant analysis* (QDA).
- The decision boundaries for this approach are not linear any more.

Linear discriminant analysis

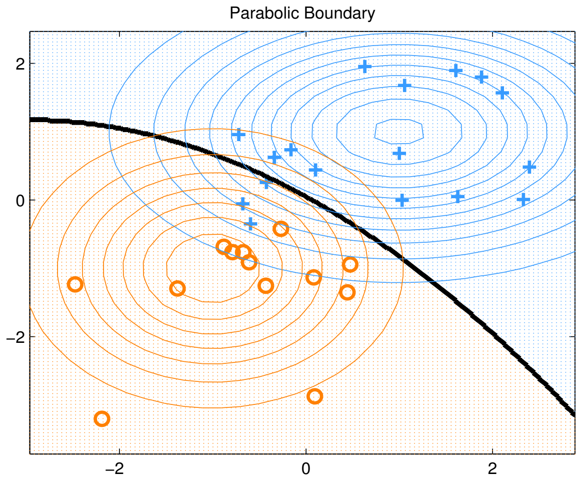


Figure: Figure 4.3a in Murphy (2012)

Naive Bayes models

- This is an alternative classification model, which is especially attractive when the dimension p is large.
- In this approach, we again use Bayes theorem to obtain the probability of each class given the observed values of predictors,

$$P(y = k|x_1, \dots, x_p) \quad = \quad \frac{P(y = k)P(x_1, \dots, x_p|y = k)}{\sum_{k'=1}^{K} P(y = k')P(x_1, \dots, x_p|y = k')}$$

- This time, however, we make an assumption that is naive and possibly wrong, but it simplifies the model: we assume that given a class $y = k$, the predictors are independent,

$$P(x_1, \dots, x_p|y = k) \quad = \quad \prod_{j=1}^{p} P(x_j|y = k)$$

Naive Bayes models

- As a result of the above naive assumption, the model simplifies to

$$P(y = k|x_1, \ldots, x_p) = \frac{P(y = k) \prod_{j=1}^{p} P(x_j|y = k)}{\sum_{k'=1}^{K} P(y = k') \prod_{j=1}^{p} P(x_j|y = k')}$$

- As before, we assign each case, i, to the class with the highest conditional probability given $x_{i1}, \ldots, x_{ip}$.
- It is more common to distinguish between two classes using the following logit function

$$\begin{aligned}\log \frac{P(y = k|x_1, \ldots, x_p)}{P(y = l|x_1, \ldots, x_p)} &= \log \frac{P(y = k) \prod_{j=1}^{p} P(x_j|y = k)}{P(y = l) \prod_{j=1}^{p} P(x_j|y = l)} \\ &= \log \frac{\pi_k}{\pi_l} + \sum_{j=1}^{p} \log \frac{P(x_j|y = k)}{P(x_j|y = l)}\end{aligned}$$

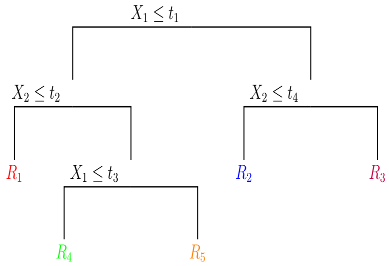
Naive Bayes models

- In practice, we estimate π_k using the proportion of observed cases that belong to class k.
- To estimate $P(x_j|k)$, we first need to assume a probability distribution model for x_j given k.
- If x_j is categorical, we can estimate $P(x_j|k)$ using the observed proportion of each category of x_j for cases with $y = k$.
- If x_j is continuous, we can assume $x_j|k$ has a Gaussian distribution and estimate its mean and variance using the cases with $y = k$.

Tree Models

Tree Models

- Decision trees are models that recursively partition the input space into regions and define a local map between each region and the response variable.



