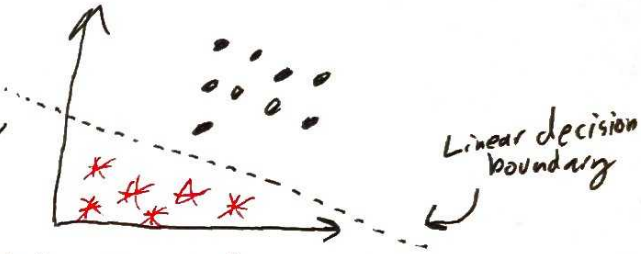
Perceptrons: Linear Classifiers and Kernels

In this unit, we will explore the perceptron learning algorithm, first with simple linear classification, and then later with the kernel trick to unlock new predictive abilities.

Linear Classification: In linear classification, we restrict our decision boundary to just be a single linear equation. (In 2-D, this makes our boundary a single line, in 3-D it is a single plane, and in higher dimensions it is a hyperplane.)



With such a restricted class of decision boundaries, we can represent our prediction rule with only a small amount of numbers, generally called parameters or weights. We will put our weights into a single vector called w.

$$\boxed{\text{Prediction Rule } f(x) \text{ using weights } w\text{: predict } \operatorname{sign}(\langle w, x\rangle)}$$

Note: With such a simple boundary, we only have the ability to do binary classification where we only have 2 possible labels. So when working with linear classifiers, we generally must designate one label as positive $(y=+1)$ and designate the other possible label as negative $(y=-1)$.

Note: The function sign is defined as $\operatorname{sign}(z) = \begin{cases} +1 & \text{if } z>0 \\ 0 & \text{if } z=0 \\ -1 & \text{if } z<0 \end{cases}$

If we end up with a prediction of 0, we can use randomized tie-breaking

If our data has labels ~~other~~ that aren't just the set $\{-1, +1\}$, we will have to pick and follow~~ing~~ a mapping between the labels we care about (in our collected data and task) and the labels necessary for this classifier to work (-1 and $+1$)

What is w?

w is a vector containing the weights of the linear classifier. It is also used to parameterize our prediction rule $f(x)$, such that the goal of our learning algorithm is to find a good w.

$\boxed{\text{Training Data}} \rightarrow \boxed{\text{Learning Algorithm}} \rightarrow$ Parameter w for a prediction rule $f_w(x)$

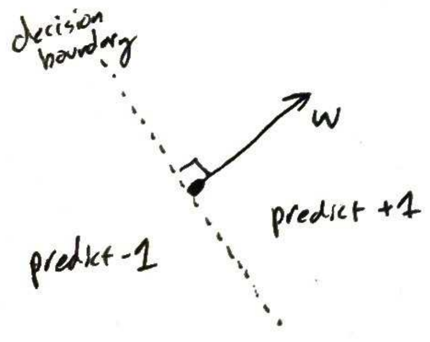
Just-the-math interpretation: w gives a weight to each coordinate of x towards predicting the positive class of the negative class.

$$f_w(x) = \text{sign}(\langle w, x \rangle) = \text{sign}\Big(\sum_{i=1}^{d} \underbrace{w^{(i)} x^{(i)}}\Big)$$

Here $w^{(i)}$ and $x^{(i)}$ refer to the i^{th} coordinates of w and x.

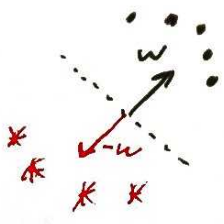
each coordinate x_i is multiplied by w_i and we check whether this sum is positive or negative.

Geometric interpretation: Our decision boundary lies on the points x where $\langle w, x \rangle$ is neither positive or negative, the points where $\langle w, x \rangle = 0$. This means the decision boundary lies orthogonal to w, and we can call ~~w~~ the normal vector of our classifier



Another view is that we want $\langle w, x \rangle > 0$ for x in the positive class, and $\langle w, x \rangle < 0$ for x in the negative class.

This means the vector w should generally point towards our positive class examples, and the vector $-w$ should generally point towards our negative class examples



Note: as written here, the boundary always goes through the origin, since $\langle w, 0 \rangle = 0$ for any w. We will find a way to relax this restriction later.

