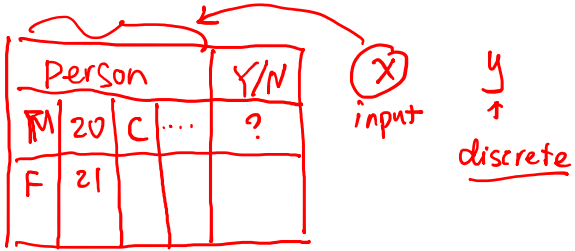
Linear Classification

Instructor: Hui Guan

Slides adapted from: https://github.com/jermwatt/machine_learning_refined

Outline

- Logistic regression and the cross entropy cost
- Logistic regression and the softmax cost
- Perceptron
- SVM
- Classification Quality Metrics



- Common examples of two class classification problems include
- face and general object detection, with classes consisting of with a face or object versus non-facial/object images
- textual sentiment analysis where classes consist of written product reviews ascribing a positive or negative opinion
- automatic diagnosis of medical conditions where classes consist of medical data corresponding to patients who either do or do not have a specific malady

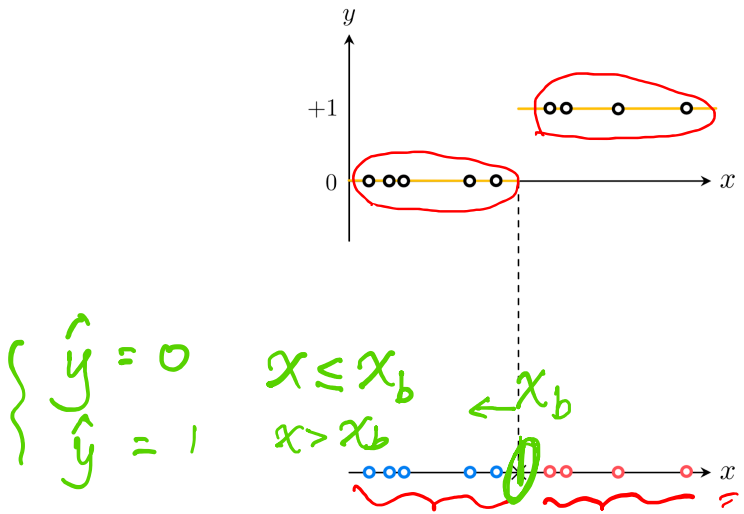
- The problem spurs the development of new cost functions that are better-suited to deal with such data.
- This includes *logistic regression*, the *perceptron*, and *support vector machines* perspectives on two-class classification.
- While these perspectives widely differ on the surface they all - as we will see - reduce to virtually the same essential principle for two-class classification.

6.2 Logistic Regression and the Cross Entropy Cost

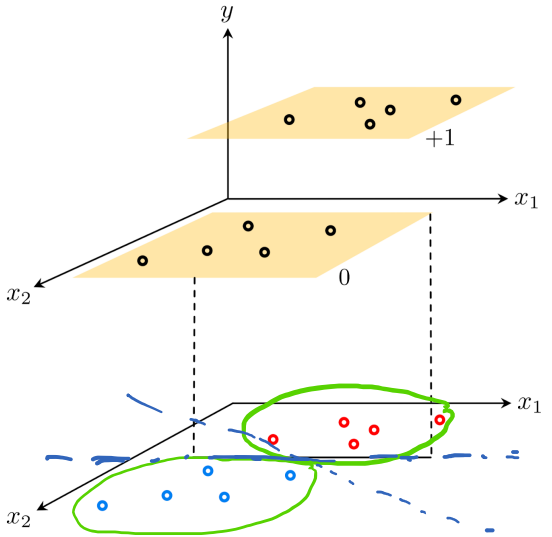
- Two class classification is a particular instance of *regression*.
- Here the output of a dataset of ***P*** points $\{(\mathbf{x}_p, y_p)\}_{p=1}^{P}$ is no longer continuous but takes on two fixed numbers.
- The actual value of these numbers is in principle arbitrary, but particular value pairs are more helpful than others for derivation purposes.
- Here we will use the values $y_p \in \{0, +1\}$ - that is every output takes on either the value ***0*** or ***+1***.

- Often in the context of classification the output values y_p are called *labels*, and all points sharing the same label value are referred to as a *class* of data.
- Hence a dataset containing points with label values $y_p \in \{0, +1\}$ is said to be a dataset consisting of two classes.

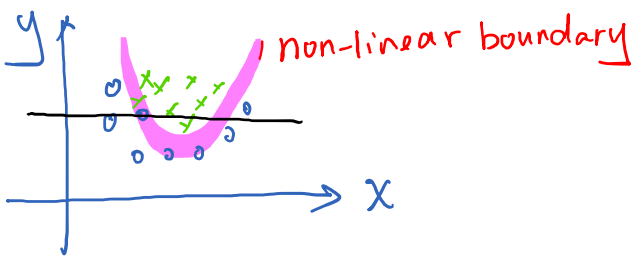
- Here the 'bottom' step is the region of the space containing most of the points that have label value $y_p = 0$.
- The 'top step' likewise contains most of the points having label value $y_p = +1$.
- These steps are largely separated by a point when $N = 1$, a line when $N = 2$, and a hyperplane when N is larger.



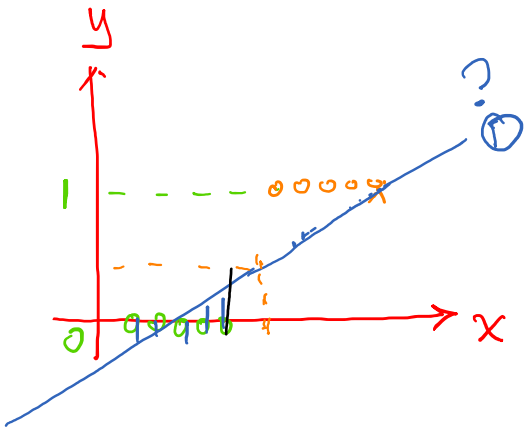
1d input: decision boundary is a point



2d input: decision boundary is a line



- This is the simplest sort of dataset with binary output we could aim to perform regression on - one with a *linear* boundary.
- More generally the boundary could certainly be nonlinear.
- We will deal with this more general potentiality later on - when discussing neural networks, trees, and kernel-based methods.
- But first we deal with the current scenario: **How can we perform regression on a dataset like the ones described in the figure above**?



$$\min \| \hat{y} - y \|_2^2$$

$$\hat{y} = x^T w + b$$

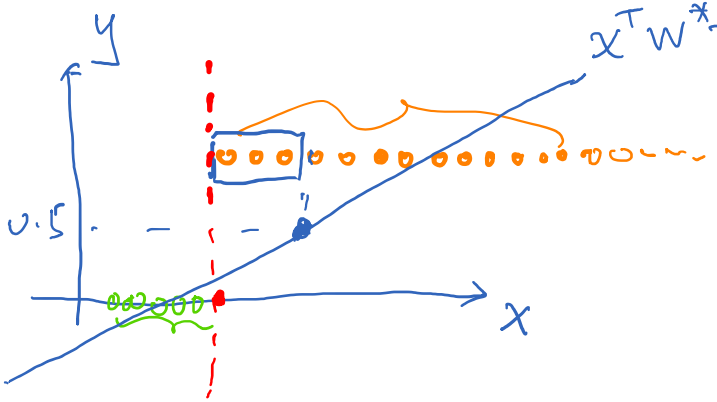
y: ground truth

① $y = x^T w^* + b^*$

$$\begin{cases} \hat{y} < 0.5 \rightarrow \hat{y} = 0 \\ \hat{y} \geq 0.5 \rightarrow \hat{y} = 1 \end{cases}$$

"Step function"

Trying to fit a discontinuous step function



Q: linear regression:

$$w^*, b^* = \arg\min \| \hat{y} - y \|_2^2$$

$$\text{step}(\hat{y}) = \begin{cases} 0 & \text{if } \hat{y} < 0.5 \\ 1 & \text{otherwise} \end{cases}$$

- Intuitively it is obvious that simply fitting a line of the form $y = w_0 + w_1 x$ to such a dataset will result in an extremely subpar results.
- The line by itself is simply too inflexible to account for the nonlinearity present in such data.
- A dataset that is roughly distributed on two steps needs to be fit with a function that matches this general shape.
- In other words such data needs to be fit with a *step function*.

- So ideally we would like to fit a ***discontinuous step function*** *with a* ***linear boundary*** to such a dataset.
- When $N = 1$ a *linear model* defining this boundary is just a line $w_0 + x\ w_1$ composed with the *step function* $\text{step}\,(\cdot)$ as

$$\text{step}\,(w_0 + x\ w_1)$$

- where the step function is defined as

$$\text{step}(x) = \begin{cases} 1 & \text{if } x \geq 0.5 \\ 0 & \text{if } x < 0.5 \end{cases}.$$

- Note here that what happens with $\text{step}(0.5)$ is - for our purposes - arbitrary (i.e., it can be set to any fixed value or left undefined as we have done).
- The linear boundary between the two steps is defined by all points ***x*** where

$$w_0 + xw_1 = 0.5$$

More generally with general N dimensional input we can write the linear model defining the boundary as $\dot{\mathbf{x}}^T\mathbf{w} = w_0 + x_1 w_1 + x_2 w_2 + \cdots + x_N w_N$

$$\mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix} \quad \text{and} \quad \dot{\mathbf{x}} = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix}.$$

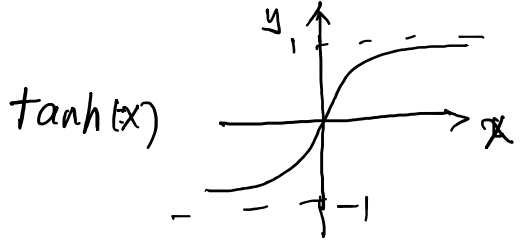
- A corresponding step function is then simply the $\text{step}\left(\cdot\right)$ of this linear combination

$$\text{step}\left(\dot{\mathbf{x}}^T\mathbf{w}\right)$$

- and the linear boundary between the steps is defined by all points $\dot{\mathbf{x}}$ where

$$\dot{\mathbf{x}}^T\mathbf{w} = 0.5$$

$$\tanh(x) = \frac{2}{1+e^{-x}} - 1$$



- **How do we tune the parameters of the line?**

$$\arg\min \| \dot{x}^T w - y \|_2^2 \quad \text{step}\left(\dot{\mathbf{x}}^T \mathbf{w}\right)$$

Goal: f : differentiable

→ ① fit $\dot{x}^T w$ → step($\dot{x}^T w^*$) → poor accuracy

$$\min \| f(\dot{x}^T w) - y \|_2^2$$

⇒ ② replace it ??

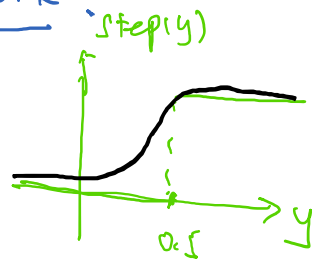
$$\arg\min_w \| \text{step}(\dot{x}^T w) - y \|_2^2 \quad g(w)$$

? can this work ?

$\hat{y}$

$$\frac{\partial g(w)}{\partial w} = \frac{\partial g(w)}{\partial (\dot{x}^T w)} \cdots = 0$$

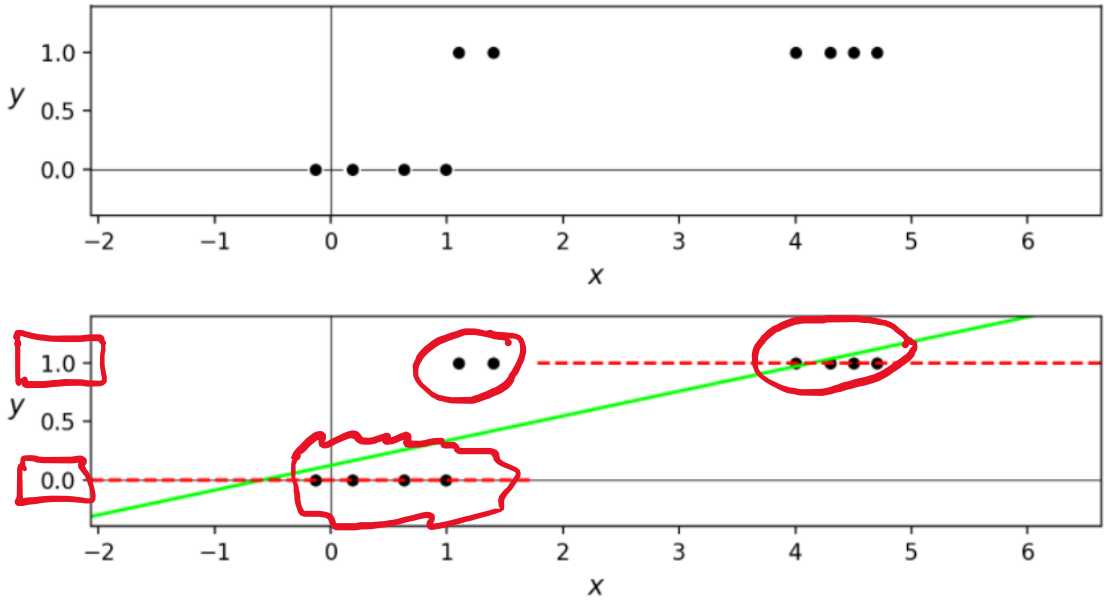
$$\text{step}(y) = \begin{cases} 0 \\ 1 \end{cases}$$



- **How do we tune the parameters of the line?**
- We could try to take the lazy way out and *first* fit the line to the classification dataset via linear regression, then compose the line with the step function to get a step function fit.
- However this does not work well in general - as we will see even in the simple instance below.
- Instead we need to tune the parameters w_0 and w_1 *after* composing the linear model with the step function, or in other words we need to tune the parameters of $\text{step}\left(\dot{\mathbf{x}}^T\mathbf{w}\right)$

Example: Fitting a line first and taking the step afterward fails to represent a two-class dataset well

- Below we load in a simple two-class dataset (top panel), fit a line to this dataset via linear regression; We then compose the fitted line with the step function to produce a step function fit.
- Both the linear fit (in green) as well as its composition with the step function (in dashed red) are shown along with the data in the bottom panel.