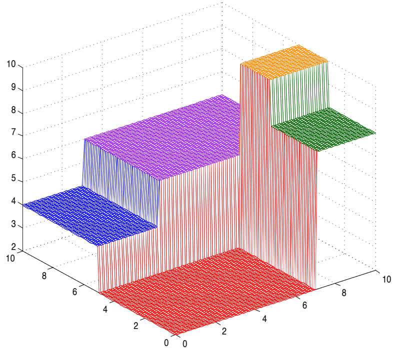


Figure: Figure 16.1 in Murphy (2012).

Tree Models

- Starting from the root, when a rule is satisfied, we move to the left branch; otherwise, we move to the right branch.
- When we reach a leaf, we use the subset of data, which fall in the corresponding region, to estimate the response variable.
- The corresponding model can be presented as follows:

$$\hat{y}_i = \sum_{m=1}^{M} \hat{y}_m I(x_i \in R_m)$$

Tree Models

- For regression model, $\hat{y}_m$ can be simply the mean response in region R_m, or the regression estimate using the subset in R_m.
- For classification models, we use the sample proportions, p_{mc}, within region R_m as the estimate for the probability of class c.
- We usually assign a case to the class with the highest probability.

CART

- The most commonly used decision tree method is Classification and Regression Tree (CART).
- To build a CART model, we first *grow* a tree using recursive *binary* splits.
- We usually stop the procedure when some stopping criterion is met. For example: the leaves must have at least m observations.
- The resulting model is typically too complex.
- Next, we *prune* the tree to obtain a simpler model and to avoid overfitting.

Growing a Tree

- To grow a tree,
  - we choose a *cost* function, which typically reflects "impurity"
  - at each node (starting with the whole data at the root), we find the best input variable (feature), j^*, to split the data,
  - and find the best cutoff, t^* for the split.
- For numerical variables, the best (j^*, t^*) is defined as follows:

$$\begin{aligned}(j^*, t^*) &= \arg \min_{j \in \{1, \dots, p\}} \min_t \{\mathrm{cost}(x_{ij}, y_i : x_{ij} \le t) \\ &\quad + \mathrm{cost}(x_{ij}, y_i : x_{ij} > t)\}\end{aligned}$$

- When x_j is categorical with K categories, the splits are usually based on one group versus all other groups: $x_{ij} = k$ vs. $x_{ij} \neq k$.

Cost Functions

- For regression trees, the square error cost function is commonly used,

$$\mathrm{cost}(\mathcal{D}) = \sum_{i \in \mathcal{D}} (y_i - \hat{y}_{\mathcal{D}})^2$$

- $\hat{y}_{\mathcal{D}}$ is the estimate of the response variable (e.g., mean or regression estimate) using the observations in $\mathcal{D}$.

Cost Functions

- For classification trees, commonly used cost functions are
  - misclassification rate

$$\mathrm{cost}(\mathcal{D}) = \frac{1}{|\mathcal{D}|} \sum_{i \in \mathcal{D}} (y_i \neq \hat{y}_{\mathcal{D}})$$

  - Entropy

$$\mathrm{cost}(\mathcal{D}) = \sum_{\mathcal{D};c=1}^{C} p_c \log \frac{1}{p_c}$$

  - Gini index

$$\mathrm{cost}(\mathcal{D}) = \sum_{\mathcal{D};c=1}^{C} p_c(1 - p_c)$$

- p_c is the sample proportion of class c in the subset $\mathcal{D}$.

Cost Functions

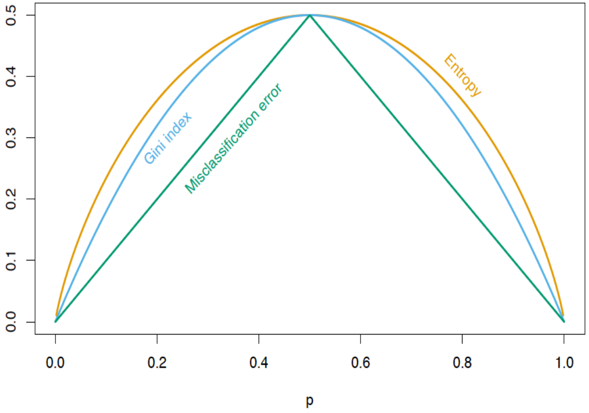


Figure: Figure 9.3 in Hastie et al. (2010) for binary classification.

