Tufts

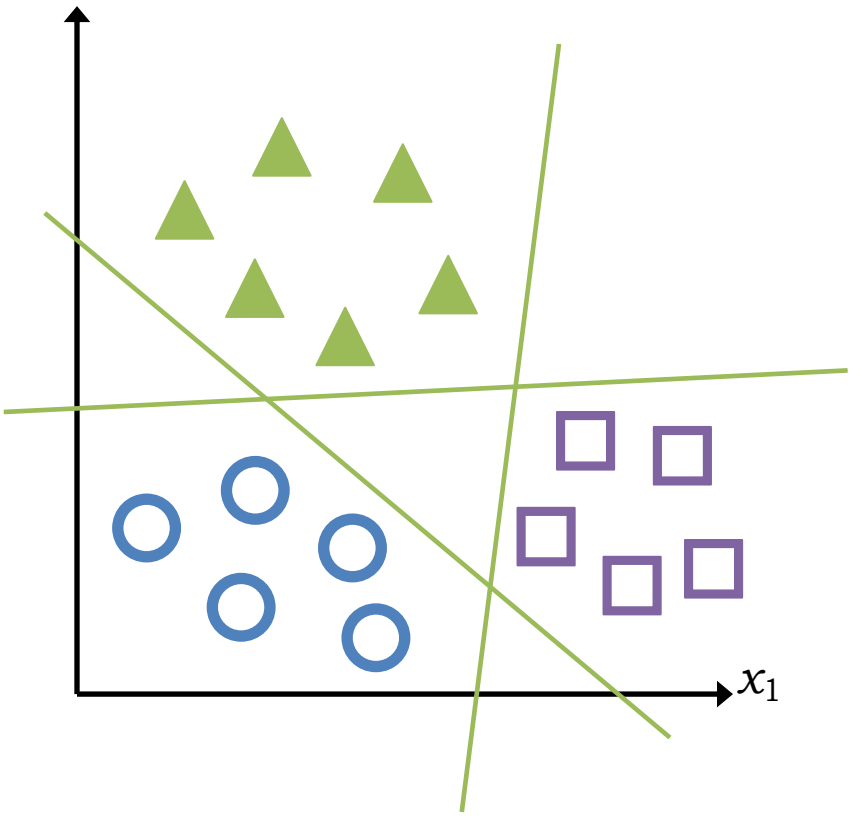
CS135
Introduction to Machine Learning

Lecture 9: Classifiers (Metrics, Logistic, Perceptron, and SVM)

Binary and Other Classification

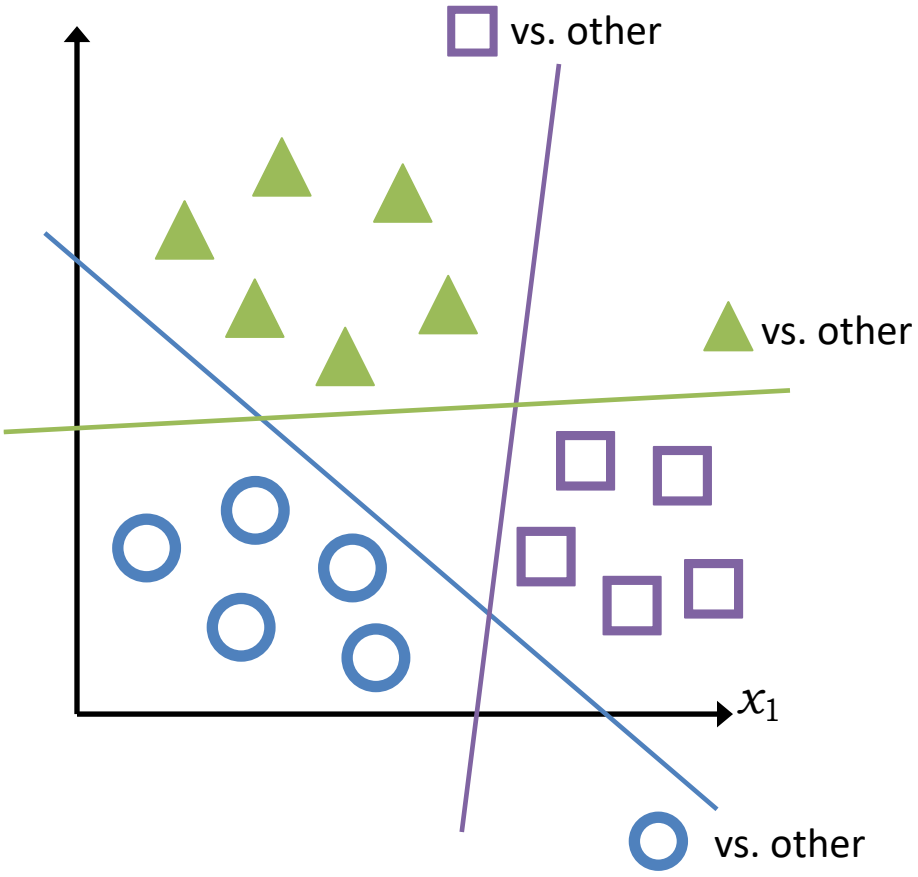
- We will generally discuss binary classifiers, which divide data into one of two classes.
- Linear classifiers are inherently binary, defining the classes based on two regions separated by a linear function.
  - However, many of the things we discuss can be applied to more than two classes.