- Of course the line itself provides a terrible representation of the nonlinear dataset.
- But its evaluation through the step is also quite poor for such a simple dataset, failing to properly identify two points on the top step.
- In the parlance of classification these types of points are referred to as *misclassified points*.

- **How do we tune these parameters properly?**
- As with linear regression, here we can try to setup a proper Least Squares function that - when minimized - recovers our ideal weights.
- We can do this by simply reflecting on the sort of ideal relationship we want to find between the input and output of our dataset.

- Take a single point $(\mathbf{x}_p, y_p)$.
- Notice in the example above that ideally for a good fit we would like our weights to be such if this point has a label $+1$ it lies in the positive region of the space.
- That is where $\dot{\mathbf{x}}^T \mathbf{w} > 0.5$ so that $\text{step}\left(\dot{\mathbf{x}}^T \mathbf{w}\right) = +1$ matches its label value.

- Likewise if this point has label *0* we would like it to lie in the negative region where $\dot{\mathbf{x}}^T\mathbf{w} < 0.5$ so that $\text{step}\left(\dot{\mathbf{x}}^T\mathbf{w}\right) = 0$ matches its label value.
- So in short what we would ideally like for this point is that its evaluation matches its label value, i.e., that

$$\text{step}\left(\dot{\mathbf{x}}_p^T\mathbf{w}\right) = y_p.$$

- And of course we would like this to hold for every point.

- To find weights that satisfy this set of *P* equalities as best as possible we could square the difference between both sides of each and average them.
- This gives the Least Squares cost function

$$g(\mathbf{w}) = \frac{1}{P}\sum_{p=1}^{P}\left(\text{step}\left(\dot{\mathbf{x}}^{T}\mathbf{w}\right) - y_p\right)^2$$

- Unfortunately because this Least Squares cost takes on only integer values.
- In other words, **it is impossible to minimize with local optimization** (e.g., gradient-based techniques), as at every point the function is completely flat.
- Because of this local optimization cannot easily take even a single step 'downhill' regardless of where they are initialized.
- This problem is inherited from our use of the step function, itself a discontinuous step.

The logistic sigmoid function

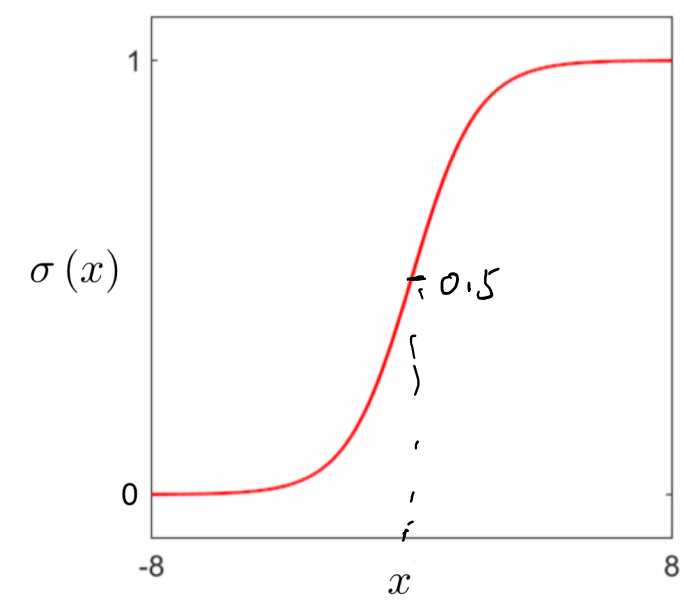
- As mentioned above, we cannot directly minimize the Least Squares above due to our use of the step function.
- In other words, we cannot directly fit a *discontinuous* step function to our data.
- In order to go further we need to replace the step, ideally with a ***continuous* function** that matches it very closely everywhere.

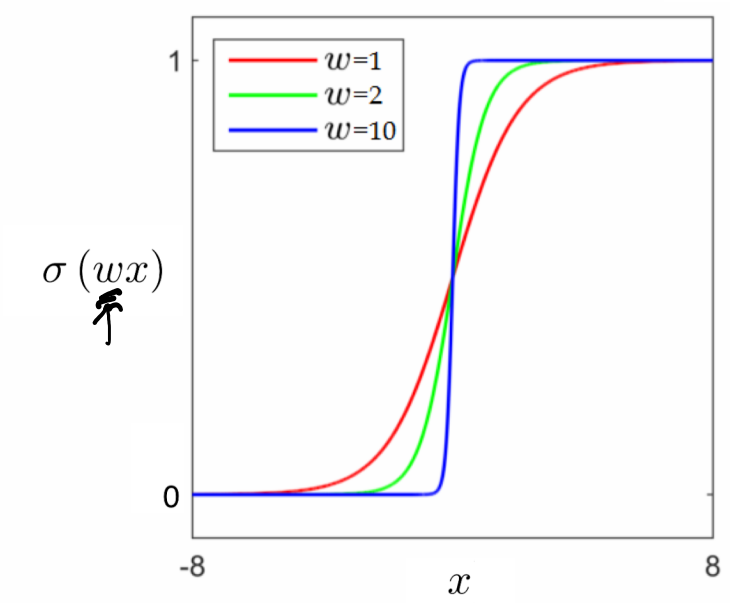
- Thankfully such a function is readily available: the sigmoid function, $\sigma(\cdot)$

$$\sigma(x) = \frac{1}{1+e^{-x}}.$$

- This kind of function is alternatively called a *logistic* or *logistic sigmoid* function.
- When we fit such a function to a classification dataset we are therefore performing regression with a logistic or *logistic regression*.

- In the figure below we plot the sigmoid function (left panel), as well as several internally weighted versions of it (right panel).
- For the correct setting of internal weights the hyperbolic tangent function can be made to look arbitrarily similar to the step function.

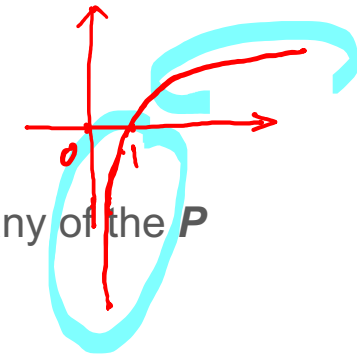




Logistic regression using the Least Squares cost

$\min \| \text{step}(\dot{x}^T w) - y \|_2^2$

$\min_p \| \sigma(\dot{x}^T w) - y \|_2^2$ cost function

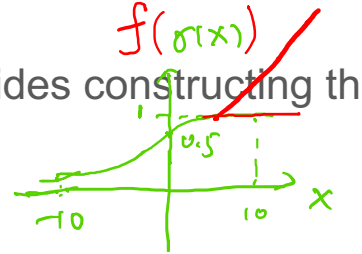


- Swapping out the step function with the sigmoid we aim to satisfy as many of the ***P*** equations that follow hold

$$\sigma\left(\dot{\mathbf{x}}^T\mathbf{w}\right) = y_p \qquad p = 1, .., P$$

- To *learn* parameters that force these approximations to hold, we can do precisely what we did in the case of linear regression
- We can try to minimize the e.g., the squared error between both sides constructing the Least Squares point-wise cost

$\sigma(x) = \frac{1}{1+e^{-x}}$



$\sigma(\dot{x}_p^T w) \approx 1$

$$g_p(\mathbf{w}) = \left(\sigma\left(\dot{\mathbf{x}}_p^T\mathbf{w}\right) - y_p\right)^2 \qquad p = 1, .., P.$$

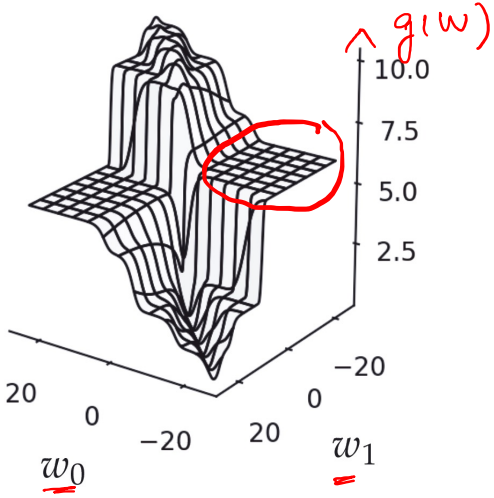
big (>10, <-10)

$\dot{x}_p^T w > 10$?

- Taking the average of these squared errors gives a Least Squares cost for *logistic regression*

$$g\left(\mathbf{w}\right) = \frac{1}{P}\sum_{p=1}^{P}\left(\sigma\left(\dot{\mathbf{x}}^{T}\mathbf{w}\right) - y_p\right)^2$$

- This function is generally **non-convex and contains large flat regions**.
- This can make it difficult, but not impossible, to minimize properly using some local methods (e.g., standard gradient descent).



Logistic regression using the Cross Entropy cost

- There is more than one way to form a cost function whose minimum forces as many of the ***P*** equalities to hold as well as possible.
- The squared error / point-wise cost $g_p\left(\mathbf{w}\right) = \left(\sigma\left(\dot{\mathbf{x}}_p^T\mathbf{w}\right) - y_p\right)^2$ penalty works universally, regardless of the values taken by the output by y_p .
- However because **we *know* that the output we deal with now is limited to the *discrete* values** $y_p \in \{0, 1\}$ it is reasonable to ask if we cannot create a more appropriate cost that is customized to deal with just such output.

- Such a point-wise cost does exist for these restricted output values.
- One such cost - which we call the *Log Error* - is as follows

$$g_p(\mathbf{w}) = \begin{cases} -\log\left(\sigma\left(\dot{\mathbf{x}}_p^T\mathbf{w}\right)\right) & \text{if } y_p = 1 \\ -\log\left(1-\sigma\left(\dot{\mathbf{x}}_p^T\mathbf{w}\right)\right) & \text{if } y_p = 0. \end{cases}$$

$-\log(0) = +\infty$ $\leftarrow -10$ $\log(1) = 0$ > 10

if $\hat{y}_p = 1$ $g_p(w) = 0$

else $\hat{y}_p = 0$ $g_p(w) = +\infty$

$$g_p(w) = -\log\left(\sigma\left(\dot{x}_p^T w\right)\right) \cdot y_p + \left(1-y_p\right) \cdot \left(-\log\left(1-\sigma\left(\dot{x}_p^T w\right)\right)\right)$$

Regardless of the weight values, this cost is *always nonnegative* and takes on a minimum value at ***0***.

- We can then form the so-called *Cross Entropy* cost function by taking the average of the Log Error costs over all P points as

$$g\left(\mathbf{w}\right) = \frac{1}{P}\sum_{p=1}^{P} g_p\left(\mathbf{w}\right).$$

- Finally notice that we can write the Log Error equivalently - combining the two cases - in a single line as

$$g_p\left(\mathbf{w}\right) = -y_p \log\left(\sigma\left(\dot{\mathbf{x}}_p^T\mathbf{w}\right)\right) - \left(1 - y_p\right)\log\left(1 - \sigma\left(\dot{\mathbf{x}}_p^T\mathbf{w}\right)\right).$$

- We can form the same cost function as above by taking the average of this form of the Log Error giving

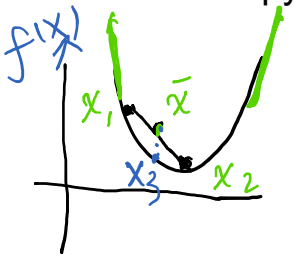
$$g(\mathbf{w}) = \frac{1}{P}\sum_{p=1}^{P} g_p(\mathbf{w})$$
$$= -\frac{1}{P}\sum_{p=1}^{P} y_p \log\left(\sigma\left(\dot{\mathbf{x}}_p^T\mathbf{w}\right)\right) + (1-y_p)\log\left(1-\sigma\left(\dot{\mathbf{x}}_p^T\mathbf{w}\right)\right)$$

- This is the more common way of expressing the Cross Entropy cost.