Pruning

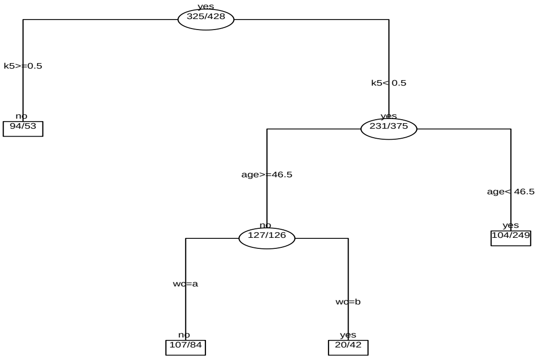
- After growing a full tree, we prune it back to avoid overfitting.
- Pruning involves collapsing some internal nodes to find a subtree without increasing the overall cost substantially.
- We can find the cross-validation costs for all possible subtrees, T, and choose the subtree with the lowest *cost-complexity* value defined as

$$C_\alpha(T) = \sum_{m=1}^{|T|} n_m C_m + \alpha|T|$$

where $|T|$ is the number of terminal nodes for the subtree; n_m and C_m are the number of observations and cost for the mth terminal node; $\alpha > 0$ is a tuning parameter.

US Women's labor-force participation

- The dataset `Mroz` in the package `car` includes the work status of 753 women along with 7 other variables
- We obtain the following model for predicting US women's work status after growing and pruning a tree model



Pros and Cons of Trees

- Prose
  - Easy to interpret
  - Perform automatic variable selection
  - Automatically captures interactions and nonlinear relationships
  - Robust to outliers
- Cons
  - Low predictive power
  - High variability

Random Forests

- *Random forests* models (Breiman, 2001) attempt to reduce variance and improve predictive power by using *bagged* and *de-correlated* (decoupled) trees
- A random forests model is an ensemble of trees, each developed based on a random subset of input variables (features) and a [bootstrap] sample of observations (i.e., randomly selected observations with replacement).
- For regression, we average over the estimates from individual trees.
- For classification, each tree casts a vote, and we choose the class with the highest vote.

Nearest Neighbor Methods

Background

- Given a training set (x, y) of size n, we want to predict the response value, $\tilde{y}$, of a test case (i.e., a future observation) with input values $\tilde{x}$
- So far, we have discussed models that build a map between the input and response variable using some parameters, θ, after which we can forget the training set and use the resulting map to estimate $\tilde{y}$ given $\tilde{x}$
- A possible issue with these method is that they rely on strong assumptions (e.g., linearity, normality), which could be unrealistic
- Alternatively, we can avoid making strong assumptions and build memory-based models that remember the original training set and use it for predicting $\tilde{y}$ directly
- To this end, we can find training cases that are close to the test case in the input space and use their average y (or median, or mode) as our estimate of $\tilde{y}$

K-nearest neighbor

- To achieve this, we need a metric to measure closeness and specify k, the number of observations with the closest distance to the test case
- We commonly use Euclidean distance to measure closeness
- For each test case, after measuring its distance to all training cases and identifying the neighborhood N_k with the top k observations (with smallest distance to the test case), we estimate its response value as follows:

$$\hat{y}(\tilde{x}) = \frac{1}{k} \sum_{i \in N_k(x)} y_i$$

- We can use the same approach for regression, with a numerical response variable, and classification, where y is an indicator variable

A binary classification problem with $k = 15$

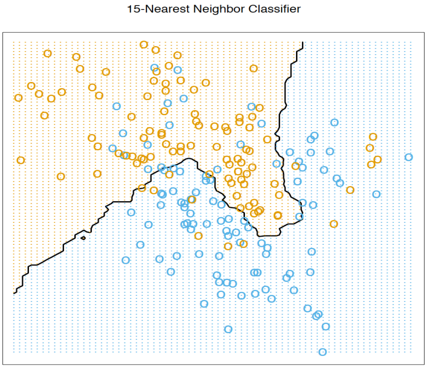


Figure: Figure 2.2 Hastie et. al. (2010).

