

SDM5013:

Deep Learning and Reinforcement Learning

Zhiyun Lin



南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY



设计智造学院
School of
System Design and
Intelligent Manufacturing

Lecture: Linear Classification

- ❑ Classification Problems
- ❑ Binary Classification with Linear Regression
- ❑ Metrics to Evaluate Classification
- ❑ Logistic Regression
- ❑ Multi-class Classification

Recap: Regression

❑ Regression: to predict continuous outputs t

- Consider proper **features** (inputs): x (or \mathbf{x} if vectors)
- **Training examples**, many $x(i)$ for which $t(i)$ is known (labeled)
- A **model**, a function that represents the relationship between x and t

$$y = f(x, w)$$

- A **loss** or a **cost** or an **objective** function, which tells us how well our model approximates the training examples
- **Optimization**, a way of finding the parameters w of our model that minimizes the loss function

Cat or dog?

- ❑ Machine learn to classify a cat or dog
- ✓ Offer a large number of pictures with cat or dog labels and train a model
- ✓ Use the trained model to predict where there is a cat or a dog



Is this a dog?

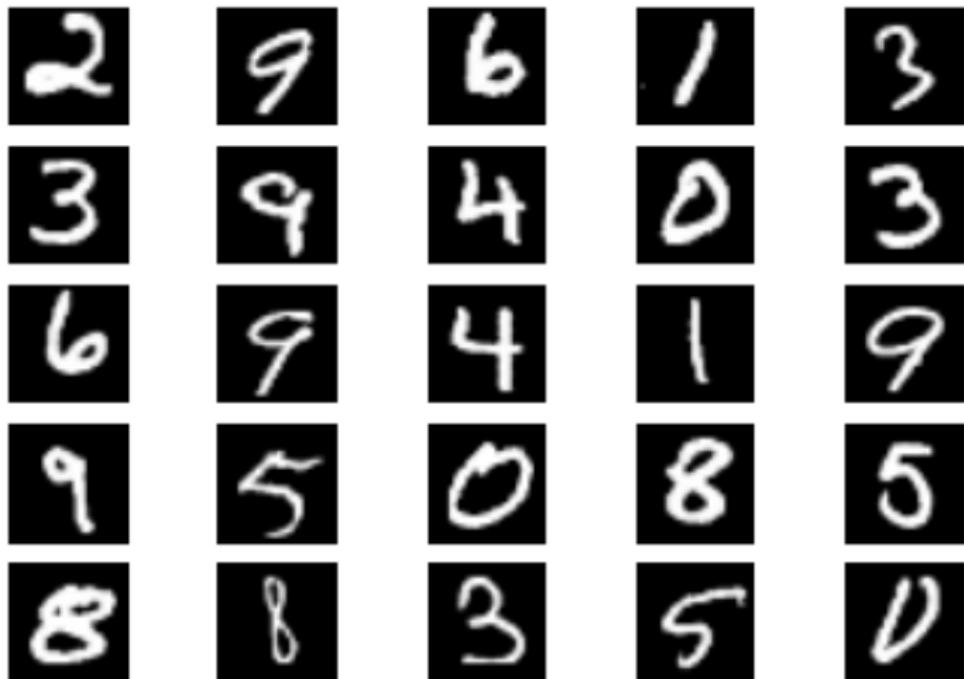


Am I going to pass the exam?



Am I going to pass the exam?

Examples of Problems



What digit is this?

How can I predict this? What are my input features?

Classification

What do all these problems have in common?

Categorical **outputs**, called **labels**

(eg, yes/no, dog/cat/person/other)

Assigning each input vector to one of a finite number of labels is called **classification**

Binary classification: two possible labels (eg, yes/no, 0/1, cat/dog)

Multi-class classification: multiple possible labels

We will first look at binary problems, and discuss multi-class problems later in class

Classification vs Regression

We are interested in mapping the input $\mathbf{x} \in \mathcal{X}$ to a *label* $t \in \mathcal{Y}$

In regression typically $\mathcal{Y} = \mathbb{R}$

Now \mathcal{Y} is categorical

Classification as Regression

Can we do this task using what we have learned in previous lectures?

Simple hack: Ignore that the output is categorical!

Suppose we have a binary problem, $t \in \{-1, 1\}$

Assuming the standard model used for (linear) regression

$$y(\mathbf{x}) = f(\mathbf{x}, \mathbf{w}) = \mathbf{w}^T \mathbf{x}$$

How can we obtain \mathbf{w} ?

Use least squares, $\mathbf{w} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{t}$. How is \mathbf{X} computed? and \mathbf{t} ?

Which loss are we minimizing? Does it make sense?

$$\ell_{square}(\mathbf{w}, t) = \frac{1}{N} \sum_{n=1}^N (t^{(n)} - \mathbf{w}^T \mathbf{x}^{(n)})^2$$

How do I compute a label for a new example? Let's see an example

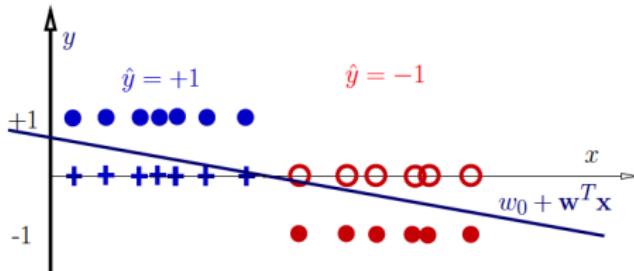
Classification as Regression

One dimensional example (input x is 1-dim)



The colors indicate labels (a blue plus denotes that $t^{(i)}$ is from the first class, red circle that $t^{(i)}$ is from the second class)

Decision Rules



Our classifier has the form

$$f(\mathbf{x}, \mathbf{w}) = w_o + \mathbf{w}^T \mathbf{x}$$

A reasonable **decision rule** is

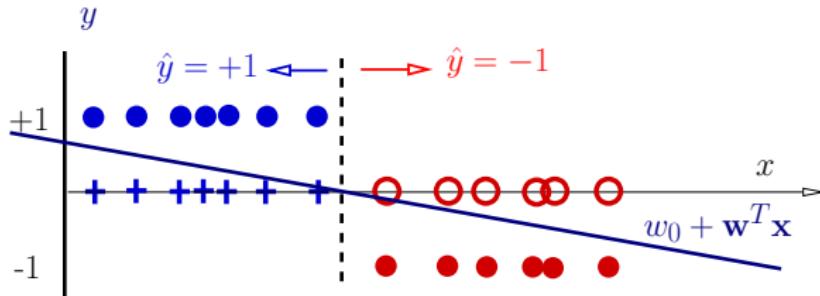
$$y = \begin{cases} 1 & \text{if } f(\mathbf{x}, \mathbf{w}) \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

How can I mathematically write this rule?

$$y(\mathbf{x}) = \text{sign}(w_0 + \mathbf{w}^T \mathbf{x})$$

What does this function look like?

Decision Rules



How can I mathematically write this rule?