- To optimally tune the parameters $\mathbf{w}$ we want to *minimize* the Cross Entropy cost as

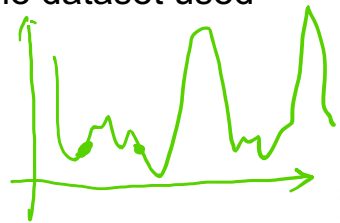
$$\underset{\mathbf{w}}{\text{minimize}} \; -\frac{1}{P}\sum_{p=1}^{P} y_p \log\left(\sigma\left(\dot{\mathbf{x}}_p^T\mathbf{w}\right)\right) + \left(1-y_p\right)\log\left(1-\sigma\left(\dot{\mathbf{x}}_p^T\mathbf{w}\right)\right)$$

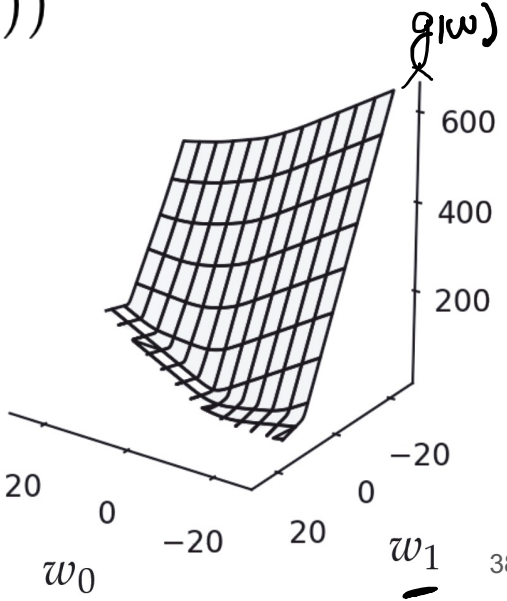
The Cross Entropy cost is *always convex* regardless of the dataset used



$\bar{x} = \alpha \cdot x_1 + (1-\alpha)\, x_2$

$f(\bar{x}) \geq f(x_3)$





- Since the Cross Entropy cost function is convex, a variety of local optimization schemes can be more easily used to properly minimize it.
- For this reason, the Cross Entropy cost is used more often in practice for logistic regression than is the logistic Least Squares cost.

Implementing and minimizing a modular Cross Entropy cost in `Python`

$$\text{model}\left(\mathbf{x}_p, \mathbf{w}\right) = \dot{\mathbf{x}}_p^T \mathbf{w} \ .$$

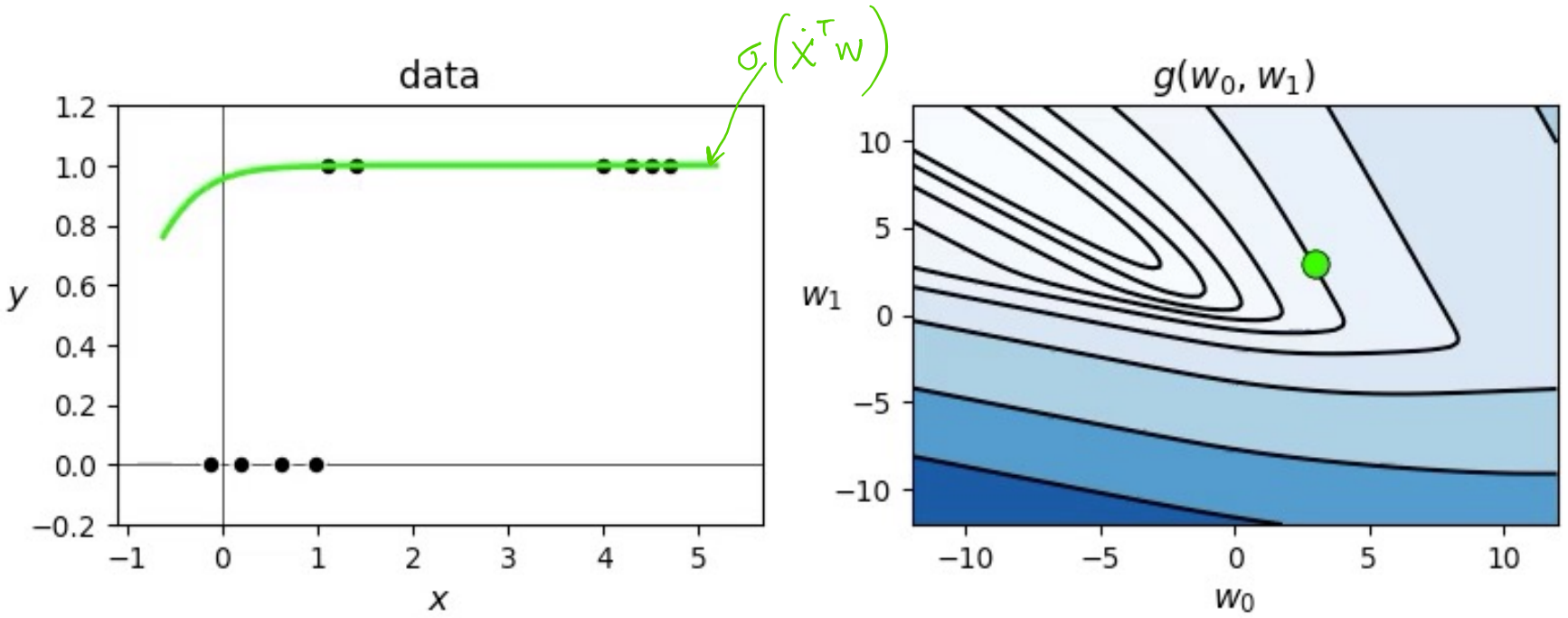
```
# compute linear combination of input point
def model(x,w):
    a = w[0] + np.dot(x.T,w[1:])
    return a.T
```

$$g(\mathbf{w}) = -\frac{1}{P}\sum_{p=1}^{P} y_p \log\left(\sigma\left(\text{model}\left(\mathbf{x}_p, \mathbf{w}\right)\right)\right) + \left(1 - y_p\right)\log\left(1 - \sigma\left(\text{model}\left(\mathbf{x}_p, \mathbf{w}\right)\right)\right)$$

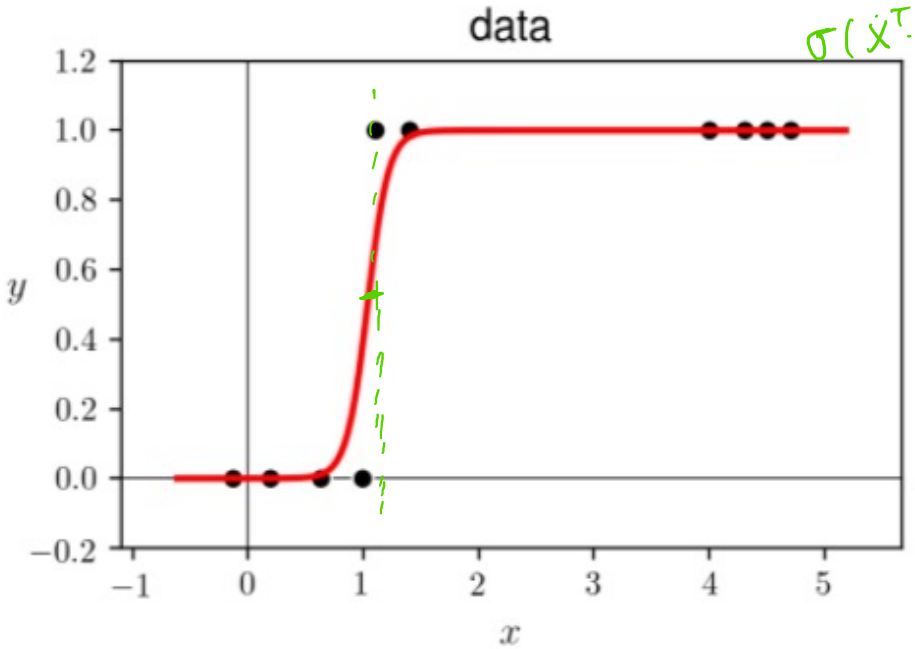
```
# define sigmoid function
def sigmoid(t):
    return 1/(1 + np.exp(-t))

# the convex cross-entropy cost function
def cross_entropy(w):
    # compute sigmoid of model
    a = sigmoid(model(x,w))

    # compute cost of label 0 points
    ind = np.argwhere(y == 0)[:,1]
    cost = -np.sum(np.log(1 - a[:,ind]))

    # add cost on label 1 points
    ind = np.argwhere(y==1)[:,1]
    cost -= np.sum(np.log(a[:,ind]))

    # compute cross-entropy
    return cost/y.size
```

Example: Using gradient descent to perform logistic regression using the Cross Entropy cost

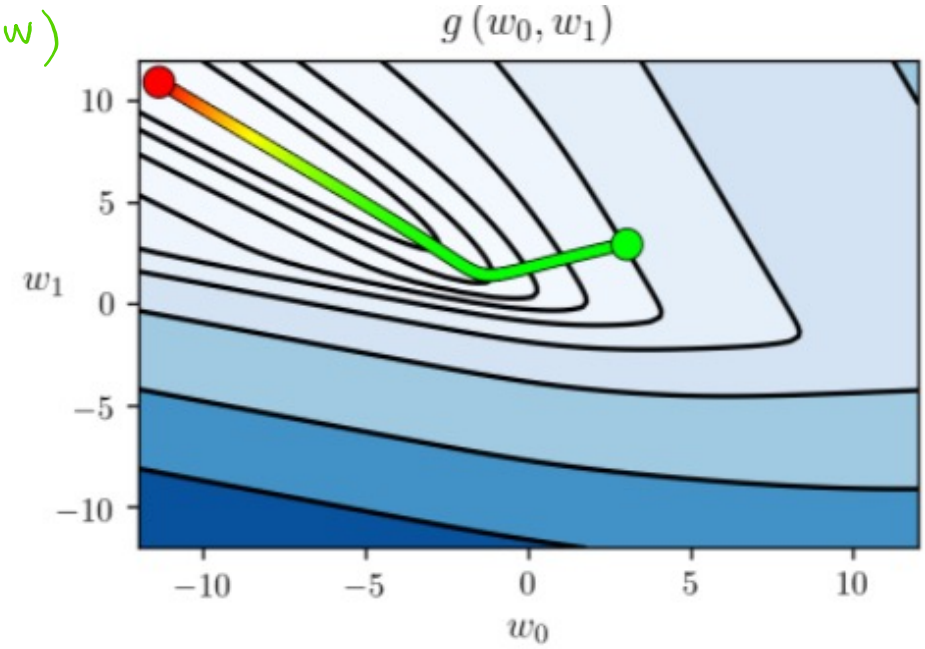
- In this example we minimize the Cross Entropy cost using gradient descent.
- We initialize at the point $w_0 = 3$ and $w_1 = 3$, set $\alpha = 1$, and run for 25 steps. (Note this slide has Animation)



Below we show the result of running gradient descent with the same initial point and fixed steplength parameter for ***2000*** iterations, which results in a better fit.



$\sigma(x^T \cdot w)$



Quiz:

https://kawsar34.medium.com/machine-learning-quiz-04-logistic-regression-7fd714bb7fff

Question 1: Logistic regression is used for ___?

(A) classification

(B) regression

(C) clustering

(D) All of these

Question 1: Logistic regression is used for ___?

(A) classification

(B) regression

(C) clustering

(D) All of these

Question 2: Logistic Regression is a Machine Learning algorithm that is used to predict the probability of a ___?

(A) categorical independent variable

(B) categorical dependent variable.

(C) numerical dependent variable.

(D) numerical independent variable.

Question 2: Logistic Regression is a Machine Learning algorithm that is used to predict the probability of a ___?

(A) categorical independent variable

(B) categorical dependent variable.

(C) numerical dependent variable.

(D) numerical independent variable.

Question 4: In a logistic regression model, the decision boundary can be ___.

(A) linear

(B) non-linear

(C) both (A) and (B)

(D) none of these

Question 4: In a logistic regression model, the decision boundary can be ___.

(A) linear

(B) non-linear

(C) both (A) and (B)

(D) none of these

Question 5: What's the cost function of the logistic regression?

(A)Sigmoid function
(B)Logistic Function
(C)Both (a) and (b)
(D)None of these

Question 5: What's the cost function of the logistic regression?

(A) Sigmoid function

(B) Logistic Function

(C) both (A) and (B)

(D) none of these

Question 6: Why cost function which has been used for linear regression can't be used for logistic regression?

(A) Linear regression uses mean squared error as its cost function. If this is used for logistic regression, then it will be a non-convex function of its parameters. Gradient descent will converge into global minimum only if the function is convex.

(B) Linear regression uses mean squared error as its cost function. If this is used for logistic regression, then it will be a convex function of its parameters. Gradient descent will converge into global minimum only if the function is convex.

(C) Linear regression uses mean squared error as its cost function. If this is used for logistic regression, then it will be a non-convex function of its parameters. Gradient descent will converge into global minimum only if the function is non-convex.

(D) Linear regression uses mean squared error as its cost function. If this is used for logistic regression, then it will be a convex function of its parameters. Gradient descent will converge into global minimum only if the function is non-convex.

Question 6: Why cost function which has been used for linear regression can't be used for logistic regression?

(A) Linear regression uses mean squared error as its cost function. If this is used for logistic regression, then it will be a non-convex function of its parameters. Gradient descent will converge into global minimum only if the function is convex.