A binary classification problem with $k = 1$

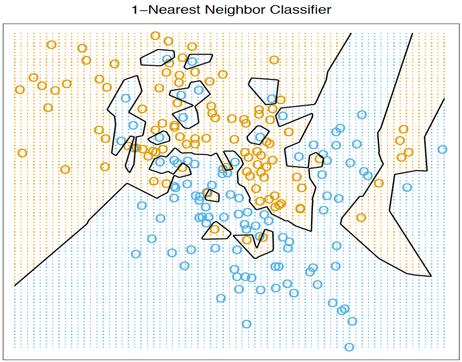


Figure: Figure 2.3 Hastie et. al. (2010).

Setting k

- As we can see in this example, the results could be very sensitive to the choice of the tuning parameter k
- With small values of k, we run the risk of overfitting; with large values of k, we average over cases far from the test case
- As usual, we can use cross-validation or data splitting strategy to set this tuning parameter
- Note that although there seems to be a single parameter k, the *effective* number of parameters is n/k, i.e., the number of means we have to estimate assuming the neighborhoods are not overlapping
- Finally, note that this approach assigns 0-1 weights to the training cases; in future, we will discuss *kernel* methods where the weights are a function of distance and go smoothly to zero as distance increases

Gaussian Process Models

Introduction

- In this lecture, we discuss Gaussian process for regression.
- To learn more about this topic, refer to “Regression and classification using Gaussian process priors” (with discussion), by Neal, R. M. (1998).
- Gaussian process can be used as a distribution over functions $y = f(x)$.
- Note that this is a stochastic function, i.e., it includes a noise term, so even if $x_1 = x_2$, $f(x_1)$ may not be same as $f(x_2)$ in general.

Gaussian process models

- To introduce this concept, we start with a simple linear regression model.
- Recall that we presented a linear regression model as

$$y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + ... \beta_p x_{ip} + \epsilon_i$$

- Using normal priors (with mean zero, and in general, different variances) for β's

$$\beta_j | \sigma_j \sim N(0, \sigma_j^2) \qquad j = 0, ..., p$$

Gaussian process models

- In prior, β has a $(p+1)$ dimensional multivariate normal distribution

$$\beta | \Sigma_\beta \sim N(0, \Sigma_\beta)$$

- ϵ also has an n dimensional multivariate normal distribution

$$\epsilon | \Sigma_\epsilon \sim N(0, \Sigma_\epsilon)$$

- To obtain the distribution of y we multiply β by the matrix x and add ϵ to it.
- Based on the properties of multivariate normal distribution, the resulting distribution would still be multivariate normal $N(0, C)$ where

$$C = x\Sigma_\beta x^T + \Sigma_\epsilon$$

Gaussian process models

- This gives us the prior distribution on the function $y(x)$.
- Since any finite subset of $y(x)$ (e.g., for the n observed cases) would have a Gaussian distribution, the prior distribution on $y(x)$ is a *Gaussian process*.
- Similar to the Gaussian distribution, the Gaussian process is also defined by its mean (here, the mean is 0 in prior) and its covariance function C.
- For the above linear model, the elements of C are

$$C_{ij} = Cov(y_i, y_j) = \sigma_0^2 + \sum_{u=1}^{p} x_{iu} x_{ju} \sigma_u^2 + \delta_{ij} \sigma_\epsilon^2$$

where δ_{ij} is equal to 1 if $i = j$, and 0 otherwise.

Gaussian process models

- Setting up the model this way, we are putting the prior directly on the relationship between x and y as opposed to on some parameters that represent this relationship (i.e., we cut out the middleman).
- This is specially useful if our objective is to predict future cases as opposed to making inference about the relationship between x and y.
- Note that the prior here is implicit and reflects our choice of the functional form.
- In the above example, we are assuming the relationship is linear. In general, we could use other covariance functions, C, to create nonlinear relationship.