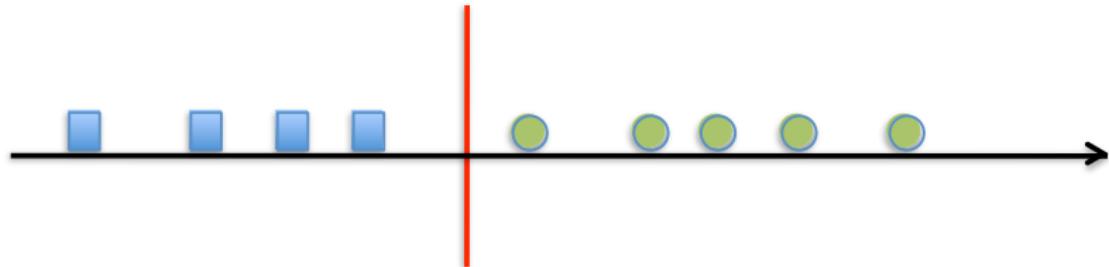
$$y(\mathbf{x}) = \text{sign}(w_0 + \mathbf{w}^T \mathbf{x})$$

This specifies a [linear classifier](#): it has a [linear boundary \(hyperplane\)](#)

$$w_0 + \mathbf{w}^T \mathbf{x} = 0$$

which separates the space into two "half-spaces"

Example in 1D



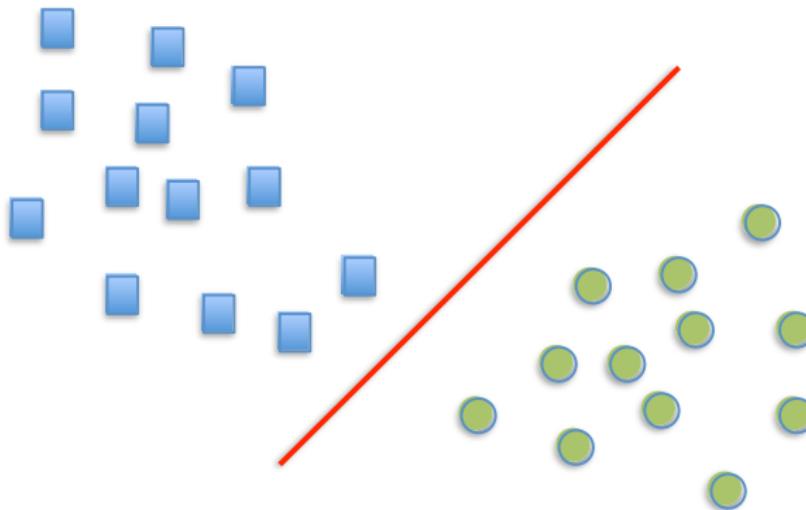
The linear classifier has a linear boundary (hyperplane)

$$w_0 + \mathbf{w}^T \mathbf{x} = 0$$

which separates the space into two "half-spaces"

In 1D this is simply a threshold

Example in 2D



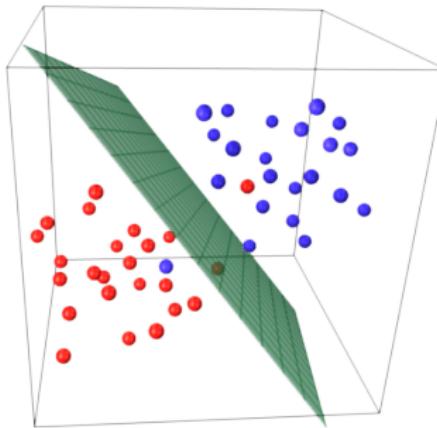
The linear classifier has a linear boundary (hyperplane)

$$w_0 + \mathbf{w}^T \mathbf{x} = 0$$

which separates the space into two "half-spaces"

In 2D this is a line

Example in 3D



The linear classifier has a linear boundary (hyperplane)

$$w_0 + \mathbf{w}^T \mathbf{x} = 0$$

which separates the space into two "half-spaces"

In 3D this is a plane

What about higher-dimensional spaces?

Geometry

$\mathbf{w}^T \mathbf{x} = 0$ a line passing through the origin and orthogonal to \mathbf{w}
 $\mathbf{w}^T \mathbf{x} + w_0 = 0$ shifts it by w_0

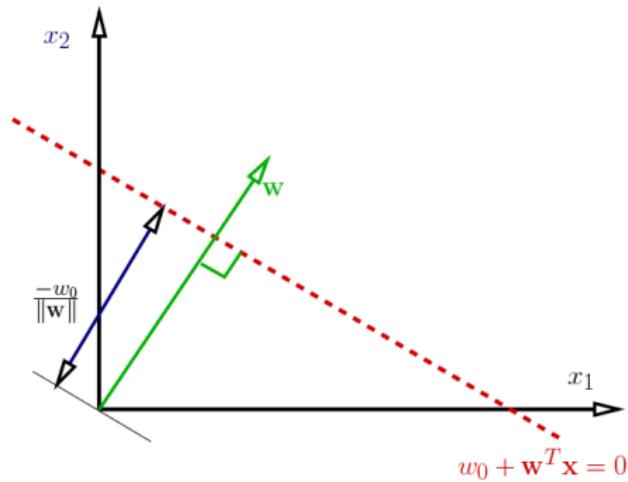


Figure from G. Shaknarovich

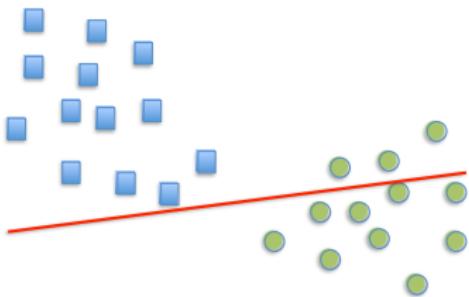
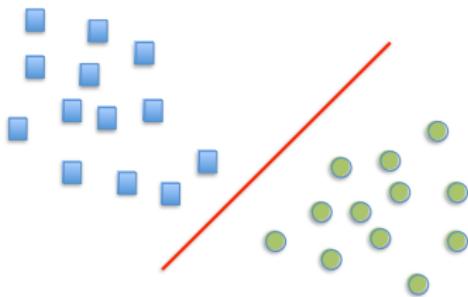
Learning Linear Classifiers

Learning consists in estimating a “good” decision boundary

We need to find \mathbf{w} (direction) and w_0 (location) of the boundary

What does “good” mean?

Is this boundary good?



We need a criteria that tell us how to select the parameters

Do you know any?

Loss functions

Classifying using a linear decision boundary reduces the data dimension to 1

$$y(\mathbf{x}) = \text{sign}(w_0 + \mathbf{w}^T \mathbf{x})$$

What is the cost of being wrong?

Loss function: $L(y, t)$ is the loss incurred for predicting y when correct answer is t

Loss functions

A possible loss to minimize is the zero/one loss

$$L(y(\mathbf{x}), t) = \begin{cases} 0 & \text{if } y(\mathbf{x}) = t \\ 1 & \text{if } y(\mathbf{x}) \neq t \end{cases}$$

Is this minimization easy to do? Why?

Other Loss functions

Zero/one loss for a classifier

$$L_{0-1}(y(\mathbf{x}), t) = \begin{cases} 0 & \text{if } y(\mathbf{x}) = t \\ 1 & \text{if } y(\mathbf{x}) \neq t \end{cases}$$

Asymmetric Binary Loss

$$L_{ABL}(y(\mathbf{x}), t) = \begin{cases} \alpha & \text{if } y(\mathbf{x}) = 1 \wedge t = 0 \\ \beta & \text{if } y(\mathbf{x}) = 0 \wedge t = 1 \\ 0 & \text{if } y(\mathbf{x}) = t \end{cases}$$

Squared (quadratic) loss

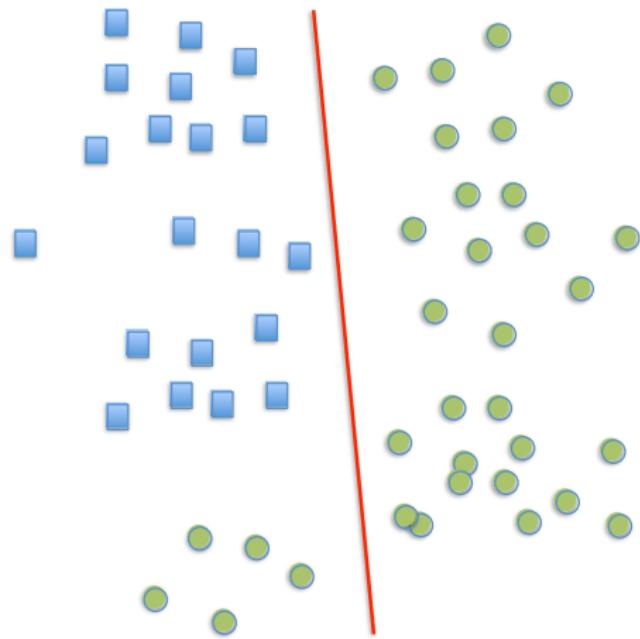
$$L_{\text{squared}}(y(\mathbf{x}), t) = (t - y(\mathbf{x}))^2$$

Absolute Error

$$L_{\text{absolute}}(y(\mathbf{x}), t) = |t - y(\mathbf{x})|$$

Can we always separate the classes?