Extending Binary Linear Classification



- In the presence of more than two classes, a single basic linear classifier can't properly divide data.
- Even if that data is linearly separable by class, any single line drawn must include elements of more than one class on at least one side
- We can combine ***multiple*** such classifiers, however…

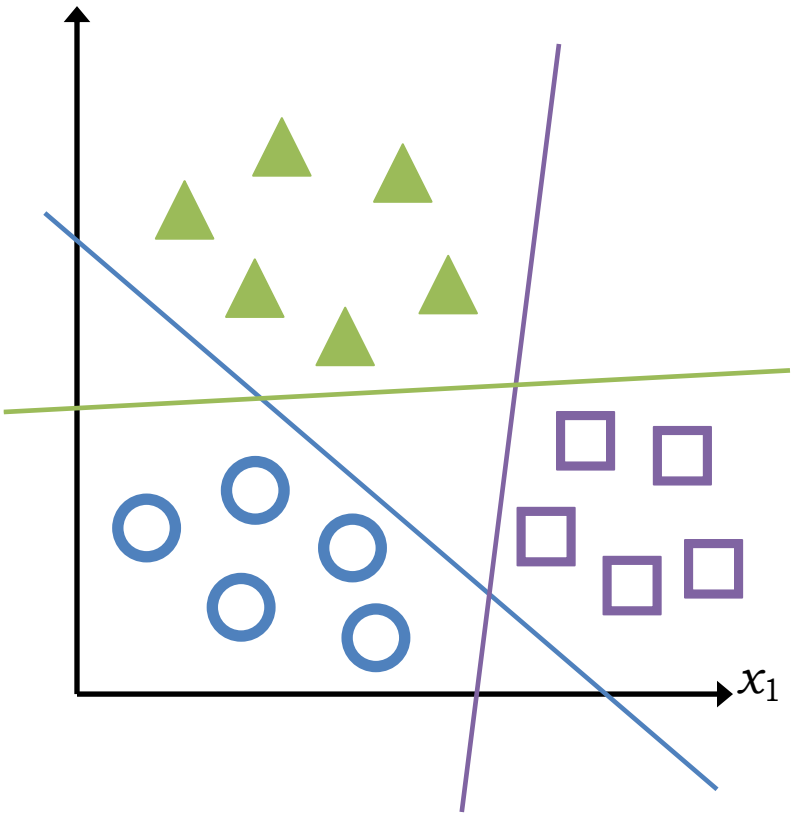
One-Versus-All Classification (OVA)



- In an OVA scheme, with k different classes:

1. Train k different 1/0 classifiers, one for each output class
2. On any new data item, apply each classifier to it, and assign it the class corresponding to the classifier for which it receives a 1

Issues with OVA Classification



- The basic OVA idea requires that each linear classifier separate one class from all others.
- As the number of classes increases, this added linear separability constraint gets harder to satisfy.