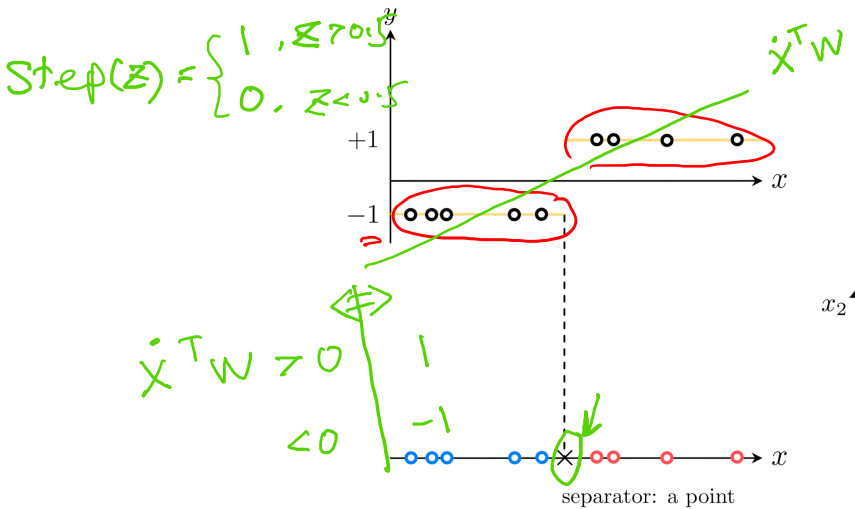
(B) Linear regression uses mean squared error as its cost function. If this is used for logistic regression, then it will be a convex function of its parameters. Gradient descent will converge into global minimum only if the function is convex.

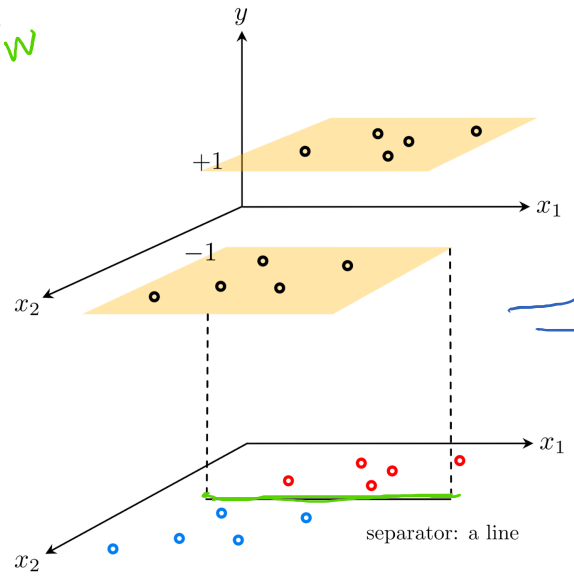
(C) Linear regression uses mean squared error as its cost function. If this is used for logistic regression, then it will be a non-convex function of its parameters. Gradient descent will converge into global minimum only if the function is non-convex.

(D) Linear regression uses mean squared error as its cost function. If this is used for logistic regression, then it will be a convex function of its parameters. Gradient descent will converge into global minimum only if the function is non-convex.

6.3 Logistic Regression and the Softmax Cost

- If we change the label values from $y_p \in \{0, 1\}$ to $y_p \in \{-1, +1\}$ much of the story we saw previously unfolds here as well, with only slight differences.
- That is, instead of our data ideally sitting on a step function (with linear boundary) with lower / upper steps taking on the values ***0*** and ***1*** respectively, they take on values ***-1*** and ***+1*** as shown below for prototypical cases where $N = 1$ (left panel) and $N = 2$ (right panel).





- This step function taking on values $\{-1, +1\}$, having a *linear boundary between its bottom and top steps*

$$\text{sign}\left(\dot{\mathbf{x}}^T \mathbf{w}\right)$$

- Here we use our compact vector notation used in the previous Section and where the $\text{sign}\left(\cdot\right)$ function is defined as

$$\text{sign}(x) = \begin{cases} +1 & \text{if } x \geq 0 \\ -1 & \text{if } x < 0 \end{cases}.$$

- Notice in this case that our linear decision boundary lies along those $\dot{\mathbf{x}}^T$ where $\dot{\mathbf{x}}^T\mathbf{w} = 0$.
- As with data using labels ***0*** and ***1***, linear regression would in general provide poor performance for data of this sort.
- Because we want to tune the weights of our model $\mathbf{w}$ so that our parameterized step function maps inputs to outputs correctly as

$$\text{sign}\left(\dot{\mathbf{x}}_p^T\mathbf{w}\right) \approx y_p$$

- We could employ a Least Squares cost function involving $\text{sign}\left(\cdot\right)$ directly as

$$g(\mathbf{w}) = \frac{1}{P}\sum_{p=1}^{P}\left(\text{sign}\left(\left(\dot{\mathbf{x}}_p^T\mathbf{w}\right)\right) - y_p\right)^2.$$

- However like the analagous Least Squares cost involving a step function, this is discontinuous, completely flat everywhere, and cannot be optimized easily.

The Tanh logistic sigmoid function

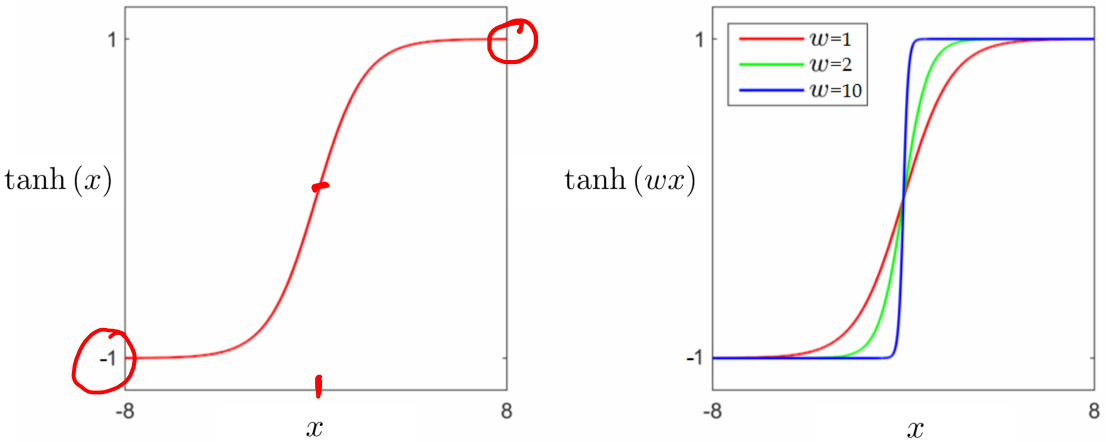
We could then look to replace the discrete $\text{sign}(\cdot)$ with a ***smooth approximation***.

A slightly re-scaled version of the *sigmoid* function - so that its values range between ***-1*** and ***1*** instead of ***0*** and ***1*** - serves this purpose well.

This scaled version of the sigmoid is often called the *hyperbolic tangent function* and is written as

$$\tanh(x) = 2\,\sigma(x) - 1 = \frac{2}{1+e^{-x}} - 1.$$

- Given that the sigmoid function $\sigma(\cdot)$ ranges smoothly between *0* and *1*, it is easy to see why $\tanh(\cdot)$ as defined above ranges smoothly between *-1* and *+1*.
- In the figure below we plot the tanh function (left panel), as well as several internally weighted versions of it (right panel).
- As we can see in the figure, for the correct setting of internal weights the hyperbolic tangent function can be made to look arbitrarily similar to the sign function.



Logistic regression using the Least Squares cost

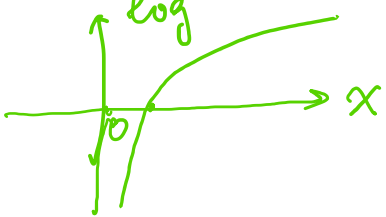
- Replacing $\text{sign}(\cdot)$ with $\tanh(\cdot)$ gives a similar desired relationship (assuming ideal weights are known)

$$\tanh\left(\dot{\mathbf{x}}_p^T \mathbf{w}\right) \approx y_p$$

- The analagous Least Squares cost function for recovering these weights

$$g(\mathbf{w}) = \frac{1}{P}\sum_{p=1}^{P}\left(\tanh\left(\left(\dot{\mathbf{x}}_p^T \mathbf{w}\right)\right) - y_p\right)^2$$

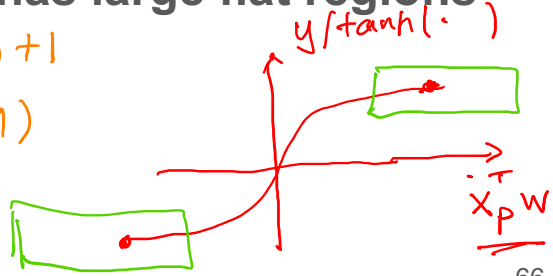
- However this Least Squares cost is **non-convex and has large flat regions** that can impair optimization progress.



$\min \log\left(\tanh\left(\dot{x}_p^T w\right)\right)$

$\tanh\left(\dot{x}_p^T w\right) \in (0, +\infty)$

$\tanh\left(\dot{x}_p^T w\right) + 1 \in (-1, 1)$



Logistic regression using the Softmax cost

- Because of this relationship we can employ *point-wise cost function* called the ***Log Error*** only our current label values

$$\tanh(x) = 2\sigma(x) - 1$$

$$g_p(\mathbf{w}) \overset{?}{=} \begin{cases} -\log\left(\sigma\left(\dot{\mathbf{x}}_p^T\mathbf{w}\right)\right) & \text{if } y_p = +1 \\ -\log\left(1 - \sigma\left(\dot{\mathbf{x}}_p^T\mathbf{w}\right)\right) & \text{if } y_p = -1. \end{cases}$$

$$\arg\min_{\mathbf{w}} -\log\left(\sigma\left(\dot{\mathbf{x}}^T\mathbf{w}\right)\right)$$

What we introduced earlier:

$$g_p(\mathbf{w}) = \begin{cases} -\log\left(\sigma\left(\dot{\mathbf{x}}_p^T\mathbf{w}\right)\right) & \text{if } y_p = 1 \\ -\log\left(1 - \sigma\left(\dot{\mathbf{x}}_p^T\mathbf{w}\right)\right) & \text{if } y_p = 0. \end{cases}$$

$$y \in \{0, 1\}$$

$$g_p(w) = -\log(\cdot)\cdot y_p + (1 - y_p)\cdot(-\log(1 - \cdot))$$

$$\arg\min_{\mathbf{w}} -\left(\sigma\left(\dot{\mathbf{x}}^T\mathbf{w}\right)\right)$$

$$\arg\min_{\mathbf{w}} -\left(\tanh\left(\dot{\mathbf{x}}^T\mathbf{w}\right)\right)$$

- We can then form the *Softmax* cost for logistic regression by taking an average of these Log Error costs as

$$g\left(\mathbf{w}\right) = \frac{1}{P}\sum_{p=1}^{P} g_p\left(\mathbf{w}\right).$$

- As with the Cross Entropy cost, it is far more common to express the Softmax cost differently by re-writing the Log Error in a equivalent way as follows.

- First notice that because

$$1 - \sigma\left(x\right) = 1 - \frac{1}{1+e^{-x}} = \frac{1}{1+e^{x}} = \sigma(-x)$$

- The second case in the point-wise cost above can be re-written equivalently as $-\log\left(\sigma\left(-\dot{\mathbf{x}}^{T}\mathbf{w}\right)\right)$ and so the point-wise cost function can be written as

$$g_p\left(\mathbf{w}\right) = \begin{cases} -\log\left(\sigma\left(\dot{\mathbf{x}}_p^{T}\mathbf{w}\right)\right) & \text{if } y_p = +1 \\ -\log\left(\sigma\left(-\dot{\mathbf{x}}_p^{T}\mathbf{w}\right)\right) & \text{if } y_p = -1. \end{cases}$$

$$\rightarrow -\log\left(1-\sigma\left(-\dot{x}_p^{T}w\right)\right)$$

$\sigma(\cdot) = \frac{1}{1+e^{-x}}$

- Now notice that *because we are using the label values* ± 1 we can move the label value in each case *inside* the inner most parenthesis, and we can write *both cases* in a single line as

$$g_p\left(\mathbf{w}\right) = -\log\left(\sigma\left(y_p\,\dot{\mathbf{x}}_p^T\mathbf{w}\right)\right) = \log\frac{1}{\sigma(\cdot)} = \log\left(1+e^{-(\cdot)}\right)$$

- We can re-write the point-wise cost above equivalently as

$$g_p\left(\mathbf{w}\right) = \log\left(1 + e^{-y_p\dot{\mathbf{x}}_p^T\mathbf{w}}\right)$$

- Taking the average of this point-wise cost over all *P* points we have a more common appearance of the *Softmax cost for logistic regression*

$$g(\mathbf{w}) = \frac{1}{P}\sum_{p=1}^{P} g_p\left(\mathbf{w}\right) = \frac{1}{P}\sum_{p=1}^{P}\log\left(1 + e^{-y_p\dot{\mathbf{x}}_p^T\mathbf{w}}\right)$$

- This cost function, like the Cross Entropy cost, is *always convex regardless of the dataset used*.