If we can separate the classes, the problem is [linearly separable](#)



Can we always separate the classes?

Causes of non perfect separation:

- Model is too simple

- Noise in the inputs (i.e., data attributes)

- Simple features that do not account for all variations

- Errors in data targets (mis-labelings)

Should we make the model complex enough to have perfect separation in the training data?

Metrics

How to evaluate how good my classifier is? How is it doing on dog vs no-dog?



— TP (True Positive)

— FP (False Positive)

— FN (False Negative)

Metrics

How to evaluate how good my classifier is?

Recall: is the fraction of relevant instances that are retrieved

$$R = \frac{TP}{TP + FN} = \frac{TP}{\text{all groundtruth instances}}$$

Precision: is the fraction of retrieved instances that are relevant

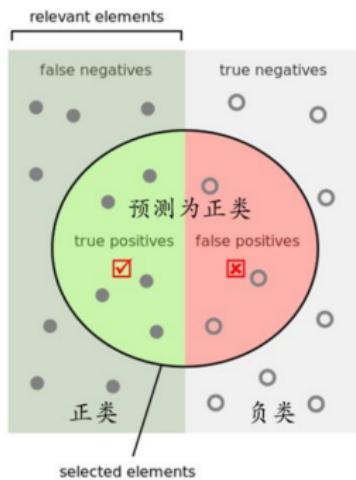
$$P = \frac{TP}{TP + FP} = \frac{TP}{\text{all predicted}}$$

F1 score: harmonic mean of precision and recall

$$F1 = 2 \frac{P \cdot R}{P + R}$$

More on Metrics

How to evaluate how good my classifier is?



How many selected items are relevant?

$$\text{Precision} = \frac{\text{true positives}}{\text{true positives} + \text{false positives}}$$



How many relevant items are selected?

$$\text{Recall} = \frac{\text{true positives}}{\text{true positives} + \text{false negatives}}$$



TP: positive examples classified as positive

FP: negative examples classified as positive

FN: positive examples classified as negative

TN: negative examples classified as negative

Metrics vs Loss

Metrics on a dataset is what we care about (performance)

We typically cannot directly optimize for the metrics

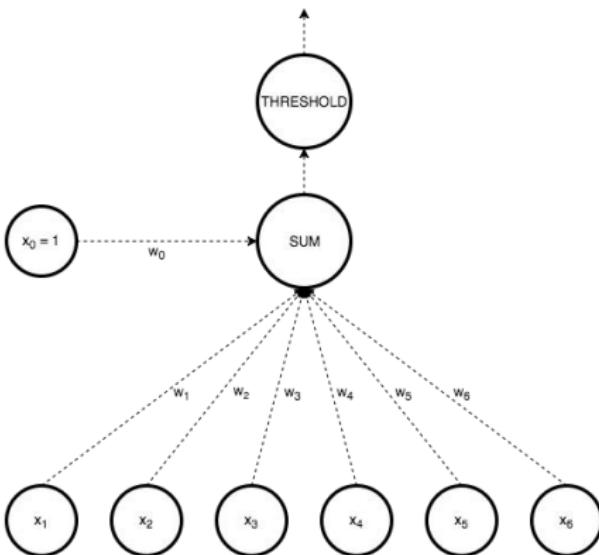
Our loss function should reflect the problem we are solving. We then hope it will yield models that will do well on our dataset

Linear Classifiers

- Linear classifiers classify data into labels based on a linear combination of input features. Therefore, these classifiers separate data using a line or plane or a hyperplane (a plane in more than 2 dimensions).

Perceptron

In Perceptron, we take weighted linear combination of input features and pass it through a threshold function which outputs 1 or 0.



Summary of Perceptron

Algorithm

- Let the input feature vector be

$$x = [x_1, x_2, \dots, x_n]^T, \quad \bar{x} = [1, x_1, x_2, \dots, x_n]^T,$$

- Let the weight vector be

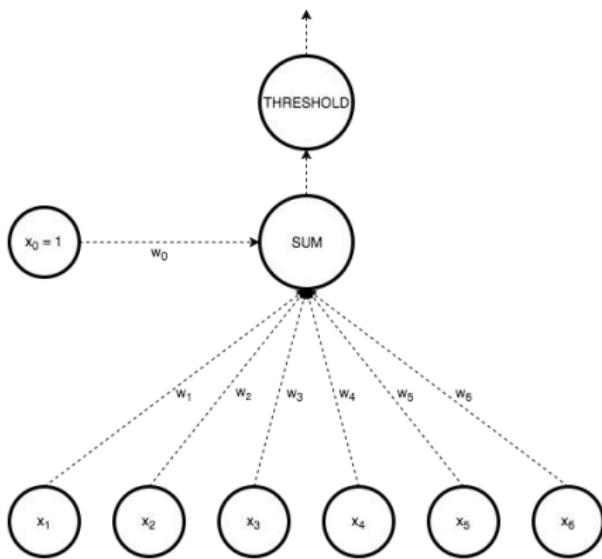
$$w = [w_0, w_1, w_2, \dots, x_n]^T$$

- Perceptron
- ✓ Transfer function

$$f(x, w) = \sum_{i=0}^n w_i x_i = w^T \bar{x}$$

- ✓ Activation function

$$\text{output} = \begin{cases} 1, & f(x, w) \geq 0 \\ 0, & f(x, w) < 0 \end{cases}$$



Intuition of Perceptron

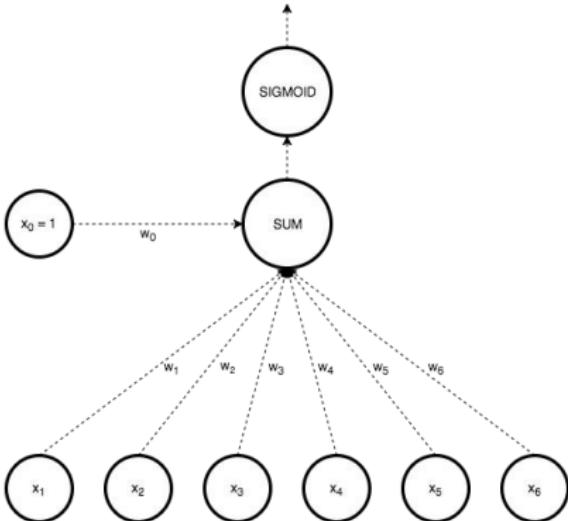
	Friend 1's opinion	Friend 2's opinion	Your Decision
week 1	Good	Good	You watch the movie and like it
week 2	Good	Bad	You watch the movie and do not like it
week 3	Bad	Good	Would you watch the movie

- Suppose your opinions about movies do not match with friend 1 multiple times, you are unlikely to ever consider his opinion to decide whether to watch a movie or not
- Thus every week you update the weights that you have assigned to each of your friend's opinion. The Perceptron algorithm works in a similar fashion

Linear Classifiers

❑ Logistic Regression

- take weighted linear combination of input features and pass it through a sigmoid function, which outputs a number between 1 and 0.
- Unlike perceptron, which just tells us which side of the plane the point lies on, logistic regression gives a probability of a point lying on a particular side of the plane.
- The probability of classification will be very close to 1 or 0 as the point goes far away from the plane. The probability of classification of points very close to the plane is close to 0.5



Logistic Regression

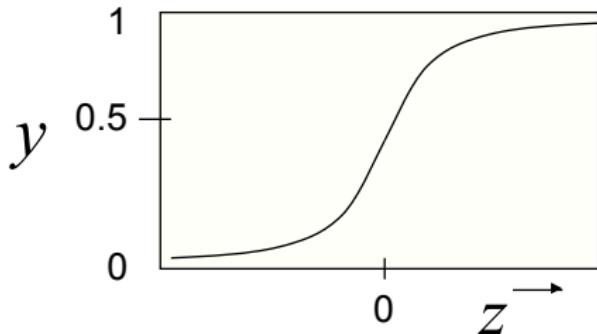
An alternative: replace the $\text{sign}(\cdot)$ with the **sigmoid** or **logistic function**