The Perceptron Algorithm: This is just one way (of many) for finding a good w.

1. Initialize $w_1 = 0$

- the subscript in w_1 here is just to indicate our first vector for w. When we update our weights we will have w_2. These subscripts will be important for a proof later, but coding this algorithm would generally just overwrite a variable w.

2. For $t = 1, 2, 3, \dots, T$

   If $y_t \langle w_t, x_t \rangle \leq 0$
   then
   $w_{t+1} = w_t + y_t x_t$
   else
   $w_{t+1} = w_t$

- We can run this loop as long as we want, for any T number of iterations. When we have processed all the training data ($t == n$) we call that one pass. We then just start repeating the data set and doing more passes until we decide to stop.

  $\boxed{\text{i.e. You can let } (x_t, y_t) = (x_{t \bmod n}, y_{t \bmod n}) \text{ when } t > n}$

- The body of this loop could be summarized as "If $y_t \langle w_t, x_t \rangle \leq 0$, then update w."

What is this condition "$y_t \langle w_t, x_t \rangle \leq 0$"?

This is true under two cases:

- Case 1: $y_t = +1$, and $\langle w_t, x_t \rangle \leq 0$

  so $\text{sign}(\langle w_t, x_t \rangle) \neq +1$, and x_t is not on the "positive side" of w's hyperplane.

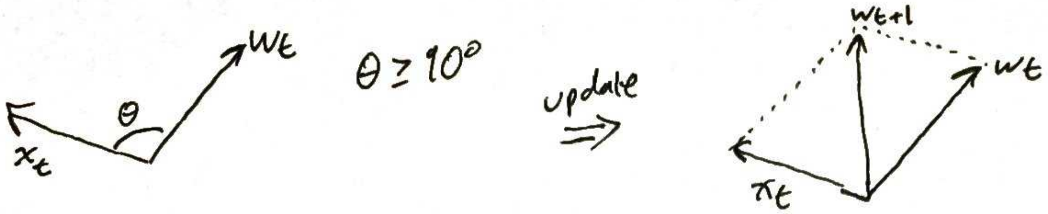
- Case 2: $y_t = -1$, and $\langle w_t, x_t \rangle \geq 0$

  so $\text{sign}(\langle w_t, x_t \rangle) \neq -1$, and x_t is not on the "negative side" of w's hyperplane

In both cases $y_t \langle w_t, x_t \rangle \leq 0$ means $\text{sign}(\langle w_t, x_t \rangle) \neq y_t$ which further means the current weights w_t are not correctly predicting x_t's label. Thus this is a good time to update w before moving on the next data point.

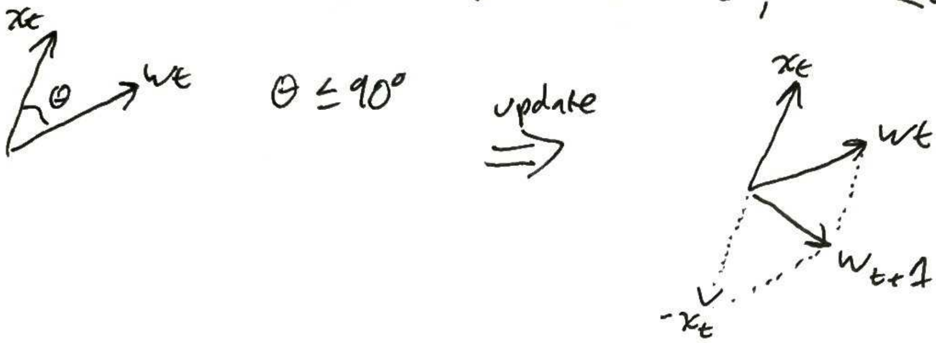
Geometric Intepretation of the update: $w_{t+1} = w_t + y_t x_t$

- Case 1, $y_t = +1$, $w_{t+1} = w_t + x_t$, $\langle w_t, x_t \rangle \leq 0$



w_{t+1} moves "closer to" the direction of x_t, moving x_t towards the positive side of the decision boundary

- case 2, $y_t = -1$, $w_{t+1} = w_t - x_t$, $\langle w_t, x_t \rangle \geq 0$



w_{t+1} moves "away from" the direction of x_t, moving x_t towards the negative side of the decision boundary

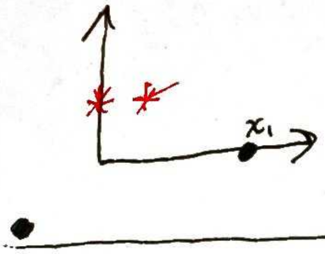
In both cases, this update moves us towards a "better" w.