One-Versus-One Classification (OVO)

- Another idea is to train a separate classifier for each possible *pair* of output classes
  - Only requires each such pair to be ***individually*** separable, which is somewhat more reasonable
  - For k classes, it requires a larger number of classifiers:

$$\binom{k}{2} = \frac{k\,(k-1)}{2} = \mathrm{O}(k^2)$$

  - Relative to the size of data sets, this is generally manageable, and each classifier is often simpler than in an OVA setting

- A new data item is again tested against all the classifiers and given the class of the ***majority*** of those for which it is given a non-negative (1) value.
  - May still suffer from some ambiguities

Evaluating a Classifier

- It is often helpful to separate the results generated by a classifier according to what it gets right or not:
  - True Positives (TP): those that it identifies correctly as relevant
  - False Positives (FP): those that if identity wrongly as relevant
  - False Negatives (FN): those that are relevant but missed
  - True Negatives (TN): those it correctly labels as non-relevant
- These categories make sense when we are interested in separating one relevant class from another (again, we return to binary classification for simplicity)
- Of course, relevance depends upon what we care about:
  - Picking out the actual earthquakes in seismic data (earthquakes are relevant; explosions are not)
  - Picking out the explosions in seismic data (explosions are relevant; earthquakes are not)

Evaluating a Classifier

- It is often helpful to separate the results generated by a classifier according to what it gets right or not:
  - True Positives (TP): those that it identifies correctly as relevant
  - False Positives (FP): those that if identity wrongly as relevant
  - False Negatives (FN): those that are relevant but missed
  - True Negatives (TN): those it correctly labels as non-relevant

		Classifier Output	
		Positive (1)	Negative (0)
Ground Truth	Positive (1)	TP	FN
	Negative (0)	FP	TN

Basic Accuracy

- The simplest measure of accuracy is just the fraction of correct classifications:

$$\frac{\#\ \text{Correct}}{|\text{Data-set}|} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

- Basic accuracy treats both types of correctness—and therefore both types of error—as the same
- This isn't always what we want, however; sometimes false positives and false negatives are entirely different things

Basic Accuracy

- The most straightforward measure of accuracy can also be misleading, depending upon the data set itself:

$$\frac{\#\ \text{Correct}}{|\text{Data-set}|} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

- In a data set of 100 examples, with 99 positive and only a single negative example, any classifier that says positive (1) for everything would have 99% "accuracy."
- Such a classifier might only be useful for real-world classification problems, however!

Confusion Matrices

- One way to separate positive and negative examples and better analyze the behavior of a classifier is to break down the overall success/failure case by case
- For 100 data points, 50 of each type, we might have behavior as shown in the following table:

		Classifier Output	
		Positive (1)	Negative (0)
Ground Truth	Positive (1)	49	1
	Negative (0)	10	40

- What can this tell us?

Confusion Matrices

		Classifier Output	
		Positive (1)	Negative (0)
Ground Truth	Positive (1)	10	40
	Negative (0)	49	1

- In this data, the *overall* accuracy is $89/100 = 89\%$
- However, we see that the accuracy over the two types of data is quite different:

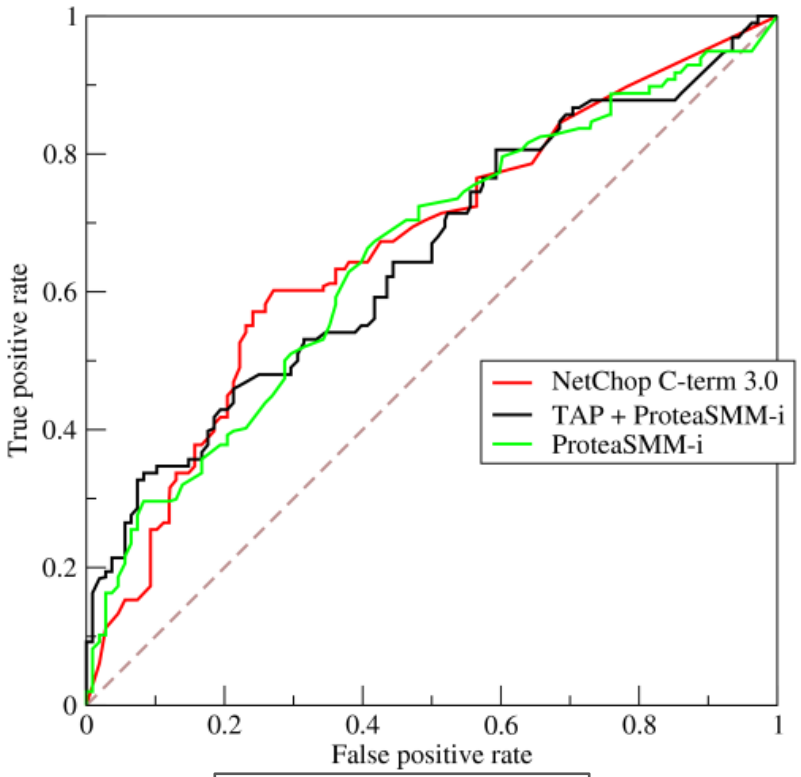
1. For negative data, accuracy is just $40/50 = 80\%$, with a 20% rate of false positives
2. For positive data, accuracy is $49/50 = 98\%$, with only a 2% rate of false negatives