We assumed a particular functional form: sigmoid applied to a linear function of the data

$$y(\mathbf{x}) = \sigma(\mathbf{w}^T \mathbf{x} + w_0)$$

where the sigmoid is defined as

$$\sigma(z) = \frac{1}{1 + \exp(-z)}$$



The output is a smooth function of the inputs and the weights. It can be seen as a **smoothed and differentiable** alternative to $\text{sign}(\cdot)$

Logistic Regression

We assumed a particular functional form: sigmoid applied to a linear function of the data

$$y(\mathbf{x}) = \sigma(\mathbf{w}^T \mathbf{x} + w_0)$$

where the sigmoid is defined as

$$\sigma(z) = \frac{1}{1 + \exp(-z)}$$

- ▶ One parameter per data dimension (feature) and the bias
- ▶ Features can be discrete or continuous
- ▶ Output of the model: value $y \in [0, 1]$
- ▶ Allows for gradient-based learning of the parameters

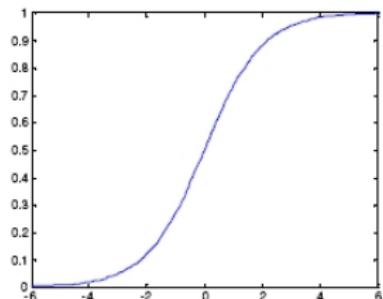
Shape of the Logistic Function

Let's look at how modifying w changes the shape of the function

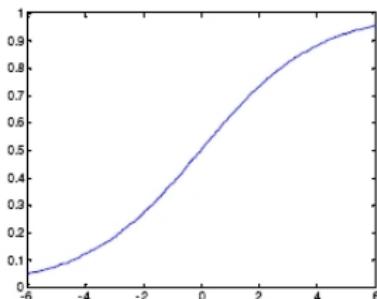
1D example:

$$y = \sigma(w_1 x + w_0)$$

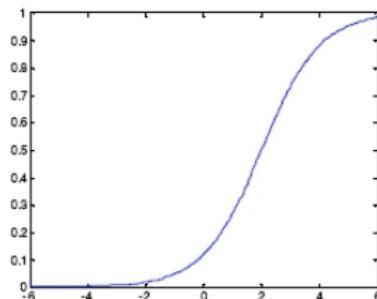
$$w_0 = 0, w_1 = 1$$



$$w_0 = 0, w_1 = 0.5$$



$$w_0 = -2, w_1 = 1$$



Probabilistic Interpretation

If we have a value between 0 and 1, let's use it to model class probability

$$p(C = 0|\mathbf{x}) = \sigma(\mathbf{w}^T \mathbf{x} + w_0) \quad \text{with} \quad \sigma(z) = \frac{1}{1 + \exp(-z)}$$

Substituting we have

$$p(C = 0|\mathbf{x}) = \frac{1}{1 + \exp(-\mathbf{w}^T \mathbf{x} - w_0)}$$

Suppose we have two classes, how can I compute $p(C = 1|\mathbf{x})$?

Use the marginalization property of probability

$$p(C = 1|\mathbf{x}) + p(C = 0|\mathbf{x}) = 1$$

Thus

$$p(C = 1|\mathbf{x}) = 1 - \frac{1}{1 + \exp(-\mathbf{w}^T \mathbf{x} - w_0)} = \frac{\exp(-\mathbf{w}^T \mathbf{x} - w_0)}{1 + \exp(-\mathbf{w}^T \mathbf{x} - w_0)}$$

Decision Boundary for Logistic Regression

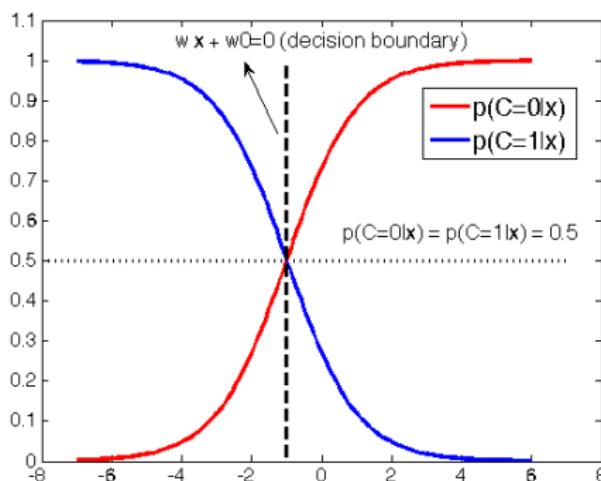
What is the **decision boundary** for logistic regression?

$$p(C = 1|\mathbf{x}, \mathbf{w}) = p(C = 0|\mathbf{x}, \mathbf{w}) = 0.5$$

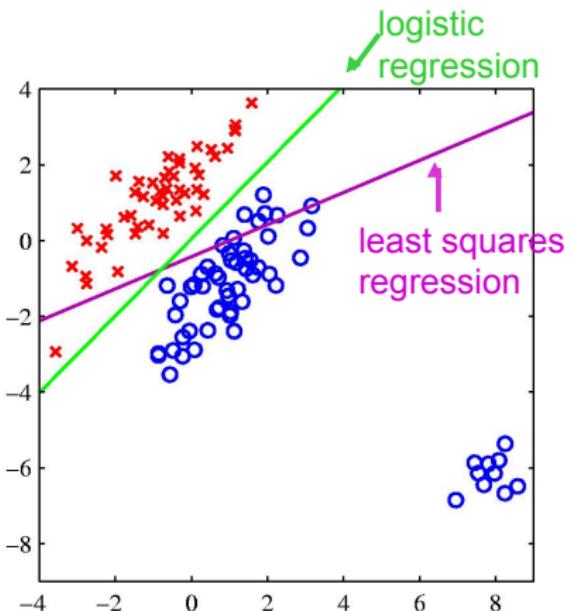
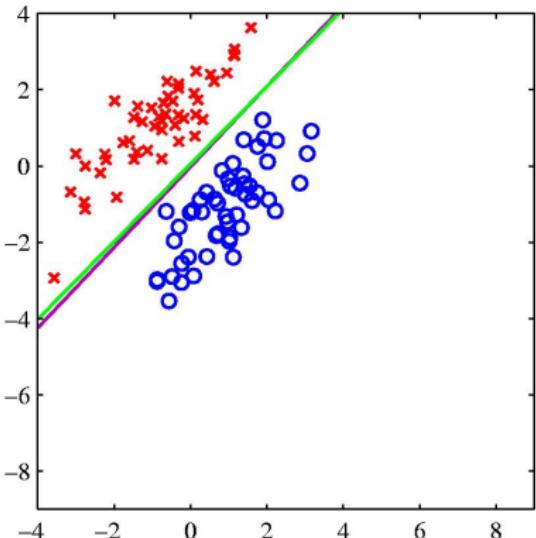
$$p(C = 0|\mathbf{x}, \mathbf{w}) = \sigma(\mathbf{w}^T \mathbf{x} + w_0) = 0.5, \text{ where } \sigma(z) = \frac{1}{1+\exp(-z)}$$

Decision boundary: $\mathbf{w}^T \mathbf{x} + w_0 = 0$

Logistic regression has a **linear decision boundary**



Logistic Regression vs Least Squares Regression



If the right answer is 1 and the model says 1.5, it loses, so it changes the boundary to avoid being “too correct” (tilts away from outliers)

Example

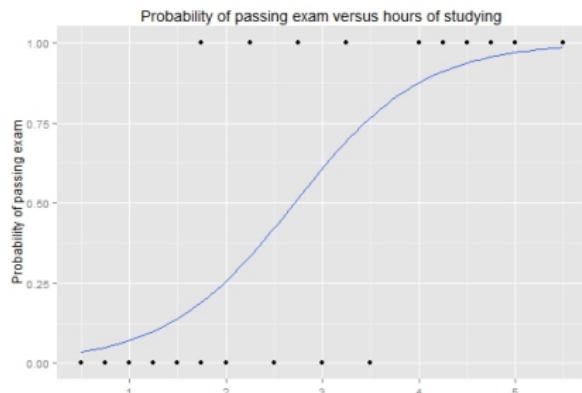
Problem: Given the number of hours a student spent learning, will (s)he pass the exam?