Exercise: Suppose w_t makes a mistake on (x_t, y_t), and we update w_t to w_{t+1} as above. Is it possible for w_{t+1} to also make a mistake on (x_t, y_t)?

What purpose might we have in doing multiple passes over the training data in this algorithm?

Example: Training data $((4,0), \bullet)$ $((1,1), *)$ $((0,1), *)$ $((-2,-2), \bullet)$

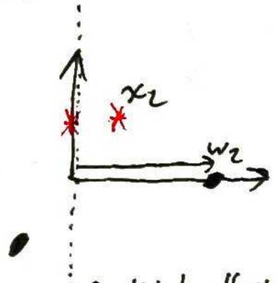
Arbitrarily, we will make $\bullet$ into the positive class ($y=+1$) and $*$ into the negative class ($y=-1$)



Round 1: $w_1 = (0,0)$ $(x_1, y_1) = ((4,0), 1)$

$y_1 \langle w_1, x_1 \rangle = 0 \leq 0$, so update

$w_2 = w_1 + y_1 x_1 = (4,0)$

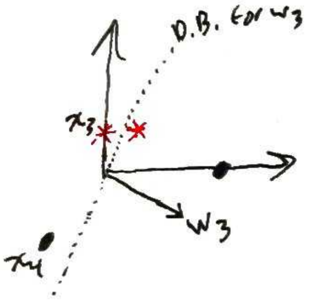


D.B, slightly offset for illustration

Round 2: $t=2$, $w_2 = (4,0)$ $(x_2, y_2) = ((1,1), -1)$

$y_2 \langle w_2, x_2 \rangle = -4 \leq 0$, so update

$w_3 = w_2 + y_2 x_2 = (3,-1)$



Round 3: $t=3$, $w_3 = (3,-1)$ $(x_3, y_3) = ((0,1), -1)$

$y_3 \langle w_3, x_3 \rangle = 1 > 0$, so no update

$w_4 = w_3$

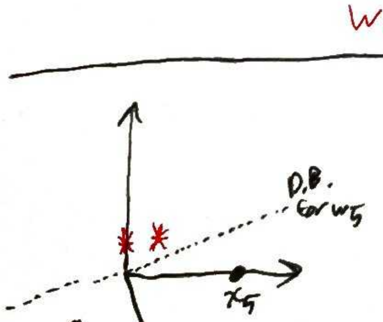
[Same picture $w_4 = w_3$]

Round 4: $t=4$, $w_4 = (3,-1)$ $(x_4, y_4) = ((-2,-2), 1)$

$y_4 \langle w_4, x_4 \rangle = -4 \leq 0$, so update

$w_5 = w_4 + y_4 x_4 = (1,-3)$

We have done one pass over the training data, but we can keep going.



Round 5: $t=5$, $w_5 = (1,-3)$ $(x_5, y_5) = (x_1, y_1) = ((4,0), 1)$

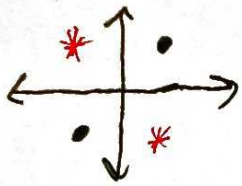
for $t > n$, we just repeat the training data again for another pass.

$y_5 \langle w_5, x_5 \rangle = 4 > 0$, so no update

$w_6 = w_5$

... And so on... In this case, we could stop after round 4. w_5 correctly separates the data, and the algorithm has <u>converged</u>: no further updates will be made.