- Moreover, as we can see here by its derivation, the Softmax and Cross Entropy cost functions are completely equivalent (upon change of label value $y_p = -1$ to $y_p = 0$ and vice-versa) having been built using the same point-wise cost function.

$$g_p(\mathbf{w}) = \begin{cases} -\log\left(\sigma\left(\dot{\mathbf{x}}_p^T\mathbf{w}\right)\right) & \text{if } y_p = 1 \\ -\log\left(1-\sigma\left(\dot{\mathbf{x}}_p^T\mathbf{w}\right)\right) & \text{if } y_p = 0. \end{cases}$$

$$g(\mathbf{w}) = -\frac{1}{P}\sum_{p=1}^{P} y_p \log\left(\sigma\left(\mathring{\mathbf{x}}_p^T\mathbf{w}\right)\right) + \left(1-y_p\right)\log\left(1-\sigma\left(\mathring{\mathbf{x}}_p^T\mathbf{w}\right)\right).$$

cross-entropy

$$g_p(\mathbf{w}) = \begin{cases} -\log\left(\sigma\left(\dot{\mathbf{x}}_p^T\mathbf{w}\right)\right) & \text{if } y_p = +1 \\ -\log\left(1-\sigma\left(\dot{\mathbf{x}}_p^T\mathbf{w}\right)\right) & \text{if } y_p = -1. \end{cases}$$

$$g(\mathbf{w}) = \frac{1}{P}\sum_{p=1}^{P} \log\left(1+e^{-y_p\mathring{\mathbf{x}}_p^T\mathbf{w}}\right).$$

softmax

Implementing and minimizing a modular Softmax cost in `Python`

$$\text{model}\left(\mathbf{x}_p, \mathbf{w}\right) = \dot{\mathbf{x}}_p^T \mathbf{w} \ .$$

```
# compute linear combination of input point
def model(x,w):
    a = w[0] + np.dot(x.T,w[1:])
    return a.T
```

$$g(\mathbf{w}) = \frac{1}{P}\sum_{p=1}^{P}\log\left(1 + e^{-y_p \text{model}(\mathbf{x}_p,\mathbf{w})}\right).$$

```
# the convex softmax cost function
def softmax(w):
    cost = np.sum(np.log(1 + np.exp(-y*model(x,w))))
    return cost/float(np.size(y))
```

6.4 The Perceptron

- As we have seen with logistic regression we treat classification as a particular form of nonlinear regression (employing - with the choice of label values $y_p \in \{-1, +1\}$ - a tanh nonlinearity).
- This results in the learning of a proper nonlinear regressor, and a corresponding *linear decision boundary*

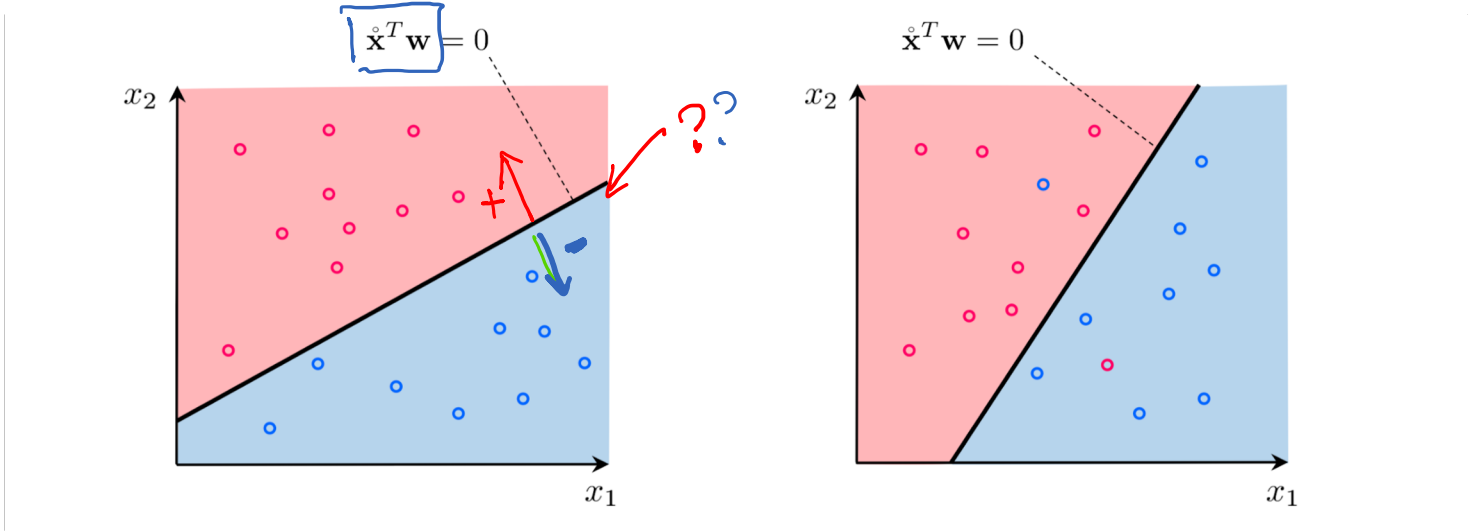
$$\dot{\mathbf{x}}^T \mathbf{w} = 0.$$

- Instead of learning this decision boundary as a result of a nonlinear regression, the *perceptron* derivation described in this Section aims at determining this ideal lineary decision boundary directly.
- While we will see how this direct approach leads back to the *Softmax cost function*.
- Practically speaking **the perceptron and logistic regression *often results in learning the same linear decision boundary***, the perceptron's focus on learning the decision boundary directly provides a valuable new perspective on the process of two-class classification.

The Perceptron cost function

- With two-class classification we have a training set of *P* points $\{(\mathbf{x}_p, y_p)\}_{p=1}^{P}$ - where y_p 's take on just two label values from $\{-1, +1\}$ - consisting of two classes which we would like to learn how to distinguish between automatically.
- As we saw in our discussion of logistic regression, in the simplest instance our two classes of data are largely separated by a *linear decision boundary* with each class (largely) lying on either side.
- This decision boundary, written as $\dot{\mathbf{x}}^T \mathbf{w} = 0$

- This boundary is a *point* when the dimension of the input is $N = 1$, a *line* when $N = 2$, and is more generally for arbitray *N* a *hyperplane* defined in the input space of a dataset.
- This scenario can be best visualized in the case $N = 2$, where we view the problem of classification 'from above' - showing the input of a dataset colored to denote class membership.



- A linear decision boundary cuts the input space into two *half-spaces*, one lying 'above' the hyperplane where $\dot{\mathbf{x}}^T\mathbf{w} > 0$ and one lying 'below' it where $\dot{\mathbf{x}}^T\mathbf{w} < 0$.
- Notice then that a proper set of weights $\mathbf{w}$ define a linear decision boundary that separates a two-class dataset as well as possible with *as many members of one class as possible lying above it, and likewise as many members as possible of the other class lying below it*.

- So our *desired* set of weights define a hyperplane where as often as possible we have that

$$\begin{aligned} \dot{\mathbf{x}}_p^T \mathbf{w} &> 0 \quad \text{if} \quad y_p = +1 \\ \dot{\mathbf{x}}_p^T \mathbf{w} &< 0 \quad \text{if} \quad y_p = -1. \end{aligned}$$

$\Rightarrow$ $y_p \cdot \dot{x}_p^T w$ always positive

- Because of our choice of label values we can consolidate the ideal conditions above into the single equation below

$$-y_p \dot{\mathbf{x}}_p^T \mathbf{w} \ < 0.$$

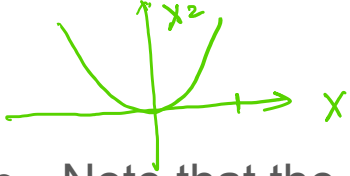
- Again we can do so specifically because we chose the label values $y_p \in \{-1, +1\}$
- Likewise by taking the maximum of this quantity and zero we can then write this ideal condition as

$$\min \ g_p(\mathbf{w}) = \max\left(0, -y_p \dot{\mathbf{x}}_p^T \mathbf{w}\right) = 0$$

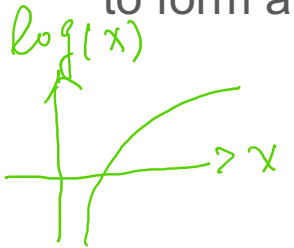
$\dot{x}_p^T w > 0 \Rightarrow y = +1$

$y = -1$

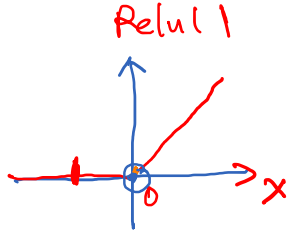
$-y_p \cdot \dot{x}_p^T w > 0$



- Note that the expression $\max\left(0, -y_p \dot{\mathbf{x}}_p^T \mathbf{w}\right)$ is always nonnegative.
- The functional form of this point-wise cost $\max\left(0, \cdot\right)$ is called a *rectified linear unit (ReLU)*.
- Because these point-wise costs are nonnegative and equal *zero* when our weights are tuned correctly, we can take their average over the entire dataset to form a proper cost function as



$$g\left(\mathbf{w}\right) = \frac{1}{P}\sum_{p=1}^{P} g_p\left(\mathbf{w}\right) = \frac{1}{P}\sum_{p=1}^{P} \max\left(0, -y_p \dot{\mathbf{x}}_p^T \mathbf{w}\right).$$



- This cost function goes by many names such as the *perceptron* cost, the *rectified linear unit* cost (or *ReLU cost* for short), and the *hinge cost*.

The smooth softmax approximation to the ReLU cost

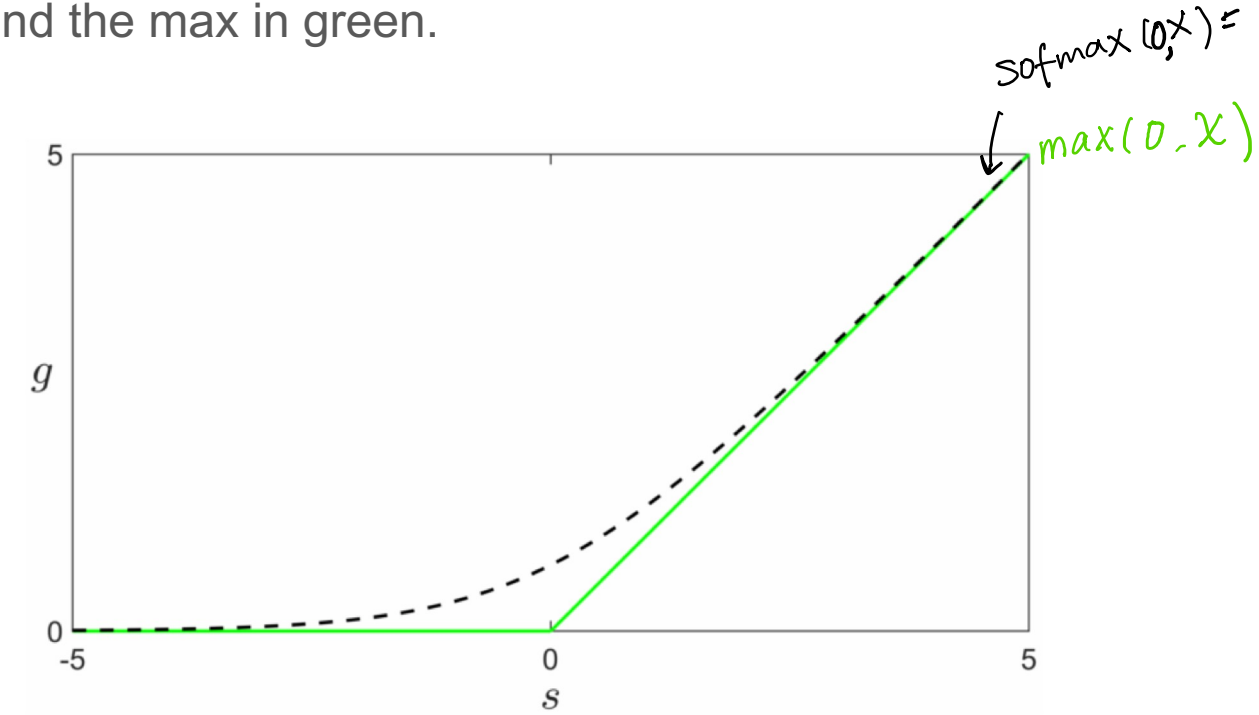
- One popular way of approximating the ReLU cost function is via the *softmax* function defined as

$$\text{soft}\left(s_0, s_1, ..., s_{C-1}\right) = \log\left(e^{s_0} + e^{s_1} + \cdots + e^{s_{C-1}}\right)$$

- Here $s_0,\ s_1,\ \ldots,\ s_{C-1}$ are any C scalar values - which is a generic smooth approximation to the *max* function, i.e.,

$$\text{soft}\left(s_0, s_1, \ldots, s_{C-1}\right) \approx \max\left(s_0, s_1, \ldots, s_{C-1}\right)$$

- The fact that the softmax approximates the maximum can be proved formally
- Below we show the one dimensional comparison. The softmax is shown in dashed black, and the max in green.



- Replace the p^{th} summand of our ReLU cost with its softmax approximation

$$g_p\left(\mathbf{w}\right) = \text{soft}\left(0, -y_p\dot{\mathbf{x}}_p^T\mathbf{w}\right) = \log\left(1 + e^{-y_p\dot{\mathbf{x}}_p^T\mathbf{w}}\right)$$

- The overall cost function is then:

$$g\left(\mathbf{w}\right) = \sum_{p=1}^{P} g_p\left(\mathbf{w}\right) = \sum_{p=1}^{P} \log\left(1 + e^{-y_p\dot{\mathbf{x}}_p^T\mathbf{w}}\right)$$

- This is the *Softmax cost* we saw previously derived from the logistic regression perspective.