Training data (top row: $x^{(i)}$, bottom row: $t^{(i)}$)

Hours	0.50	0.75	1.00	1.25	1.50	1.75	1.75	2.00	2.25	2.50	2.75	3.00	3.25	3.50	4.00	4.25	4.50	4.75	5.00	5.50
Pass	0	0	0	0	0	0	1	0	1	0	1	0	1	0	1	1	1	1	1	1

Learn w for our model, i.e., logistic regression (coming up)

Make predictions:



Hours of study	Probability of passing exam
1	0.07
2	0.26
3	0.61
4	0.87
5	0.97

Learning?

When we have a d-dim input $\mathbf{x} \in \Re^d$

How should we learn the weights $\mathbf{w} = (w_0, w_1, \dots, w_d)$?

We have a probabilistic model

Let's use **maximum likelihood**

Probability vs Likelihood

- You can estimate a **probability** of an event using the function that describes the probability distribution and its parameters.
- For example, you can estimate the outcome of a fair coin flip by using the Bernoulli distribution and the probability of success 0.5. In this ideal case, you already know how the data is distributed.

- But the real world is messy. Often you don't know the exact parameter values, and you may not even know the probability distribution that describes your specific use case.
- Instead, you have to estimate the function and its parameters from the data.
- The **likelihood** describes the relative evidence that the data has a particular distribution and its associated parameters.

Probability vs Likelihood

- We can describe the **likelihood** as a function of an observed value of the data x , and the distributions' unknown parameter θ .
$$f(x, \theta)$$
- In short, when estimating the **probability**, you go from a distribution and its parameters to the event.

Probability: $p(\text{event} | \text{distribution})$

- When estimating the **likelihood**, you go from the data to the distribution and its parameters.

Likelihood: $L(\text{distribution} | \text{data})$

Probability vs Likelihood

- Recall that a coin flip is a Bernoulli trial, which can be described in the following function.

$$P(X = x) = p^x(1 - p)^{1-x}$$

- The probability p is a parameter of the function.
- To be consistent with the likelihood notation, we write down the formula for the likelihood function with θ instead of p .

$$L(x, \theta) = \theta^x(1 - \theta)^{1-x}$$

Probability vs Likelihood

- Let's say we throw the coin 3 times. It comes up heads the first 2 times. The last time it comes up tails. What is the likelihood that hypothesis A given the data?
- Now, we need a hypothesis about the parameter theta. We assume that the coin is fair. The probability of obtaining heads is 0.5.

$$P(X = 1) = 0.5^1(1 - 0.5)^{1-1} = 0.5$$

$$P(X = 1) = 0.5^1(1 - 0.5)^{1-1} = 0.5$$

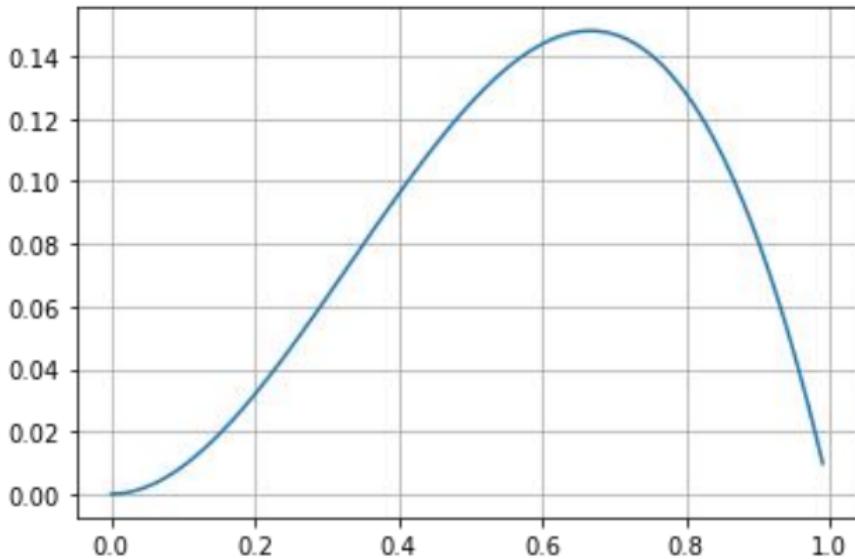
$$P(X = 1) = 0.5^0(1 - 0.5)^{1-0} = 0.5$$

- Multiplying all of these gives us the following value

$$\frac{1}{2} \times \frac{1}{2} \times \frac{1}{2} = \frac{1}{8}$$

Probability vs Likelihood

- For three coin tosses with 2 heads, the plot would look like this by varying the parameter from 0 to 1, for which the likelihood is maximized at $2/3$.



- In other words: Given the fact that 2 of our three coin tosses landed up heads, it seems more likely that the true probability of getting heads is $2/3$.

Probability vs Likelihood

- Mathematically, we can denote the maximum likelihood estimation as a function that results in the theta maximizing the likelihood.

$$\theta_{ML} = \arg \max_{\theta} L(\theta, x) = \prod_{i=1}^n p(x_i, \theta)$$

- The variable x represents the range of examples drawn from the unknown data distribution, which we would like to approximate and n the number of examples.
- For most practical applications, maximizing the log-likelihood is often a better choice because the logarithm reduced operations by one level. Multiplications become additions; powers become multiplications, etc.

$$\theta_{ML} = \arg \max_{\theta} l(\theta, x) = \sum_{i=1}^n \log(p(x_i, \theta))$$

Conditional Likelihood

Assume $t \in \{0, 1\}$, we can write the probability distribution of each of our training points $p(t^{(1)}, \dots, t^{(N)} | \mathbf{x}^{(1)}, \dots, \mathbf{x}^{(N)}; \mathbf{w})$

Assuming that the training examples are **sampled IID**: independent and identically distributed, we can write the *likelihood function*:

$$L(\mathbf{w}) = p(t^{(1)}, \dots, t^{(N)} | \mathbf{x}^{(1)}, \dots, \mathbf{x}^{(N)}; \mathbf{w}) = \prod_{i=1}^N p(t^{(i)} | \mathbf{x}^{(i)}; \mathbf{w})$$

We can write each probability as (will be useful later):

$$\begin{aligned} p(t^{(i)} | \mathbf{x}^{(i)}; \mathbf{w}) &= p(C = 1 | \mathbf{x}^{(i)}; \mathbf{w})^{t^{(i)}} p(C = 0 | \mathbf{x}^{(i)}; \mathbf{w})^{1-t^{(i)}} \\ &= \left(1 - p(C = 0 | \mathbf{x}^{(i)}; \mathbf{w})\right)^{t^{(i)}} p(C = 0 | \mathbf{x}^{(i)}; \mathbf{w})^{1-t^{(i)}} \end{aligned}$$

We can learn the model by **maximizing the likelihood**

$$\max_{\mathbf{w}} L(\mathbf{w}) = \max_{\mathbf{w}} \prod_{i=1}^N p(t^{(i)} | \mathbf{x}^{(i)}; \mathbf{w})$$

Easier to maximize the log likelihood $\log L(\mathbf{w})$

Loss Function

$$\begin{aligned}L(\mathbf{w}) &= \prod_{i=1}^N p(t^{(i)} | \mathbf{x}^{(i)}) \quad (\text{likelihood}) \\&= \prod_{i=1}^N \left(1 - p(C = 0 | \mathbf{x}^{(i)})\right)^{t^{(i)}} p(C = 0 | \mathbf{x}^{(i)})^{1-t^{(i)}}\end{aligned}$$

We can convert the maximization problem into minimization so that we can write the **loss function**:

$$\begin{aligned}\ell_{log}(\mathbf{w}) &= -\log L(\mathbf{w}) \\&= -\sum_{i=1}^N \log p(t^{(i)} | \mathbf{x}^{(i)}; \mathbf{w}) \\&= -\sum_{i=1}^N t^{(i)} \log(1 - p(C = 0 | \mathbf{x}^{(i)}, \mathbf{w})) - \sum_{i=1}^N (1 - t^{(i)}) \log p(C = 0 | \mathbf{x}^{(i)}; \mathbf{w})\end{aligned}$$

It's a convex function of \mathbf{w} . Can we get the global optimum?