Example Training Data $((1,1),1)$ $((1,-1),-1)$ $((-1,1),-1)$ $((-1,-1),1)$

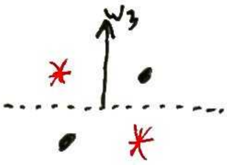


Round 1: $y_1\langle w_1, x_1\rangle = 0 \le 0$ so update. $\boxed{\text{Recall: } w_1 = \text{all zeroes}}$

$w_2 = w_1 + y_1 x_1 = (1,1)$

Round 2: $y_2\langle w_2, x_2\rangle = 0 \le 0$ so update

$w_3 = w_2 + y_2 x_2 = (0,2)$



Round 3: $y_3\langle w_3, x_3\rangle = -2 \le 0$ so update

$w_4 = w_3 + y_3 x_3 = (1,1)$

Round 4: $y_4\langle w_4, x_4\rangle = -2 \le 0$ so update

$w_5 = w_4 + y_4 x_4 = (0,0)$

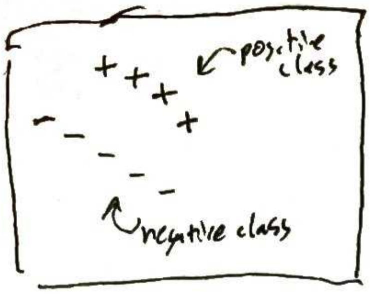
... and so on...

Here, the algorithm never converges (by coincidence here, after one pass we end up with $w =$ all zeroes, so all future rounds resemble these 4, so it is obvious it never converges)

When does the perceptron algorithm converge? Can we prove it?

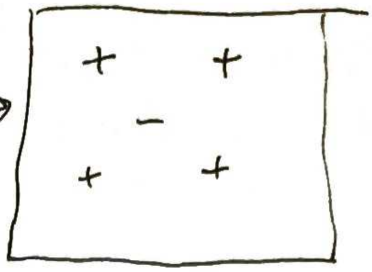
Let us get a taste of learning theory, the theoretical exploration of the properties of learning algorithms.

Linear Separability: We say data is linearly separable if there exists a hyperplane separating the two classes.



Linearly separable

Not Linearly separable



Note: for now, we will actually mean linearly separable through the origin, which requires a hyperplane to exist that passes through the origin and separates the data.

Measure of Separability: The Margin

For a vector w, and a training data set S, the margin of w w.r.t. S is:

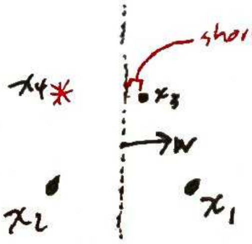
$$\gamma = \min_{(x,y)\in S} \frac{|\langle w, x\rangle|}{\|w\|}$$

Example: $\boxed{\begin{array}{l} S = \Big(((1,-1),1), ((-1,-1),1), ((0.05,0),1)\ ((-1,0),-1) \Big) \\ w = (1,0) \end{array}}$

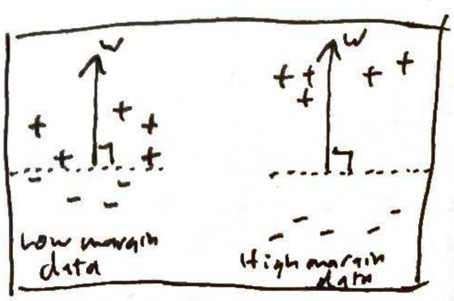
$$\frac{|\langle w, x_1\rangle|}{\|w\|} = \frac{1}{1} = 1 \qquad \frac{|\langle w, x_2\rangle|}{\|w\|} = \frac{1}{1} = 1 \qquad \frac{|\langle w, x_3\rangle|}{\|w\|} = \frac{0.05}{1} = 0.05$$

$$\frac{|\langle w, x_4\rangle|}{\|w\|} = \frac{1}{1} = 1,$$ so the margin $\gamma = 0.05$

Geometrically, the margin is the shortest length seen after projecting the data in S along the direction of w



shortest distance to the decision boundary, this distance is the margin.



Theorem: If the training data is linearly separable with a margin of γ, and if $\|x_i\| \le 1$ for all i in the training set, then the perceptron makes $\le 1/\gamma^2$ mistakes (i.e. it converges after at most $1/\gamma^2$ updates to w)

Notes:
- A lower margin $\Rightarrow$ potentially more mistakes/updates
- $1/\gamma^2$ might be larger than n, we might need more than one pass over the data to converge
- If we have $\|x_i\| \ge 1$, we could adapt this statement by scaling down our data, or by changing the proof. Either way a similar (but slightly different) bound applies.