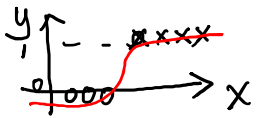
$\vec{x}_p \in \mathbb{R}^{N+1}$ $\vec{x}_p = [1 \; x_p]^T$, p: # sample, $\vec{w} \in \mathbb{R}^{N+1}$, $\vec{w} = [\vec{w}, b]$

decision boundry (hyperlane): $\vec{x}^T \cdot w = 0.5$

cost function

① cross entropy

$y \in \{0, 1\}$



function

$\text{Step}(\vec{x}^T \cdot \vec{w}) \rightarrow \sigma(\vec{x}^T \cdot \vec{w}) = \hat{y} \approx y$

approximated function

cost function

$\min_{\vec{w}} \| \hat{y} - y \|_2^2$ ✗

$$g_p(w) = \begin{cases} -\log(\sigma(\vec{x}^T w)), & y=1 \\ -\log(1-\sigma(\cdot)), & y=0 \end{cases}$$

$$g(\vec{w}) = \frac{1}{p} \sum y_p(-\log(\sigma(\cdot)) + (1-y_p)(-\log(1-\sigma(\cdot))$$

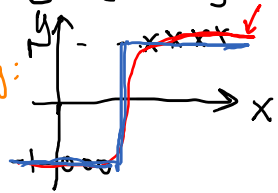
$y=1 \Rightarrow \hat{y} = \sigma(\vec{x}^T w) \approx 1 \Rightarrow g(w) = 0$

$\hat{y} = 0 \Rightarrow g(w) = +\infty$

② softmax cost

$y \in \{-1, 1\}$

decision boundry: $\vec{x}^T \cdot \vec{w} = 0$



$\text{sign}(\vec{x}^T \vec{w}) \rightarrow \tanh(\vec{x}^T \cdot \vec{w}) = \hat{y} \approx y$

$\min_{\vec{w}} \| \hat{y} - y \|_2^2$ ✗

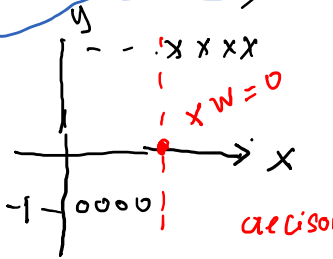
$$g(\vec{w}) = \frac{1}{p} \sum \log(1 + e^{-y_p \vec{x}^T \vec{w}})$$

$$g_p(w) = \begin{cases} -\log(\sigma(\vec{x}^T \cdot w)), & y=1 \\ -\log(1-\sigma(\cdot)), & y=-1 \end{cases}$$

③ percepton cost

$y \in \{-1, 1\}$

$\vec{x}^T \cdot w = 0$



$\vec{x}^T \cdot \vec{w} = 0$

$\begin{cases} \vec{x}^T \cdot \vec{w} > 0 \\ \vec{x}^T \cdot \vec{w} < 0 \end{cases}$

$y = 1 \Rightarrow \boxed{y \cdot \vec{x}^T \cdot \vec{w} > 0}$

$y = -1$

$\min_{\vec{w}} g_p(w) = \max\{0, -y \cdot \vec{x}^T \vec{w}\}$

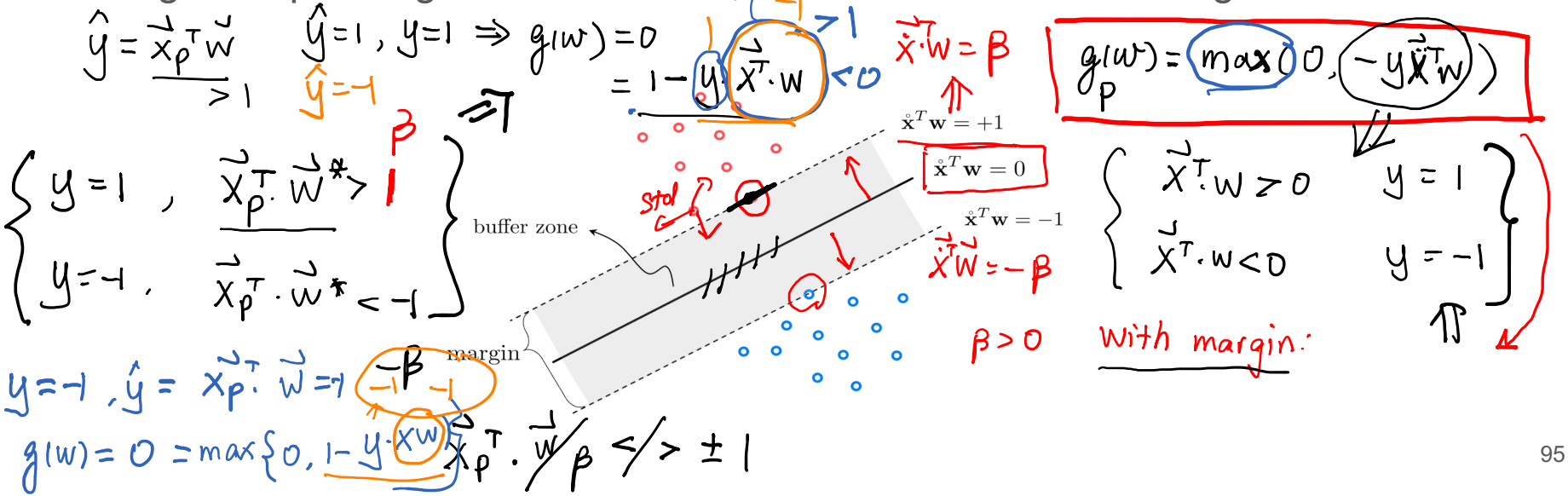
soft ( )

6.5 Support Vector Machines

- In this Section we begin by discussing an often used variation of the perceptron called the *margin perceptron*.
- The margin perceptron will lead us to discuss *Support Vector Machines*, a popular perspective on linear classification again under the assumption of linear separability.
- However we will see that in practice the Support Vector Machines approach does not provide a learned decision boundary that substantially differs from those provided by logistic regression or, likewise, the perceptron.

The margin-perceptron

- Suppose once again that we have a two-class classification training dataset of *P* points $\{(\mathbf{x}_p, y_p)\}_{p=1}^{P}$ with the labels $y_p \in \{-1, +1\}$
- Suppose we are dealing with a two-class dataset that is linearly separable with a known linear decision boundary $\mathring{\mathbf{x}}^T\mathbf{w} = 0.$
- Suppose even further that this decision boundary passes evenly through the region separating our two classes, like the illustrated in the figure below.



- This separating hyperplane creates a buffer between the two classes confined between two evenly shifted versions of itself.
- One version that lies on the positive side of the separator and just touches the class having labels $y_p = +1$ (colored red in the figure) taking the form , $\dot{\mathbf{x}}^T \mathbf{w} = +1$ and one lying on the negative side of it just touching the class with labels $y_p = -1$ (colored blue in the figure) taking the form $\dot{\mathbf{x}}^T \mathbf{w} = -1$
- The translations above and below the separating hyperplane are more generally defined as $\dot{\mathbf{x}}^T \mathbf{w} = +\beta$ and $\dot{\mathbf{x}}^T \mathbf{w} = -\beta$ respectively, where $\beta > 0$

- However by dividing off β in both equations and re-assigning the variables as $\mathbf{w} \longleftarrow \frac{\mathbf{w}}{\beta}$ we can leave out the redundant parameter β and have the two translations as stated $\dot{\mathbf{x}}^T\mathbf{w} = \pm 1$.
- The fact that all points in the *+1* class lie exactly on or on the positive side of $\dot{\mathbf{x}}^T\mathbf{w} = +1$, and all points in the *-1* class lie exactly on or on the negative side of $\dot{\mathbf{x}}^T\mathbf{w} = -1$ can be written formally as the following conditions

$$\begin{aligned} \dot{\mathbf{x}}^T\mathbf{w} &\geq 1 \qquad \text{if} \quad y_p = +1 \\ \dot{\mathbf{x}}^T\mathbf{w} &\leq -1 \qquad \text{if} \quad y_p = -1 \end{aligned}$$

- We can combine these conditions into a single statement by multiplying each by their respective label values, giving the single inequality $y_p \, \dot{\mathbf{x}}^T \mathbf{w} \geq 1$ which can be equivalently written as a point-wise cost

$$g_p \left(\mathbf{w} \right) = \max \left(0, \, 1 - y_p \, \dot{\mathbf{x}}^T \mathbf{w} \right) = 0$$

- Again, this value is always nonnegative. Summing up all *P* equations of the form above gives the *Margin-Perceptron* cost

$$g \left(\mathbf{w} \right) = \sum_{p=1}^{P} \max \left(0, \, \beta - y_p \, \dot{\mathbf{x}}^T \mathbf{w} \right)$$

$\dot{x}^T w > \beta$, $y_p = 1$

$\dot{x}^T w < -\beta$, $y_p = -1$

This additional *1* prevents the issue of a trivial zero solution with the original perceptron

- If the data is indeed linearly separable any hyperplane passing between the two classes will have parameters $\mathbf{w}$ where $g\left(\mathbf{w}\right) = 0$.
- However the margin perceptron is still a valid cost function even if the data is not linearly separable.
- The only difference is that with such a dataset we can not make the criteria above hold for all points in the dataset.
- Thus a violation for the p^{th} point adds the positive value of $1 - y_p \, \dot{\mathbf{x}}^T \mathbf{w}$ to the cost function.

The softmax and Margin-Perceptron cost

- As with the perceptron, one way to smooth out the margin-perceptron here is by replacing the $\max$ operator with softmax . Doing so gives the following approximation:

$$\text{soft}\left(0, 1 - y_p\, \dot{\mathbf{x}}^T \mathbf{w}\right) = \log\left(1 + e^{1 - y_p\, \dot{\mathbf{x}}^T \mathbf{w}}\right)$$

- Right away if we were to sum over all *P* we could form a softmax-based cost function that closely matches the margin-perceptron.

- But note how in the derivation of the margin perceptron the '1' used in the $1 - y_p \left(\dot{\mathbf{x}}^T \mathbf{w}\right)$ component of the cost could have been any number we wanted - it was a normalization factor for the width of the margin and, by convention, we used '1'.
- Instead we could have chosen any value $\epsilon > 0$
- So for all *p* and the Margin-Perceptron equivalently stated as

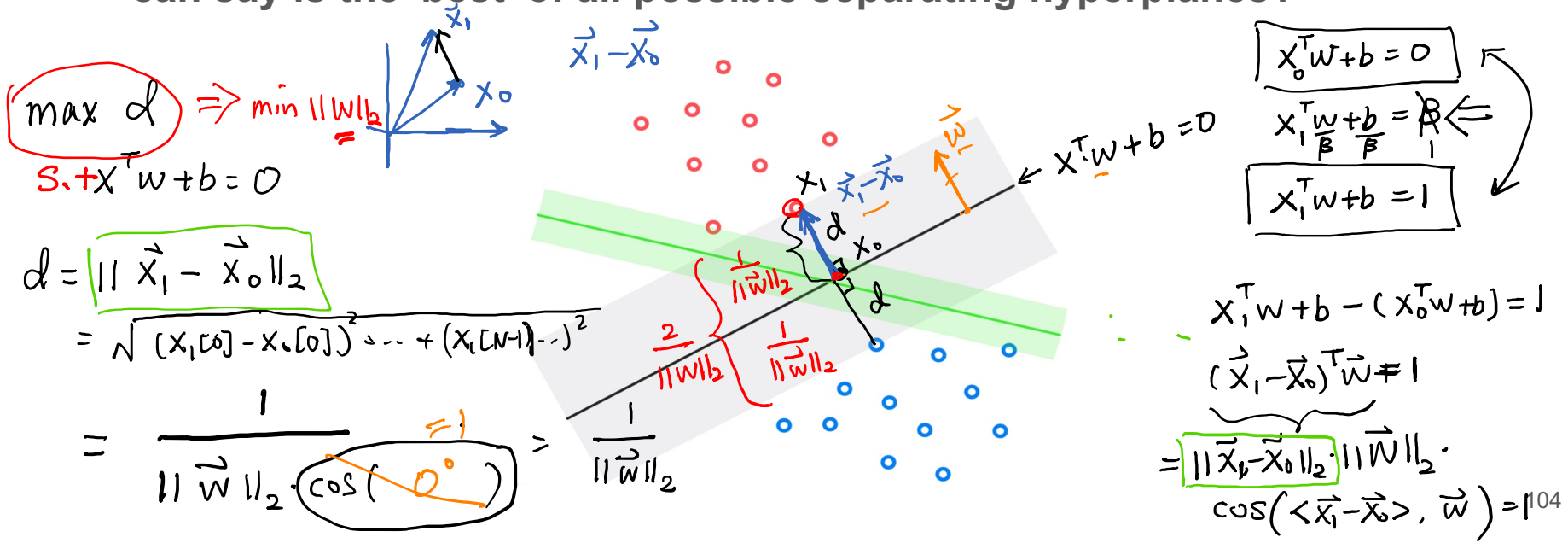
$$g\left(\mathbf{w}\right) = \sum_{p=1}^{P} \max\left(0,\, \epsilon - y_p\, \dot{\mathbf{x}}^T \mathbf{w}\right)$$

