Johns Hopkins
Whiting School of Engineering

Applied and Computational Mathematics

# Data Mining
# 625.740
## Linear Discrimination

Mike Weisman

*email:* `data.mining.625.740@gmail.com`

JOHNS HOPKINS
WHITING SCHOOL of ENGINEERING

Applied and Computational Mathematics

# Data Mining

## 625.740

### Linear Discrimination

# Discriminant Based Classification

Recall that we defined a set of discrimination functions $g_j(\mathbf{x}), j = 1, \cdots, K$ and we choose class $\mathscr{C}_j$ if $g_j(\mathbf{x}) = \max_i g_i(\mathbf{x})$

We defined the discriminant functions in terms of posterior probabilities, for example

$$g_j(\mathbf{x}) = \log P(\mathscr{C}_j|\mathbf{x})$$

This is called likelihood-based classification. Discriminant based classification assumes a model for the discriminant without estimating likelihoods or posterior probabilities.

In this module, we will be focusing on linear discriminants:

$$g_j(\mathbf{x}|w_{j0}, w_{j1}, \cdots, w_{jd}) = \sum_{k=1}^{d} w_{jk} x_k + w_{j0}$$

# Perceptron

Artificial neural networks are an attempt to model the computations of the brain. The primary building block of neural networks is the perceptron. The perceptron is a basic processing element whose inputs may come from the environment or may be outputs of other perceptrons.

Perceptrons and neural networks give a convenient framework for implementing linear discrimination.

# Perceptron definition

Associated with each input $x_j \in R, \quad j = 1 \cdots d$, is a connection weight $w_j \in R$. In the simplest case, the output, $y$, is a weighted sum of the inputs

$$y = \sum_{j=1}^{d} w_j x_j + w_0$$

$w_0$ is the intercept value and is modeled as coming from a bias unit $x_0 = 1$.

$$w = [w_0, w_1, \cdots, w_d]^T$$
$$x = [1, x_1, \cdots, x_d]^T$$

are the augmented vectors

$$y = w^T x.$$

# The Perceptron

# Regression Learning

During testing, with given weights $w$, for input $x$, we compute the output $y$. The weights are learned to produce the correct output for a given input.
When $d = 1$ we have the line

$$y = wx + w_0.$$

Thus, a perceptron with one input and one output could be used to implement a linear fit. With more than one input, we have a hyperplane, the perceptron can be used to implement a multivariate linear fit. Given a sample, the weights $w_j$ can be found by regression.

# Linear discriminant function

The perceptron defines a hyperplane and thus could be used to divide input space into two half-spaces using one perceptron to implement a linear discriminant function, it can separate two classes by checking the sign of the output. Let us define a threshold function

$$s(a) = \begin{cases} 1, & a \geq 0 \\ 0, & \text{otherwise.} \end{cases}$$

If a hyperplane $w^T x = 0$ can be found that separates the classes $C_1 (a \geq 0)$ & $C_2 (a < 0)$, then our decision is:

$$\begin{aligned} &\text{choose } C_1, && \text{if } s(w^T x) > 0, \\ &\text{choose } C_2, && \text{otherwise.} \end{aligned}$$

# Linear discriminant function

If we need a posterior probability (for example, to calculate risk), we can use the sigmoid function at the output:

$$o = w^T x,$$

$$y = \mathrm{sigmoid}(o) = \frac{1}{1 + \exp\{-w^T x\}}.$$

# Multiple Perceptrons

When there are $K \geq 2$ outputs, there are $K$ perceptrons, each of which has a weight vector $w_i$

$$y_i = \sum_{j=1}^{d} w_{ij} x_i + w_{io} = w_i^T x$$

$$y = Wx$$

$W$ is a $K \times (d+1)$ weight matrix of $w_{ij}$ whos rows the weight vectors for each of the $K$ perceptrons.
When used for classification during testing, choose $C_i$ if $y_i = \max_n y_n$.

# Multiple Perceptrons

In a neural network, the value of each perceptron is a local function of its inputs and synaptic weights. In classification, if we need posterior probabilities and not just the winner class, we need values of other outputs.

# Polynomial Approximation and Non-linear Discrimination

The linear model can be used for polynomial approximation or to perform non-linear discrimination.

Any linear discrimination method can be used to calculate $w_i$, $\quad (i = 1 \cdots k)$, off-line.
Afterwards, the weights, $w_i$, can then be plugged into the network.

# Dimensionality Reduction

The equation $y = Wx$ defines a linear transformation from a $d-$dimensional space to a $k-$dimensional space and can be used for dimensionality reduction if $k < d$. One can calculate $W$ off-line, for example, with PCA, and then use perceptrons to implement the transformation.

# Dimensionality Reduction

The equation $y = Wx$ defines a linear transformation from a $d-$dimensional space to a $k-$dimensional space and can be used for dimensionality reduction if $k < d$. One can calculate $W$ off-line, for example, with PCA, and then use perceptrons to implement the transformation. We can then implement regression or classification in the new space. In this case, we (almost) have a two-layer network!

1. The first layer implements the linear transformation.
2. The second layer implements linear regression or classification in the new space.

Because both are linear transformations, they can be combined and written as a single layer. Later we will see that combing a nonlinear first layer with a second layer results in a two layer network.