Gaussian process for nonlinear regression

- For example, the following covariance function is very useful and includes a wide range of smooth nonlinear functions:

$$Cov(y_i, y_j) = \lambda^2 + \eta^2 \exp\left(- \sum_{u=1}^{p} \rho_u^2 (x_{iu} - x_{ju})^2 \right) + \delta_{ij}\sigma_\epsilon^2$$

- The constant part is used to make sure the model fit functions where the mean of y is not zero (the x matrix does not have a vector of 1's anymore). However, it is better to center y before analysis so we don't have to use a large constant.
- There is one ρ for each predictor.
- The noise parameter, σ_ϵ^2 (also called *jitter*), is essential to improve the computation.
- Within the Bayesian framework, we usually put hyper-priors on the hyper-parameters λ, η, ρ, and σ.

The effect of parameters in the covariance function

- By using different η, ρ's, λ and σ_ϵ, we can generate a large variety of functions.

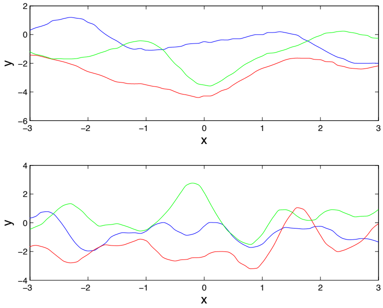


Figure: The top panel shows samples based on $\eta = 1$, $\rho = 1$, $\lambda = 1$, and $\sigma_\epsilon = 0.01$. The bottom panel is base on the same priors except we set $\rho = 2$.

Prediction

- As mentioned above, using a Gaussian process prior is especially useful if our goal is predicting future cases for which we only know the value of predictors, $\tilde{x}$.
- Assume that we have observed (x, y) for n cases, and we want to predict $\tilde{y}$ for a new observation with predictor values $\tilde{x}$.

Prediction

- Since the covariance function depends on x, we can find C_{n+1} for n the training cases and the new observation, i.e., for $\binom{x}{\tilde{x}}$. To avoid confusion we denote the covariance matrix for just the training cases as C_n.
- We can write down C_{n+1} as follows:

$$C_{n+1} = \left(\begin{array}{cc} C_n & K \\ K^T & v \end{array} \right)$$

where K is the $n \times 1$ covariance vector between $\tilde{y}$ and the n observed y. v is the prior variance of $\tilde{y}$ obtained based on the covariance function C.

Prediction

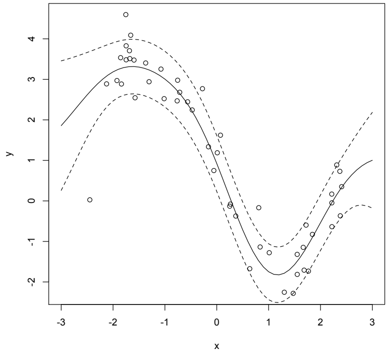
- Based the above setting, we can obtain the posterior predictive distribution for the new case.
- This distribution is also Gaussian with the following mean and variance:

$$\begin{aligned} E(\tilde{y}|y) &= K^T C_n^{-1} y \\ Var(\tilde{y}|y) &= v - K^T C_n^{-1} K \end{aligned}$$

- If we need a point estimate, we can use $E(\tilde{y}|y)$.

Example

- The following example shows a Gaussian process model trained on 100 data points uniformly sampled from -2 to 2 .



Example

- For the above model, we used the following covariance function:

$$Cov(y_i, y_j) = 2 + \exp\big(-0.5(x_i - x_j)^2\big) + \delta_{ij} \times 0.1$$

- The solid line is expected function based on a grid test points between -3 and 3.
- The dashed lines show the 95% interval for predictions.