- And the softmax version of one summand:

$$\text{soft}\left(0, \epsilon - y_p\, \mathring{\mathbf{x}}_p^T \mathbf{w}\right) = \log\left(1 + e^{\epsilon - y_p\, \mathring{\mathbf{x}}_p^T \mathbf{w}}\right).$$

A quest for the hyperplane with maximum margin

- When two classes of data are linearly separable, infinitely many hyperplanes could be drawn to separate the data.
- Given that both classifiers (as well as any other decision boundary perfectly separating the data) would perfectly classify the data, **is there one that we can say is the 'best' of all possible separating hyperplanes?**



- To recover the maximum margin separating hyperplane, it will be convenient to use our individual notation for the bias and feature-touching weights

$$\text{(bias): } b = w_0$$

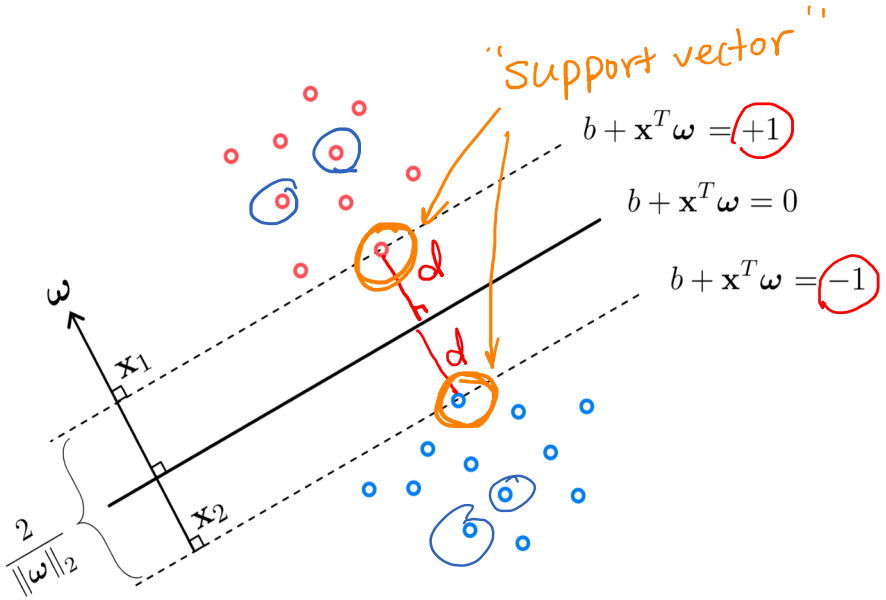
$$\text{(feature-touching weights): } \quad \boldsymbol{\omega} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix}$$

.

- With this notation we can express a linear decision boundary as

$$\dot{\mathbf{x}}^T \mathbf{w} = b + \mathbf{x}^T \boldsymbol{\omega} = 0.$$

- As shown in the figure below the margin can be determined by calculating the distance between any two points (one from each translated hyperplane) both lying on the normal vector $\boldsymbol{\omega}$.



- Denoting by $\mathbf{x}_1$ and $\mathbf{x}_2$ the points on vector $\boldsymbol{\omega}$ belonging to the *positive* and *negative* translated hyperplanes, respectively, the margin is computed simply as the length of the line segment connecting $\mathbf{x}_1$ and $\mathbf{x}_2$, i.e., $\left\|\mathbf{x}_1 - \mathbf{x}_2\right\|_2$
- The margin can be written much more conveniently by taking the difference of the two translates evaluated at $\mathbf{x}_1$ and $\mathbf{x}_2$ respectively, as

$$\left(w_0 + \mathbf{x}_1^T \mathbf{w}\right) - \left(w_0 + \mathbf{x}_2^T \mathbf{w}\right) = \left(\mathbf{x}_1 - \mathbf{x}_2\right)^T \boldsymbol{\omega} = 2$$

- Using the inner product rule, and the fact that the two vectors $\mathbf{x}_1 - \mathbf{x}_2$ and $\boldsymbol{\omega}$ are parallel to each other, we can solve for the margin directly in terms of $\boldsymbol{\omega}$, as

$$\left\| \mathbf{x}_1 - \mathbf{x}_2 \right\|_2 = \frac{2}{\|\boldsymbol{\omega}\|_2}$$

- Therefore finding the separating hyperplane with maximum margin is equivalent to **finding the one with the smallest possible normal vector** $\boldsymbol{\omega}$.

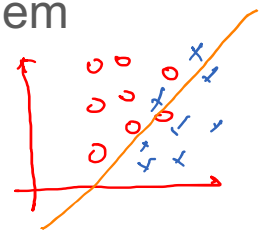
The hard-margin and soft-margin SVM problems

- In order to find a separating hyperplane for the data with minimum length normal vector we can simply combine...
- ...our desire to minimize $\|\boldsymbol{\omega}\|_2^2$ ...
- ...subject to the constraint that the hyperplane perfectly separates the data (given by the margin criterion described above).
- This gives the so-called *hard-margin support vector machine* problem

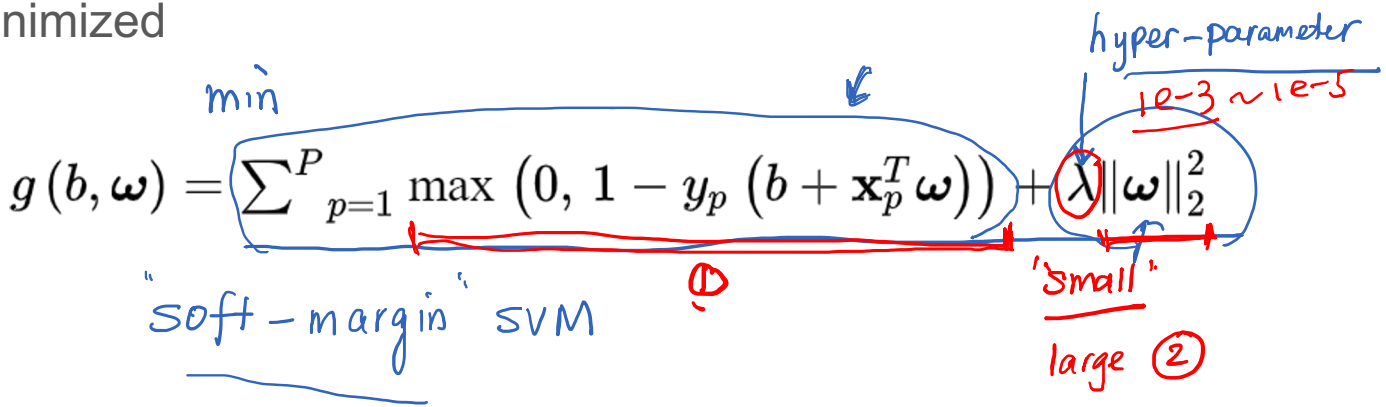
$$\underset{b,\,\boldsymbol{\omega}}{\text{minimize}} \;\; \|\boldsymbol{\omega}\|_2^2$$

Feature-touching weight

$$\text{subject to} \;\; \max\left(0,\, 1 - y_p\left(b + \mathbf{x}_p^T\boldsymbol{\omega}\right)\right) = 0, \quad p = 1, \ldots, P.$$



- While there are *constrained optimization* algorithms that are designed to solve problems like this as stated.
- We can also solve the hard-margin problem by *relaxing* the constraints and forming an unconstrained formulation of the problem.
- To do this we merely bring the constraints up, forming a single cost function to be minimized

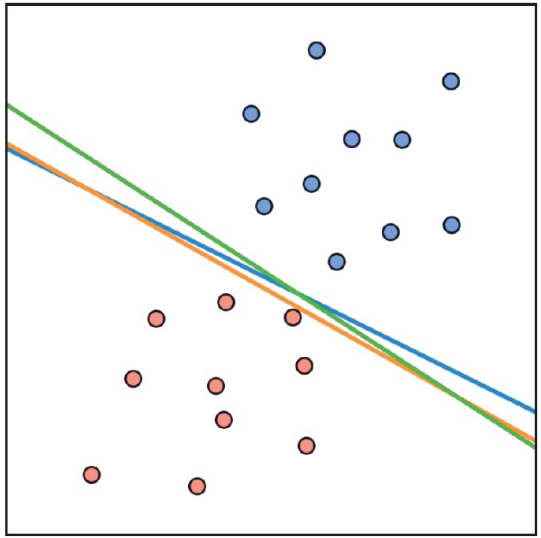


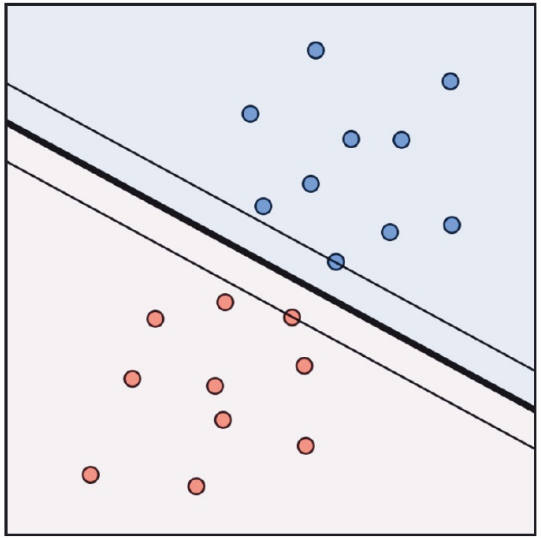
- Here the parameter $\lambda \geq 0$ is called a penalty or regularization parameter
- When λ is set to a small positive value we put more 'pressure' on the cost function to make sure the constraints

$$\max\left(0,\, 1 - y_p\left(b + \mathbf{x}_p^T\boldsymbol{\omega}\right)\right) = 0, \quad p = 1, \ldots, P$$

- This *regularized* form of the Margin-Perceptron cost function is referred to as the *soft-margin support vector machine cost*.

- In this example we compare the support vector machine decision boundary learned to three boundaries learned via the margin-perceptron.
- All of the recovered boundaries perfectly separate the two classes, but the support vector machine decision boundary is the one that provides the maximum margin (right panel)...





`sklearn.linear_model`: Linear Models

The `sklearn.linear_model` module implements a variety of linear models.

User guide: See the Linear Models section for further details.

The following subsections are only rough guidelines: the same estimator can fall into multiple categories, depending on its parameters.

Linear classifiers

`linear_model.LogisticRegression`([penalty, ...])	Logistic Regression (aka logit, MaxEnt) classifier.
`linear_model.LogisticRegressionCV`(*[, Cs, ...])	Logistic Regression CV (aka logit, MaxEnt) classifier.
`linear_model.PassiveAggressiveClassifier`(*)	Passive Aggressive Classifier.
`linear_model.Perceptron`(*[, penalty, alpha, ...])	Linear perceptron classifier.
`linear_model.RidgeClassifier`([alpha, ...])	Classifier using Ridge regression.
`linear_model.RidgeClassifierCV`([alphas, ...])	Ridge classifier with built-in cross-validation.
`linear_model.SGDClassifier`([loss, penalty, ...])	Linear classifiers (SVM, logistic regression, etc.) with SGD training.
`linear_model.SGDOneClassSVM`([nu, ...])	Solves linear One-Class SVM using Stochastic Gradient Descent.

https://scikit-learn.org/stable/modules/classes.html#module-sklearn.linear_model

class `sklearn.linear_model.`**`LogisticRegression`**(*penalty='l2'*, ***, *dual=False*, *tol=0.0001*, *C=1.0*, *fit_intercept=True*, *intercept_scaling=1*, *class_weight=None*, *random_state=None*, *solver='lbfgs'*, *max_iter=100*, *multi_class='auto'*, *verbose=0*, *warm_start=False*, *n_jobs=None*, *l1_ratio=None*) [source]

Logistic Regression (aka logit, MaxEnt) classifier.

In the multiclass case, the training algorithm uses the one-vs-rest (OvR) scheme if the 'multi_class' option is set to 'ovr', and uses the cross-entropy loss if the 'multi_class' option is set to 'multinomial'. (Currently the 'multinomial' option is supported only by the 'lbfgs', 'sag', 'saga' and 'newton-cg' solvers.)

This class implements regularized logistic regression using the 'liblinear' library, 'newton-cg', 'sag', 'saga' and 'lbfgs' solvers. **Note that regularization is applied by default**. It can handle both dense and sparse input. Use C-ordered arrays or CSR matrices containing 64-bit floats for optimal performance; any other input format will be converted (and copied).

The 'newton-cg', 'sag', and 'lbfgs' solvers support only L2 regularization with primal formulation, or no regularization. The 'liblinear' solver supports both L1 and L2 regularization, with a dual formulation only for the L2 penalty. The Elastic-Net regularization is only supported by the 'saga' solver.

Read more in the User Guide.

Parameters:

penalty : *{'l1', 'l2', 'elasticnet', None}, default='l2'*

Specify the norm of the penalty:

- `None`: no penalty is added;
- `'l2'`: add a L2 penalty term and it is the default choice;
- `'l1'`: add a L1 penalty term;
- `'elasticnet'`: both L1 and L2 penalty terms are added.

Q/A

https://analyticsarora.com/8-unique-machine-learning-interview-questions-on-svm/#Support-Vector-Machines-ML-Interview-Questions-Answers

What are support vectors in SVMs?