Proof: By our assumption of separability, there exists a vector w^* that separates the data with margin γ. We can choose w^* s.t. $\|w^*\| = 1$

First, let's show that if there is a mistake at round t, then we will have $\langle w_{t+1}, w^* \rangle \ge \langle w_t, w^* \rangle + \gamma$

On a mistake $w_{t+1} = w_t + y_t x_t$, so

$$\langle w_{t+1}, w^* \rangle = \langle w_t + y_t x_t, w^* \rangle = \langle w_t, w^* \rangle + y_t \langle x_t, w^* \rangle$$

uses margin assumption

$$\ge \langle w_t, w^* \rangle + \gamma$$

$\boxed{y_t \langle x_t, w^* \rangle \ge \gamma \text{ since } \frac{|\langle x_t, w^* \rangle|}{\|w^*\|} \ge \gamma,\ \|w^*\|=1 \text{ and } y_t \langle x_t, w^* \rangle > 0}$

uses separability

Next, let's show that if there is a mistake at round t, $\|w_{t+1}\|^2 \le \|w_t\|^2 + 1$

$$\|w_{t+1}\|^2 = \|w_t + y_t x_t\|^2 = \|w_t\|^2 + \underbrace{y_t^2 \|x_t\|^2}_{\le 1,\text{ since } \|x_t\| \le 1} + \underbrace{2 y_t \langle w_t, x_t \rangle}_{\le 0,\text{ since there is a mistake in round } t}$$

$$\le \|w_t\|^2 + 1 + 0 = \|w_t\|^2 + 1$$

Now suppose there are K mistakes. After K mistakes, $\|w_{t+1}\| \le \sqrt{K}$ and $\langle w_{t+1}, w^* \rangle \ge \gamma K$. We can also see $\frac{\langle w_{t+1}, w^* \rangle}{\|w^*\| \|w_{t+1}\|} = \cos\theta \le 1$

property of inner products

So, we must have $\frac{\gamma K}{\sqrt{K}} \le 1$, which implies $K \le \frac{1}{\gamma^2}$, and completes the proof.

What if the data is not linearly seperable?

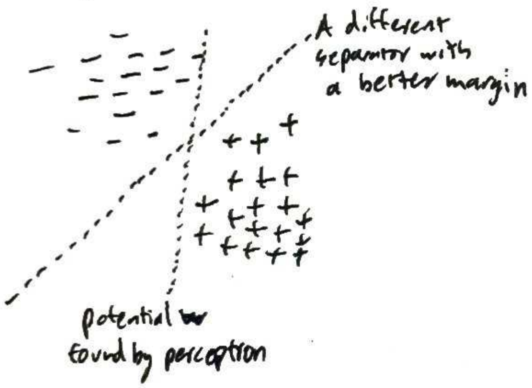
Then we cannot expect to have any linear classifier learning algorithm that gets zero mistakes, which clearly means the perceptron algorithm will not converge (we can always do more passes, find more mistakes, and keep updating w.).

Ideally we'd want to find a w that makes the minimum number of mistakes, but it turns out this problem is <u>NP Hard</u>.

However, we can still apply the perceptron algorithm: It might not find the best w, and we will have to stop sometime since it won't converge, but if the data is almost linearly separable, we can expect it to do alright on most of the data that is far from the boundary of the two classes, with some mistakes where the classes meet.

Extensions to the Perceptron Algorithm:

<u>Better Margin</u>: The perceptron algorithm converges as soon as it finds a w that separates the data, but we may want to find a max-margin separator



Max-margin linear classifiers can be computed with by using Support Vector Machines (SVMs), which are outside the scope of this course

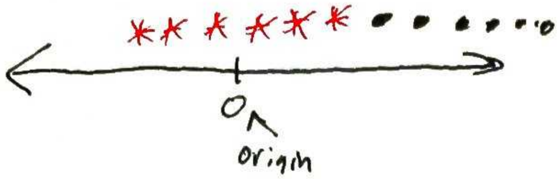
<u>Separating not through the origin</u>: So far our decision boundaries looked like $\sum_i w^{(i)} x^{(i)} = 0$, which passes through the origin. For more general boundaries, we need an equation like $\sum_i w^{(i)} x^{(i)} = b$ for some constant b, which makes the prediction rule look like $f_{w,b}(x) = \text{sign}(\langle w, x\rangle - b)$. How can we learn this extra parameter b?