Neural Networks

Neural Networks (NN)

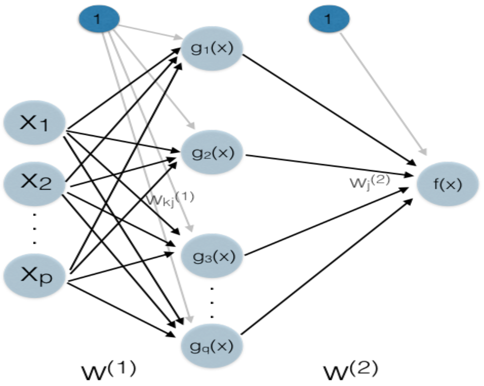


Figure: Multilayer perceptron (MLP) with p input variables, one hidden layer with q hidden units, and a single output. Here, $w^{(1)}$ represents the connection weight matrix between the input layer and the hidden layer, and $w^{(2)}$ is the vector of connection weights between the hidden layer and the output.

Neural Networks (NN)

- A multilayer perceptron (MLP, aka feedforward NN) is comprised of an input layer, output layer and a number of hidden layers in between
- The hidden layers creates a set of basis by applying nonlinear transformations, g, to their input and pass their results to the next layer until we reach the output layer.
- We refer to g as the activation or transfer function, which is usually set to the sigmoid (aka logistic) function

$$\mathrm{sigm}(a) = \frac{1}{1 + e^{-a}}$$

or the hyperbolic tangent function

$$\tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}}$$

Neural Networks (NN)

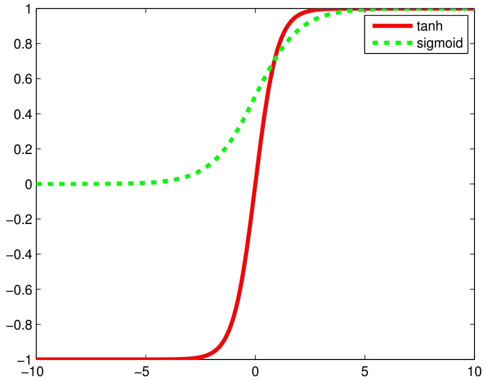


Figure: Figure 16.6 in Murphy (2012).

Neural Networks (NN)

- The output $f(x)$, which is a function used to approximate y, is a linear combination of basis defined by the hidden layers
- For linear regression models (continuous outcome)

$$P(y|x, w) = N(y|f(x), \sigma^2)$$

- For logistic regression models (binary outcome)

$$P(y|x, w) = \mathrm{Ber}(y|\mathrm{sigm}(f(x)))$$

- For multiple categories, we use the multinomial logit model, which is also known as the softmax function

Neural Networks (NN)

- For a MLP with one hidden layer and tanh activation function, we have

$$\begin{aligned} g_j(x) &= \tanh[w_{0j}^{(1)} + \sum_{k=1}^{p} w_{kj}^{(1)} x_k], \qquad \text{for } j = 1, \ldots, q \\ f(x) &= w_0^{(2)} + \sum_{j=1}^{q} w_j^{(2)} g_j(x) \end{aligned}$$

- Here w_0's, which play the role of the intercept in regression models, are called biases

Learning

- To train a MLP, we first need to specify the negative log-likelihood, which is also known as the energy function, E
- For regression, we have (squared error)

$$E = \frac{1}{2\sigma^2} \sum_{i=1}^{n} (y_i - f(x_i))^2$$

for binary classification models, we have (cross entropy)

$$E = -\sum_{i=1}^{n} y_i[\log \mathrm{sigm}(f(x_i))] + (1 - y_i)[\log(1 - \mathrm{sigm}(f(x_i)))]$$

- To estimate the weights, we minimize the energy function with respect to w

Learning

- The parameters of a neural network model are not identifiable
    - Permuting the order of hidden units does not change the model
    - If we change the sign of weights entering a hidden unit, the model remains the same as long as we also change the sign of the weights going out of that unit since $\tanh(-a) = -\tanh(a)$
- Also, in general the energy function for MLP is non-convex
- Nevertheless, we can still use common iterative optimization methods (e.g., gradient descent algorithms) to obtain locally optimal estimates
- Using the chain rule, it is easy to find the gradient