# Training

Training a perceptron implements and defines a hyperplane.

Given a data sample $\mathscr{X} = \{x_k\}$, the weight values $\mathscr{W} = \{w_k\}$ can be calculated offline then later "plugged in" to the perceptron to calculate output values.

# Training

Training a perceptron implements and defines a hyperplane.
Given a data sample $\mathscr{X} = \{x_k\}$, the weight values $\mathscr{W} = \{w_k\}$ can be calculated offline then later "plugged in" to the perceptron to calculate output values.

**Advantages of this approach:**

1. Cost savings because we need not store the training sample and intermediate results obtained during optimization.
2. Can handle time varying problem, for example, the sample distribution may be changing.
3. The system itself may change, for example, components wearing out or sensors degrading over time.

# Gradient Descent

Given the Error function $E = E(\mathbf{w}|\mathscr{X})$ The gradient is defined as

$$\Delta E = \frac{\partial E}{\partial w} = \left(\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \cdots, \frac{\partial E}{\partial w_d}\right)^T$$

To implement gradient descent we start with random **w** and at each step update **w** by moving in the direction opposite the gradient.

$$\Delta w_j = -\eta \frac{\partial E}{\partial w_j}, \forall j$$

$$w_j = w_j + \Delta w_j$$

where $\eta$ is the learning factor which gradually decreases over time for convergence.

# Improving Convergence

## Momentum

Here the previous update contributes to the current update via

$$\Delta w_j(k) = -\eta \frac{\partial E(k)}{\partial w_j} + \alpha \Delta w_j(k-1)$$

where $\alpha$ is typically between 0.5 and 1.0.
For example, initially, we may take $\alpha = 0.9$ and gradually lower $\alpha$ to 0.5.

## Adaptive Learning

The learning factor, $\eta$, generally takes a value between 0.0 and 1.0. To achieve faster convergence, we can adaptively control $\eta$, increasing *eta* when the updated energy is lower than the average over the past $N$ updates, and decreasing $\eta$ if the energy increases over the windowed average.

$$\Delta\eta = \begin{cases} \alpha, & \text{if } E(k) < \frac{1}{N}\sum_{q=1}^{N} E(k-q) \\ -\beta\eta, & \text{otherwise} \end{cases}$$

# Online Learnng

We write the error function over individual instances, not over the whole sample.

- Start with random initial weights
- At each iteration adjust parameters to minimize error (without forgetting what we have learned)

If the error function is differentiable, we can use gradient descent. For example, if we are interested in linear regression:

Let a single instance pair be $(\mathbf{x}_\alpha, r_\alpha)$.

$$\mathrm{Error}_\alpha = E_\alpha(\mathbf{w}|\mathbf{x}_\alpha, r_\alpha) = \frac{1}{2}(r_\alpha - y_\alpha)^2 = \frac{1}{2}\left[r_\alpha - \mathbf{w}^T\mathbf{x}_\alpha\right]^2$$

For $j = 0, \cdots, d$, the online update is

$$\Delta w_{\alpha,j} = \eta(r_\alpha - y_\alpha)x_{\alpha,j}$$

Gradient descent with at one data point, rather than over the full range of data, is known as stochastic gradient descent.

# Example: Logistic Regression Classification

The data for two classes is given as $(\mathbf{x}_\alpha, r_\alpha)$ with:

$$r_{\alpha,i} = 1, \text{ when } \mathbf{x}_\alpha \in C_1$$
$$r_{\alpha,i} = 0, \text{ when } \mathbf{x}_\alpha \in C_2$$

The output is

$$y_\alpha = \text{sigmoid}(\mathbf{w}^T \mathbf{x}_\alpha)$$

and

$$E_\alpha(\mathbf{w}|\mathbf{x}_\alpha, r_\alpha) = -r_\alpha \log y_\alpha - (1 - r_\alpha) log(1 - y_\alpha)$$

For $j = 0, \cdots, d$, the online update via stochastic gradient descent is

$$\Delta w_j = \eta(r_\alpha - y_\alpha) x_{\alpha,j}$$

# Example: Logistic Regression Classification

With $K > 2$ classes, for each of $(\mathbf{x}_\alpha, r_\alpha)$,

$$r_{\alpha,i} = 1, \text{ when } \mathbf{x}_\alpha \in C_i$$
$$r_{\alpha,i} = 0, \text{ otherwise.}$$

The output is

$$y_{i,\alpha} = \frac{\exp(\mathbf{w}_i^T \mathbf{x}_\alpha)}{\sum_k \exp(\mathbf{w}_k^T \mathbf{x}_\alpha)}$$

and

$$E_\alpha(\mathbf{w}|\mathbf{x}_\alpha, r_\alpha) = -\sum_k r_{k,\alpha} \log y_{k,\alpha}$$

For $i = 1, \cdots, K, \text{ and } j = 0, \cdots, d$, with each $\mathbf{x}_\alpha$, the online update via stochastic gradient descent is

$$\Delta w_{i,j} = \eta(r_{i,\alpha} - y_{i,\alpha}) x_{\alpha,j}$$

# Learning Boolean Functions: AND

# Learning Boolean Functions: OR

# Learning Boolean Functions: `XOR`

Applied and Computational Mathematics

# Data Mining

## 625.740

### Linear Discrimination