We can actually just use our perceptron algorithm, if we make a simple modification to our training data: replace each point (x_i, y_i) with a different point (z_i, y_i) where $z_i = \begin{bmatrix} 1 \\ x_i \end{bmatrix}$, i.e. we add an extra dimension to the feature vectors, but that extra dimension always equals the constant 1.

Math interpretation: $\sum_{i=1}^{d+1} w^{(i)} z^{(i)} = w^{(1)} \cdot 1 + \sum_{i=2}^{d+1} w^{(i)} x^{(i-1)}$, which looks like $b + \sum w^{(i)} x^{(i)}$

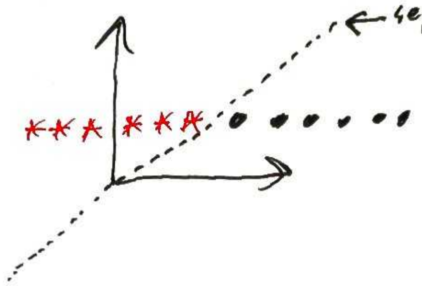
geometric interpretation: this extra dimension allows any linear separator to be represented by a higher dimensional separator that passes through the origin of the higher space.

1D data



← not separable through origin

becomes 2D data



← separates data, passes through new origin

← all data has value 1 in new dimension

Same thing applies in higher dimensions, but harder to draw.

Multiclass Prediction: Our k-nn and decision trees supported any number of labels, but our perceptrons only support two classes. However, we can try to reduce a multiclass problem into a set of multiple binary problems. Two common approaches are One-vs-Rest and One-vs-One.

One-vs-Rest Reduction (Also called One-vs-All)

With K possible labels, we train K separate perceptrons. Each one has a different task with two classes: either a point is a member of class i (the one) or it is not a member of class i (the rest). When making a prediction, we look at the predictions of our K classifiers. If only one classifier makes a strong claim about being in one class, we go with that.

Example: 3 classes, Apple, Banana, Orange we train 3 classifiers:

Apple vs. Not Apple, Banana vs Not Banana, Orange vs Not Orange

Each is a binary task which we can do. Then when predicting, we look at all three outputs.

So if we got back "Not Apple", "Banana", "Not Orange", then we should predict Banana.

One-vs-One Reduction

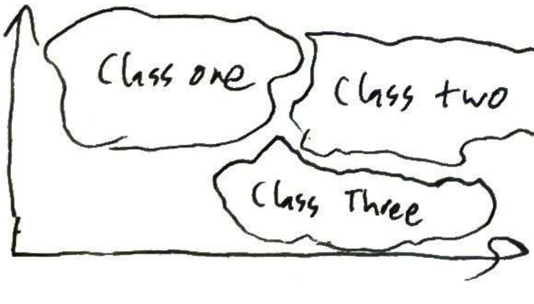
With K possible labels, we train $\frac{K(K-1)}{2}$ separate classifiers, one classifier for each pair of labels. Again, each subtask is binary, and we can train. To make a final prediction, we look for the class that "won" the most match-ups

Example 3 classes Apple, Banana, Orange

we train $\frac{3 \cdot 2}{2} = 3$ classifiers: Apple vs Banana, Apple vs Orange, Banana vs Orange

If we got back "Apple" [vs Banana], "Orange" [vs Apple], "Orange" [vs Banana], we should predict Orange

Exercise: Consider an application where our classes look like this in the space of our feature vectors:



Which approach, One-vs-Rest or One-vs-All do you think would work better here? Why?

Extension: Neural Nets: If you hook up the output of ~~two three~~ multiple perceptrons to be the inputs of another perceptron, you end up with a multi-layer perceptron. Change the function sign(...) to a different activation function, and apply a nicer learning algorithm to learn all the weights, and you end up with a modern neural net. Those are also out-of-scope for this course.