Backpropagation

- For learning (i.e., estimating the weights), we start with initializing the weights (including the biases) to some random numbers (all different values) and iteratively perform the following steps
  - At each iteration, we use *forward propagation* to find the values going to each unit, before and after transformation, until we reach the output layer
  - We find the derivatives of E with respect to each unit starting from the output and *backpropagate* using the chain rule to find the derivatives with respect to hidden units
  - Using the chain rule again, we find the derivatives with respect to the weights
  - We then update the weights by moving in the direction of the negative gradient (see the notes on optimization)
  - We repeat the above steps until some stopping criterion is reached

Backpropagation

- For a MLP with one hidden layer and tanh activation function, given the current weights, forward propagation involves finding the following values at each hidden unit before and after transformation

$$\begin{aligned} z_j &= w_{0j}^{(1)} + \sum_{k=1}^{p} w_{kj}^{(1)} x_k \\ g_j &= \tanh(z_j) \end{aligned}$$

- For the output unit, we hve

$$f = w_0^{(2)} + \sum_{j=1}^{q} w_j^{(2)} g_j$$

Backpropagation

- Backpropagation starts with finding $\partial E/\partial f$; For regression model

$$\frac{\partial E}{\partial f} = -\frac{1}{\sigma^2}\sum_{i=1}^{n}(y - f(x_i))$$

- Next, using the chain rule we have

$$\frac{\partial E}{\partial g_j} = \frac{\partial E}{\partial f}\frac{\partial f}{\partial g_j} = w_j^{(2)}\frac{\partial E}{\partial f}$$

- We then find the derivatives with respect to z_j

$$\frac{\partial E}{\partial z_j} = \frac{\partial E}{\partial g_j}\frac{\partial g_j}{\partial z_j} = (1 - g_j^2)\frac{\partial E}{\partial g_j}$$

Recall that $\frac{d}{da}\tanh(a) = 1 - \tanh^2(a)$

Backpropagation

- Finally, we find the derivatives with respect to the weights
- For connection weights between the hidden layer and output we have

$$\frac{\partial E}{\partial w_j^{(2)}} = \frac{\partial E}{\partial f} \frac{\partial f}{\partial w_j^{(2)}} = g_j \frac{\partial E}{\partial f}$$

  Note that $g_0 = 1$ when evaluating $\partial E / \partial w_0^{(2)}$

- For the weights connecting the input layer to the hidden layer we have

$$\frac{\partial E}{\partial w_{kj}^{(1)}} = \frac{\partial E}{\partial z_j} \frac{\partial z_j}{\partial w_{kj}^{(1)}} = x_k \frac{\partial E}{\partial z_j}$$

  $x_0 = 1$ when evaluating $\partial E / \partial w_0^{(1)}$

Backpropagation

- After we find the gradient ∇E, we update the parameters by taking a step in a direction of negative gradient, with stepsize t

$$\begin{aligned} \Delta w &= -\nabla E \\ w &\leftarrow w + t\Delta w \end{aligned}$$

- The stepsize (aka learning rate) is found by trial-and-error
- We could run the algorithm until the approximation error falls below a desired threshold; however, this could lead to *overfitting*
- Two common strategies to avoid this issue are *early stopping* and *weight decay*

Early stopping

- Overfitting occurs when the model performs well on the training data and performs poorly on the test (future) data
- In the "early stopping" method, we start with some initial weights close to zero and monitor the performance of the neural network model throughout the training process based on an independent *validation set* (usually 20% of the data; this is separate from any test set used for model evaluation); we stop the algorithm when the model's performance on the validation set starts to decline substantially (a sign of overfitting)
- We can use the prediction error or average log probability on the validation set as a measure of performance
- This method could be very successful for avoiding overfitting; however it is *ad hoc* and wasteful since some of the data points are not used in the training directly

Weight decay

- Alternatively, to avoid overfitting, we can penalize models agains complexity (similar to ridge regression and Lasso)
- To this end, instead of minimizing the energy function, we minimize the penalized version of it by adding the following penalty terms:

$$\lambda_1 \sum_{k=1}^{p} \sum_{j=1}^{q} [w_{kj}^{(1)}]^2 + \lambda_2 \sum_{j=1}^{q} [w_j^{(2)}]^2$$