Gradient Descent

$$\min_{\mathbf{w}} \ell(\mathbf{w}) = \min_{\mathbf{w}} \left\{ - \sum_{i=1}^N t^{(i)} \log(1 - p(C=0|\mathbf{x}^{(i)}, \mathbf{w})) - \sum_{i=1}^N (1 - t^{(i)}) \log p(C=0|\mathbf{x}^{(i)}, \mathbf{w}) \right\}$$

Gradient descent: iterate and at each iteration compute steepest direction towards optimum, move in that direction, step-size λ

$$w_j^{(t+1)} \leftarrow w_j^{(t)} - \lambda \frac{\partial \ell(\mathbf{w})}{\partial w_j}$$

You can write this in vector form

$$\nabla \ell(\mathbf{w}) = \left[\frac{\partial \ell(\mathbf{w})}{\partial w_0}, \dots, \frac{\partial \ell(\mathbf{w})}{\partial w_k} \right]^T, \quad \text{and} \quad \triangle(\mathbf{w}) = -\lambda \nabla \ell(\mathbf{w})$$

But where is \mathbf{w} ?

$$p(C=0|\mathbf{x}) = \frac{1}{1 + \exp(-\mathbf{w}^T \mathbf{x} - w_0)}, \quad p(C=1|\mathbf{x}) = \frac{\exp(-\mathbf{w}^T \mathbf{x} - w_0)}{1 + \exp(-\mathbf{w}^T \mathbf{x} - w_0)}$$

Let's Compute the Updates

The loss is

$$\ell_{log-loss}(\mathbf{w}) = - \sum_{i=1}^N t^{(i)} \log p(C = 1 | \mathbf{x}^{(i)}, \mathbf{w}) - \sum_{i=1}^N (1 - t^{(i)}) \log p(C = 0 | \mathbf{x}^{(i)}, \mathbf{w})$$

where the probabilities are

$$p(C = 0 | \mathbf{x}, \mathbf{w}) = \frac{1}{1 + \exp(-z)} \quad p(C = 1 | \mathbf{x}, \mathbf{w}) = \frac{\exp(-z)}{1 + \exp(-z)}$$

and $z = \mathbf{w}^T \mathbf{x} + w_0$

We can simplify

$$\begin{aligned}\ell(\mathbf{w})_{log-loss} &= \sum_i t^{(i)} \log(1 + \exp(-z^{(i)})) + \sum_i t^{(i)} z^{(i)} + \sum_i (1 - t^{(i)}) \log(1 + \exp(-z^{(i)})) \\ &= \sum_i \log(1 + \exp(-z^{(i)})) + \sum_i t^{(i)} z^{(i)}\end{aligned}$$

Now it's easy to take derivatives

Updates

$$\ell(\mathbf{w}) = \sum_i t^{(i)} z^{(i)} + \sum_i \log(1 + \exp(-z^{(i)}))$$

Now it's easy to take derivatives

Remember $z = \mathbf{w}^T \mathbf{x} + w_0$

$$\frac{\partial \ell}{\partial w_j} = \sum_i \left(t^{(i)} x_j^{(i)} - x_j^{(i)} \cdot \frac{\exp(-z^{(i)})}{1 + \exp(-z^{(i)})} \right)$$

What's $x_j^{(i)}$? The j -th dimension of the i -th training example $\mathbf{x}^{(i)}$

And simplifying

$$\frac{\partial \ell}{\partial w_j} = \sum_i x_j^{(i)} \left(t^{(i)} - p(C = 1 | \mathbf{x}^{(i)}; \mathbf{w}) \right)$$

Don't get confused with indices: j for the weight that we are updating and i for the training example

Gradient Descent

Putting it all together (plugging the update into gradient descent):
Gradient descent for logistic regression:

$$w_j^{(t+1)} \leftarrow w_j^{(t)} - \lambda \sum_i x_j^{(i)} \left(t^{(i)} - p(C = 1 | \mathbf{x}^{(i)}; \mathbf{w}) \right)$$

where:

$$p(C = 1 | \mathbf{x}^{(i)}; \mathbf{w}) = \frac{\exp(-\mathbf{w}^T \mathbf{x} - w_0)}{1 + \exp(-\mathbf{w}^T \mathbf{x} - w_0)} = \frac{1}{1 + \exp(\mathbf{w}^T \mathbf{x} + w_0)}$$

This is all there is to learning in logistic regression. Simple, huh?

Regularization

We can define priors on parameters \mathbf{w}

This is a form of regularization

Helps avoid large weights and **overfitting**

$$\max_{\mathbf{w}} \log \left[p(\mathbf{w}) \prod_i p(t^{(i)} | \mathbf{x}^{(i)}, \mathbf{w}) \right]$$

What's $p(\mathbf{w})$?

Regularized Logistic Regression

For example, define prior: normal distribution, zero mean and identity covariance $p(\mathbf{w}) = \mathcal{N}(0, \alpha^{-1} \mathbf{I})$

This prior pushes parameters towards zero

Including this prior the new gradient is

$$w_j^{(t+1)} \leftarrow w_j^{(t)} - \lambda \frac{\partial \ell(\mathbf{w})}{\partial w_j} - \lambda \alpha w_j^{(t)}$$

$$\begin{aligned} X &\sim N(\mu, \sigma^2) \\ f(x) &= \frac{1}{\sigma \sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \end{aligned}$$

where t here refers to iteration of the gradient descent

The parameter α is the importance of the regularization, and it's a **hyper-parameter**

How do we decide the best value of α (or a hyper-parameter in general)?

Use of Validation Set

Tuning hyper-parameters:

Never use test data for tuning the hyper-parameters

We can divide the set of training examples into two disjoint sets: [training](#) and [validation](#)

Use the first set (i.e., training) to estimate the weights w for different values of α

Use the second set (i.e., validation) to estimate the best α , by evaluating how well the classifier does on this second set

This tests how well it generalizes to unseen data

Cross-Validation

Leave-p-out cross-validation:

- ▶ We use p observations as the validation set and the remaining observations as the training set.
- ▶ This is repeated on all ways to cut the original training set.
- ▶ It requires \mathcal{C}_n^p for a set of n examples

Leave-1-out cross-validation: When $p = 1$, does not have this problem

k-fold cross-validation:

- ▶ The training set is randomly partitioned into k equal size subsamples.
- ▶ Of the k subsamples, a single subsample is retained as the validation data for testing the model, and the remaining $k - 1$ subsamples are used as training data.
- ▶ The cross-validation process is then repeated k times (the folds).
- ▶ The k results from the folds can then be averaged (or otherwise combined) to produce a single estimate

Cross-Validation (with Pictures)

Train your model:

Leave-one-out cross-validation:

k-fold cross-validation:

Leave-one-out

- Put one example in validation
- Train on remaining training examples, test on val example