What are Support Vectors in SVMs?

Support vectors are the instances located on the margin of the hyperplane in an SVM. For SVMS, the decision boundary is determined solely by using the support vectors. As a result, any instance that is not a support vector (not on the margin boundaries) has no influence on the decision boundary.

What are hard margin SVMs and soft margin SVMs?

What are Hard Margin SVMs and Soft Margin SVMs?

Hard Margin SVMs are those that work only if the data is linearly separable. They have a 'hard' constraint on them. Hence these types of SVMs are quite sensitive to outliers.

Soft Margin SVMs find a good balance between keeping the margins as large as possible while limiting the margin violation i.e. instances that end up in the middle of margin or even on the wrong side. Their constraint is 'soft.' However, by no means is it inaccurate or non-optimized.

What is the function of the hyperparameter lambda?

$$g\left(b, \boldsymbol{\omega}\right) = \sum^{P}{}_{p=1} \max\left(0,\, 1 - y_p\left(b + \mathbf{x}_p^T\boldsymbol{\omega}\right)\right) + \lambda\|\boldsymbol{\omega}\|_2^2$$

A: It helps us tune how much we want to put the pressure on points lying either inside of our margin or complete misclassification.

What is the function of the hyperparameter lambda?

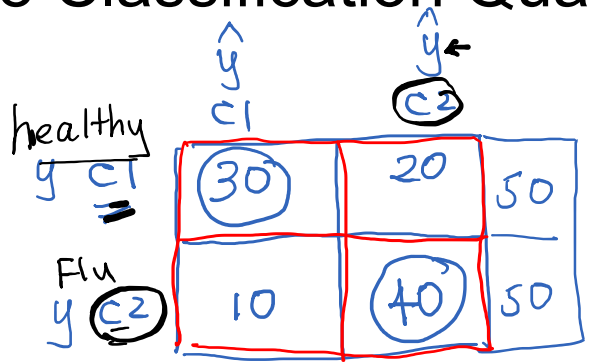
$$g\left(b, \boldsymbol{\omega}\right) = \sum\nolimits_{p=1}^{P} \max\left(0,\, 1 - y_p\left(b + \mathbf{x}_p^T\boldsymbol{\omega}\right)\right) + \lambda\|\boldsymbol{\omega}\|_2^2$$

6.8 Classification Quality Metrics

① Accuracy := $\frac{\#\text{instance correctly classified model}(\vec{w}^*)}{P}$

$= \frac{\sum I(\hat{y}_p == y_p)}{P}$

② confusion matrix/table



P = 100

#c1 = 50

#c2 = 50

$A = \frac{30+40}{100} = 70\%$

① precision Flu detection = $\frac{\#\text{true prediction}}{\#\text{predicted}} = \frac{40}{20+40} = \frac{4}{6} = 77\%$

② recall Flu detection = #truely identified / #true Flu = 40/50 = 80%

- Suppose we have an optimal set of weights found by minimizing a classification cost function (employing by default label values $y_p \in \{-1, +1\}$).
- Denote this set of optimal weights $\mathbf{w}^\star$ then we can write our fully tuned linear model as

$$\text{model}\left(\mathbf{x}, \mathbf{w}^\star\right) = \dot{\mathbf{x}}^T \mathbf{w}^\star = w_0^\star + x_1 w_1^\star + x_2 w_2^\star + \cdots + x_N w_N^\star.$$

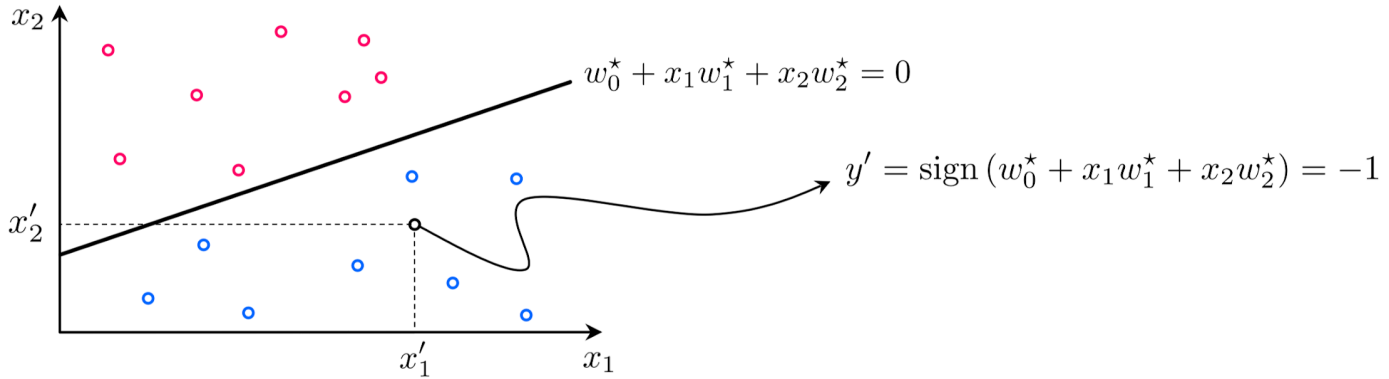
- This fully trained model defines an optimal decision boundary for the training dataset which takes the form

$$\text{model}\left(\mathbf{x}, \mathbf{w}^\star\right) = 0.$$

- To predict the label y of an input $\mathbf{x}$, we then process this model through an appropriate step function.
- This step function is conveniently defined by the $\text{sign}(\cdot)$ function and the predicted label for $\mathbf{x}$ is given as

$$\text{sign}\left(\text{model}\left(\mathbf{x}, \mathbf{w}^{\star}\right)\right) = y\,.$$

- This evaluation will always take on values ± 1 if $\mathbf{x}$ does not lie *__exactly__* on the decision boundary.
- If it does lie directly on the boundary we assign a random value from ± 1 to the point.
- It simply computes which side of the decision boundary the input $\mathbf{x}$ lies on.
- If it lies 'above' the decision boundary then $y = +1$, and if 'below' then $y = -1$. This is illustrated in the figure below.



Judging the quality of a trained model using *accuracy*

- Once we have successfully minimized the cost function for linear two-class classification it can be a delicate matter to determine our trained model's quality.
- The simplest metric for judging the quality of a fully trained model is to simply *count the number of misclassifications* it forms over our training dataset.
- This is a raw count of the number of training datapoints $\mathbf{x}_p$ whose true label y_p is predicted *incorrectly* by our trained model.

- To compare the point $\mathbf{x}_p$'s predicted label $\hat{y}_p = \text{sign}\left(\text{model}\left(\mathbf{x}_p, \mathbf{w}^\star\right)\right)$ and true label y_p we can use an identity function $\mathcal{I}(\cdot)$ and compute

$$\mathcal{I}\left(\hat{y}_p, y_p\right) = \begin{cases} 0 & \text{if } \hat{y}_p, = y_p \\ 1 & \text{if } \hat{y}_p, \neq y_p. \end{cases}$$

- Summing all *P* points gives the total number of misclassifications of our trained model

$$\text{number of misclassifications} = \sum_{p=1}^{P} \mathcal{I}\left(\hat{y}_p, y_p\right).$$

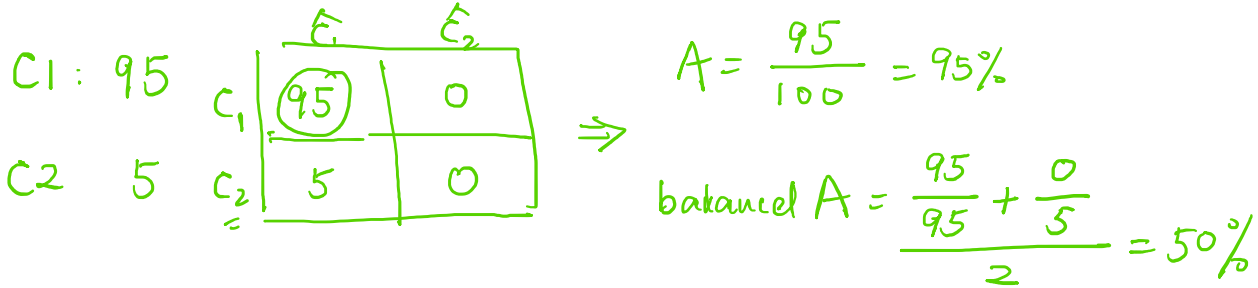
Using this we can also compute the *accuracy* - denoted $\mathcal{A}$ - of a trained model.

This is simply the percentage of training dataset whose labels are correctly predicted by the model.

$$\mathcal{A} = 1 - \frac{1}{P}\sum_{p=1}^{P}\mathcal{I}\left(\hat{y}_p,\, y_p\right).$$

The accuracy ranges from 0 (no points are classified correctly) to 1 (all points are classified correctly).

Judging the quality of a trained model using *balanced accuracy*



- If one class makes up 95% percent of all data points, a naive classifier that blindly assigns the label of the majority class to *every training point* achieves an accuracy of 95%.
- But here misclassifying 5% amounts to *completely misclassifying an entire class of data*.
- The simplest way to improve the accuracy metric to compute accuracy on *each class individually* and combine the resulting metrics.

Denote the *indices* of those points with labels $y_p = +1$ and $y_p = -1$ as Ω_{+1} and Ω_{-1} respectively.

Then we compute the number of misclassifications on each class individually as

$$\text{number of misclassifications on } +1 \text{ class} = \sum_{p \in \Omega_{+1}} \mathcal{I}\left(\hat{y}_p, y_p\right)$$
$$\text{number of misclassifications on } -1 \text{ class} = \sum_{p \in \Omega_{-1}} \mathcal{I}\left(\hat{y}_p, y_p\right)$$

- The accuracy on each class individually can then be likewise computed as (denoting the accuracy on class +1 and -1 individually as $\mathcal{A}_{+1}$ and $\mathcal{A}_{-1}$ respectively)

$$\mathcal{A}_{+1} = 1 - \frac{1}{|\Omega_{+1}|} \sum_{p \in \Omega_{+1}} \mathcal{I}\left(\hat{y}_p, y_p\right)$$

$$\mathcal{A}_{-1} = 1 - \frac{1}{|\Omega_{-1}|} \sum_{p \in \Omega_{-1}} \mathcal{I}\left(\hat{y}_p, y_p\right)$$

- Note here the $|\Omega_{+1}|$ and $|\Omega_{-1}|$ denote the number of points belonging to the +1 and -1 class respectively.

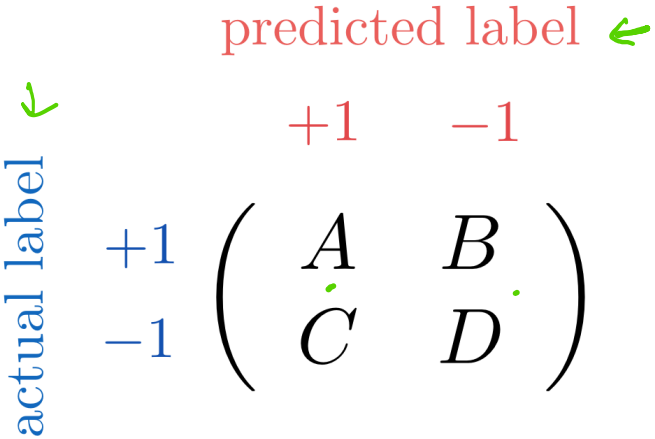
- We can then combine these two metrics into a single value by *taking their average*. This combined metric is called *balanced accuracy* (which we denote as $\mathcal{A}_{\text{balanced}}$)

$$\mathcal{A}_{\text{balanced}} = \frac{\mathcal{A}_{+1} + \mathcal{A}_{-1}}{2}.$$

- Suppose that the +1 class is the majority then while the *overall accuracy* is $\mathcal{A} = 95\%$, the accuracy on each class individually is $\mathcal{A}_{+1} = 1$ and $\mathcal{A}_{-1} = 0$ respectively.
- These metrics accurately reflect the fact that the naive classifier correctly classifies the entire +1 class, but incorrectly classifies the entire -1 class.
- Their average - the balanced accuracy metric - also captures this fact since it then takes on the value $\mathcal{A}_{\text{balanced}} = 0.5$

The confusion matrix and additional quality metrics

- Additional metrics for judging the quality of a trained model for two class classification can be formed using the *confusion matrix*.
- A confusion matrix is a simple lookup table where classification results are broken down by actual (across rows) and predicted (across columns) class membership.



- Our *accuracy* metric can be expressed in terms of the confusion matrix quantities as

$$\mathcal{A} = \frac{A+D}{A+B+C+D},$$

- As can our accuracy on each individual class

$$\mathcal{A}_{+1} = \frac{A}{A+C}$$
$$\mathcal{A}_{-1} = \frac{D}{B+D}.$$

- The *balanced accuracy* metric can likewise be expressed as

$$\mathcal{A}_{\text{balanced}} = \frac{1}{2}\frac{A}{A+C} + \frac{1}{2}\frac{D}{B+D}.$$

Summary

- Logistic regression → cross entropy $\hat{y} \in \{0, 1\}$
- Perceptron $y \in \{-1, 1\}$ → perceptron → softmax cost (max ⇒ soft)
- SVM margin perceptron
- Quality metrics for two-class classification

- Concepts: Cross entropy, Softmax, perceptron, linear boundary, SVM, margin perceptron, regularization