- This is known as "weight decay" since it shrinks the weights towards zero to encourage simpler models
- To set the values of λ_1 and λ_2, we can use an independent validation set as before or use cross-validation when the sample size is small

Illustrative Example

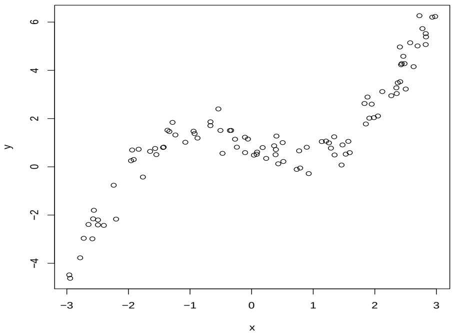


Figure: Observed data

Illustrative Example

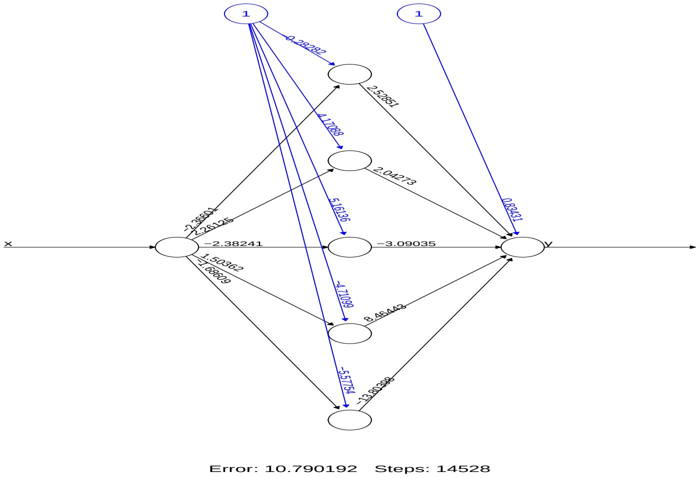


Figure: A neural network with 5 hidden units

Illustrative Example

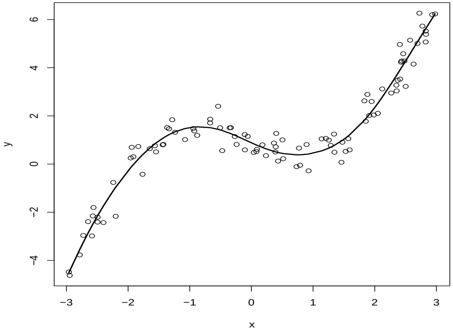


Figure: Estimated function, $f(x)$, with 5 hidden units

Illustrative Example

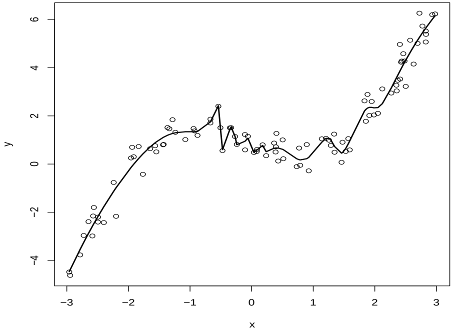


Figure: Estimated function, $f(x)$, with 50 hidden units

Illustrative Example

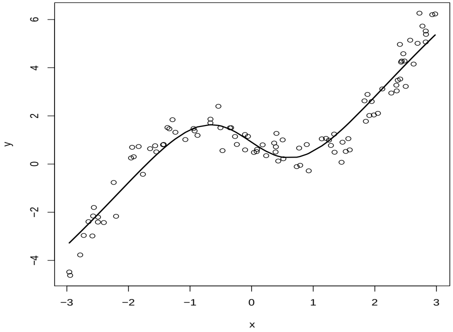


Figure: Estimated function, $f(x)$, with 50 hidden units using weight decay