Training examples



Validation



Leave-p-Out Cross-Validation

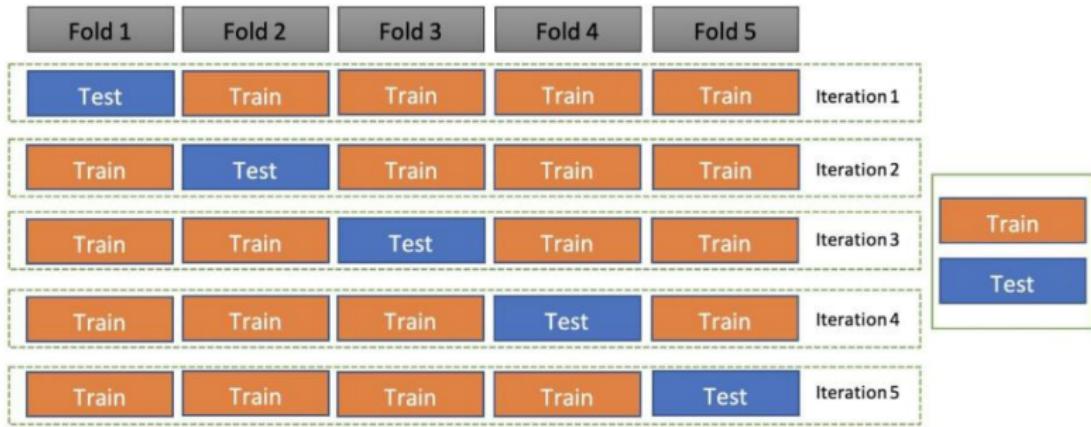
Code:

```
from sklearn.model_selection import LeavePOut,cross_val_score
from sklearn.datasets import load_iris
from sklearn.ensemble import RandomForestClassifier
iris=load_iris()
X=iris.data
Y=iris.target
lpo=LeavePOut(p=1)
lpo.get_n_splits(X)
tree=RandomForestClassifier(n_estimators=5,max_depth=3,n_jobs=-1)
score=cross_val_score(tree,X,Y,cv=lpo)
print("Cross Validation Scores-{}".format(score))
print("Average Cross Validation score :{}".format(score.mean()))
```

Leave-p-Out Cross-Validation

Output:

k-fold Cross-Validation



The image above shows 5 folds and hence, 5 iterations. In each iteration, one fold is the test set/validation set and the other $k-1$ sets (4 sets) are the train set. To get the final accuracy, you need to take the accuracy of the k -models validation data.

k-fold Cross-Validation

```
from sklearn.datasets import load_iris
from sklearn.model_selection import cross_val_score,KFold
from sklearn.linear_model import LogisticRegression
iris=load_iris()
X=iris.data
Y=iris.target
lr=LogisticRegression()
k_fold=KFold(n_splits=7)
score=cross_val_score(lr,X,Y,cv=k_fold)
print("Cross Validation Scores:{}".format(score))
print("Average Cross Validation score:{}".format(score.mean()))
```

Output:

```
Cross Validation Scores:[1.          1.          1.          0.80952381
0.95238095 0.85714286
0.9047619 ]
Average Cross Validation score:0.9319727891156463
```

Logistic Regression wrap-up

Advantages:

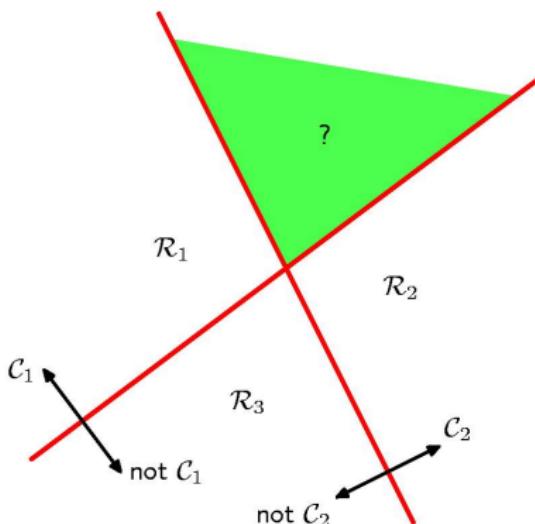
- Easily extended to multiple classes
- Natural probabilistic view of class predictions
- Quick to train
- Fast at classification
- Good accuracy for many simple data sets
- Resistant to overfitting
- Can interpret model coefficients as indicators of feature importance

Less good:

- Linear decision boundary (too simple for more complex problems?)

Discriminant Functions for $K > 2$ classes

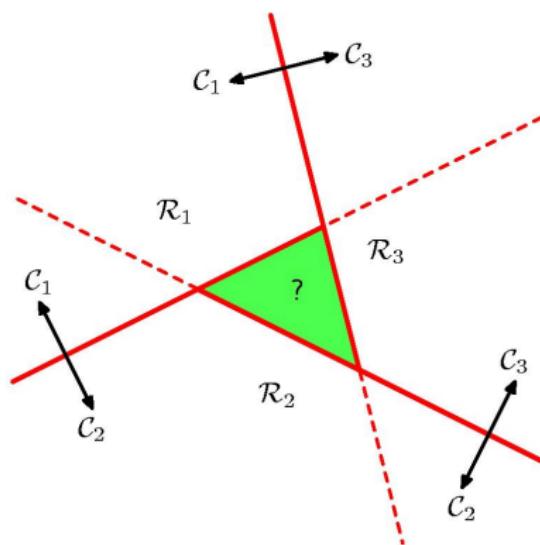
- First idea: Use K classifiers, each solving a two class problem of separating point in a class C_k from points not in the class.
- Known as **1 vs all** or **1 vs the rest** classifier



- PROBLEM: More than one good answer for green region!

Discriminant Functions for $K > 2$ classes

- Another simple idea: Introduce $K(K - 1)/2$ two-way classifiers, one for each possible pair of classes
- Each point is classified according to majority vote amongst the disc. func.
- Known as the **1 vs 1 classifier**



- PROBLEM: Two-way preferences need not be transitive

K-Class Discriminant

- We can avoid these problems by considering a single K-class discriminant comprising K functions of the form

$$y_k(\mathbf{x}) = \mathbf{w}_k^T \mathbf{x} + w_{k,0}$$

and then assigning a point \mathbf{x} to class C_k if

$$\forall j \neq k \quad y_k(\mathbf{x}) > y_j(\mathbf{x})$$

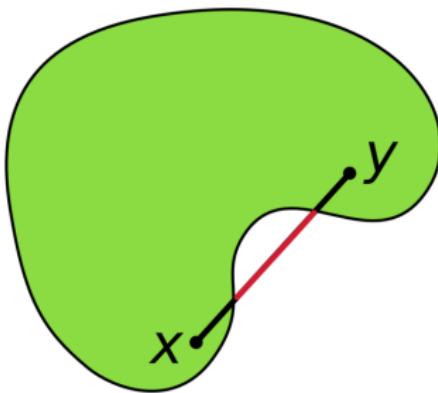
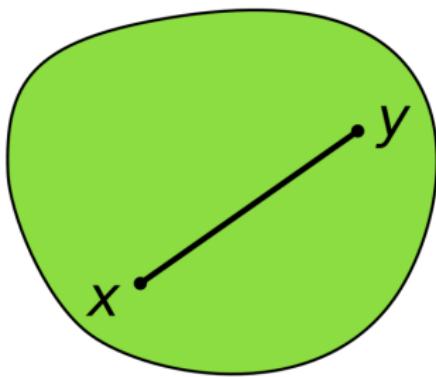
- Note that \mathbf{w}_k^T is now a vector, not the k -th coordinate
- The decision boundary between class C_j and class C_k is given by $y_j(\mathbf{x}) = y_k(\mathbf{x})$, and thus it's a $(D - 1)$ dimensional hyperplane defined as

$$(\mathbf{w}_k - \mathbf{w}_j)^T \mathbf{x} + (w_{k,0} - w_{j,0}) = 0$$

- What about the binary case? Is this different?
- What is the shape of the overall decision boundary?

K-Class Discriminant

- The decision regions of such a discriminant are always **singly connected** and **convex**
- In Euclidean space, an object is **convex** if for every pair of points within the object, every point on the straight line segment that joins the pair of points is also within the object

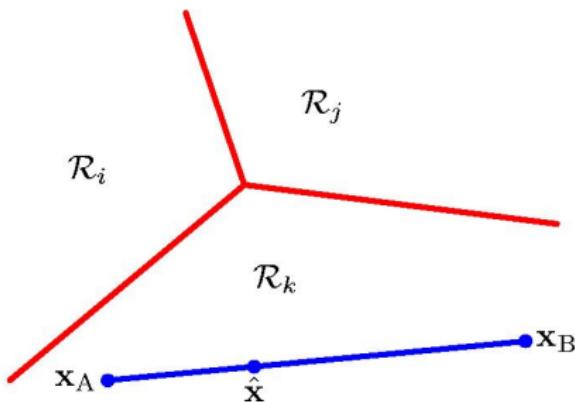


- Which object is convex?

K-Class Discriminant

- The decision regions of such a discriminant are always **singly connected** and **convex**
- Consider 2 points \mathbf{x}_A and \mathbf{x}_B that lie inside decision region R_k
- Any convex combination $\hat{\mathbf{x}}$ of those points also will be in R_k

$$\hat{\mathbf{x}} = \lambda \mathbf{x}_A + (1 - \lambda) \mathbf{x}_B$$



Proof

- A convex combination point, i.e., $\lambda \in [0, 1]$

$$\hat{\mathbf{x}} = \lambda \mathbf{x}_A + (1 - \lambda) \mathbf{x}_B$$

- From the linearity of the classifier $y(\mathbf{x})$

$$y_k(\hat{\mathbf{x}}) = \lambda y_k(\mathbf{x}_A) + (1 - \lambda) y_k(\mathbf{x}_B)$$

- Since \mathbf{x}_A and \mathbf{x}_B are in R_k , it follows that $y_k(\mathbf{x}_A) > y_j(\mathbf{x}_A)$, $y_k(\mathbf{x}_B) > y_j(\mathbf{x}_B)$, $\forall j \neq k$
- Since λ and $1 - \lambda$ are positive, then $\hat{\mathbf{x}}$ is inside R_k
- Thus R_k is singly connected and convex

Multi-class Classification with Linear Regression

- From before we have:

$$y_k(\mathbf{x}) = \mathbf{w}_k^T \mathbf{x} + w_{k,0}$$

which can be rewritten as:

$$\mathbf{y}(\mathbf{x}) = \tilde{\mathbf{W}}^T \tilde{\mathbf{x}}$$

where the k -th column of $\tilde{\mathbf{W}}$ is $[w_{k,0}, \mathbf{w}_k^T]$, and $\tilde{\mathbf{x}}$ is $[1, \mathbf{x}^T]^T$

- Training:** How can I find the weights $\tilde{\mathbf{W}}$ with the standard sum-of-squares regression loss?

1-of-K encoding:

For multi-class problems (with K classes), instead of using $t = k$ (target has label k) we often use a **1-of-K encoding**, i.e., a vector of K target values containing a single 1 for the correct class and zeros elsewhere

Example: For a 4-class problem, we would write a target with class label 2 as:

$$\mathbf{t} = [0, 1, 0, 0]^T$$

Multi-class Classification with Linear Regression

- Sum-of-least-squares loss:

$$\begin{aligned}\ell(\tilde{\mathbf{W}}) &= \sum_{n=1}^N \|\tilde{\mathbf{W}}^T \tilde{\mathbf{x}}^{(n)} - \mathbf{t}^{(n)}\|^2 \\ &= \|\tilde{\mathbf{X}} \tilde{\mathbf{W}} - \mathbf{T}\|_F^2\end{aligned}$$

where the n -th row of $\tilde{\mathbf{X}}$ is $[\tilde{\mathbf{x}}^{(n)}]^T$, and n -th row of \mathbf{T} is $[\mathbf{t}^{(n)}]^T$

- Setting derivative wrt $\tilde{\mathbf{W}}$ to 0, we get:

$$\tilde{\mathbf{W}} = (\tilde{\mathbf{X}}^T \tilde{\mathbf{X}})^{-1} \tilde{\mathbf{X}}^T \mathbf{T}$$

Multi-class Logistic Regression

- Associate a set of weights with each class, then use a normalized exponential output

$$p(C_k | \mathbf{x}) = y_k(\mathbf{x}) = \frac{\exp(z_k)}{\sum_j \exp(z_j)}$$

where the **activations** are given on

$$z_k = \mathbf{w}_k^T \mathbf{x}$$

- The function $\frac{\exp(z_k)}{\sum_j \exp(z_j)}$ is called a **softmax function**

Multi-class Logistic Regression

- The likelihood

$$p(\mathbf{T}|\mathbf{w}_1, \dots, \mathbf{w}_K) = \prod_{n=1}^N \prod_{k=1}^K p(C_k|\mathbf{x}^{(n)})^{t_k^{(n)}} = \prod_{n=1}^N \prod_{k=1}^K y_k^{(n)}(\mathbf{x}^{(n)})^{t_k^{(n)}}$$

with

$$p(C_k|\mathbf{x}) = y_k(\mathbf{x}) = \frac{\exp(z_k)}{\sum_j \exp(z_j)}$$

where k-th row of \mathbf{T} is 1-of-K encoding of example k and

$$z_k = \mathbf{w}_k^T \mathbf{x} + w_{k0}$$

- Derive the loss by computing the negative log-likelihood:

$$E(\mathbf{w}_1, \dots, \mathbf{w}_K) = -\log p(\mathbf{T}|\mathbf{w}_1, \dots, \mathbf{w}_K) = -\sum_{n=1}^N \sum_{k=1}^K t_k^{(n)} \log[y_k^{(n)}(\mathbf{x}^{(n)})]$$

This is known as the **cross-entropy** error for multiclass classification

- How do we obtain the weights?

Training Multi-class Logistic Regression

- How do we obtain the weights?

$$E(\mathbf{w}_1, \dots, \mathbf{w}_K) = -\log p(\mathbf{T} | \mathbf{w}_1, \dots, \mathbf{w}_K) = -\sum_{n=1}^N \sum_{k=1}^K t_k^{(n)} \log[y_k^{(n)}(\mathbf{x}^{(n)})]$$

- Do gradient descent, where the derivatives are

$$\frac{\partial y_j^{(n)}}{\partial z_k^{(n)}} = \underline{\delta(k, j)y_j^{(n)}} - y_j^{(n)}y_k^{(n)}$$

and

$$\frac{\partial E}{\partial z_k^{(n)}} = \sum_{j=1}^K \frac{\partial E}{\partial y_j^{(n)}} \cdot \frac{\partial y_j^{(n)}}{\partial z_k^{(n)}} = y_k^{(n)} - t_k^{(n)}$$

$$\frac{\partial E}{\partial w_{k,i}} = \sum_{n=1}^N \sum_{j=1}^K \frac{\partial E}{\partial y_j^{(n)}} \cdot \frac{\partial y_j^{(n)}}{\partial z_k^{(n)}} \cdot \frac{\partial z_k^{(n)}}{\partial w_{k,i}} = \sum_{n=1}^N (y_k^{(n)} - t_k^{(n)}) \cdot x_i^{(n)}$$

- The derivative is the error times the input

Softmax for 2 Classes

- Let's write the probability of one of the classes

$$p(C_1|\mathbf{x}) = y_1(\mathbf{x}) = \frac{\exp(z_1)}{\sum_j \exp(z_j)} = \frac{\exp(z_1)}{\exp(z_1) + \exp(z_2)}$$

- I can equivalently write this as

$$p(C_1|\mathbf{x}) = y_1(\mathbf{x}) = \frac{\exp(z_1)}{\exp(z_1) + \exp(z_2)} = \frac{1}{1 + \exp(-(z_1 - z_2))}$$

- So the logistic is just a special case that avoids using redundant parameters
- Rather than having two separate set of weights for the two classes, combine into one

$$z' = z_1 - z_2 = \mathbf{w}_1^T \mathbf{x} - \mathbf{w}_2^T \mathbf{x} = \mathbf{w}^T \mathbf{x}$$

- The over-parameterization of the softmax is because the probabilities must add to 1.