Other Measures of Accuracy

- We can focus on a variety of metrics, depending on what we care about
  - "C = X" is "Classifier says X", & "T = Y" is "Truth is Y"

Metric	Formula	How often…	Probability
True Positive Rate (Recall)	$\frac{TP}{TP + FN}$	positive examples are correctly labeled	P(C = 1 | T = 1)
True Negative Rate (Specificity)	$\frac{TN}{TN + FP}$	negative examples are correctly labeled	P(C = 0 | T = 0)
Positive Predictive Value (Precision)	$\frac{TP}{TP + FP}$	examples labeled positive actually are positive	P(T = 1 | C = 1)
Negative Predictive Value	$\frac{TN}{TN + FN}$	examples labeled negative actually are negative	P(T = 0 | C = 0)

ROC Curves



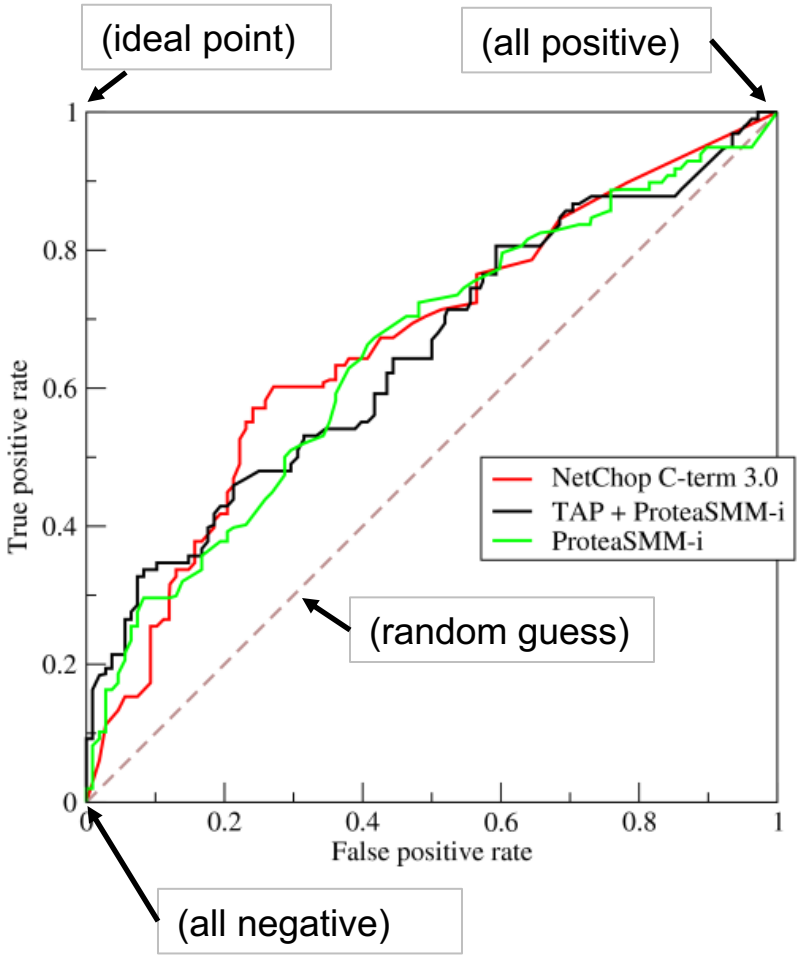
$$\text{TPR} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

$$\text{FPR} = \frac{\text{FP}}{\text{FP} + \text{TN}}$$

- Another way to look at classifier performance is the *ratio* of the rates of true positives and false ones
- That is, we compare the percentage of the true positives the classifier gives the right result for, and the percentage of errors it makes by mistakenly classifying negative examples as positive

Image source: BOR, Wikipedia (CC ASA 3.0 license)

ROC Curves



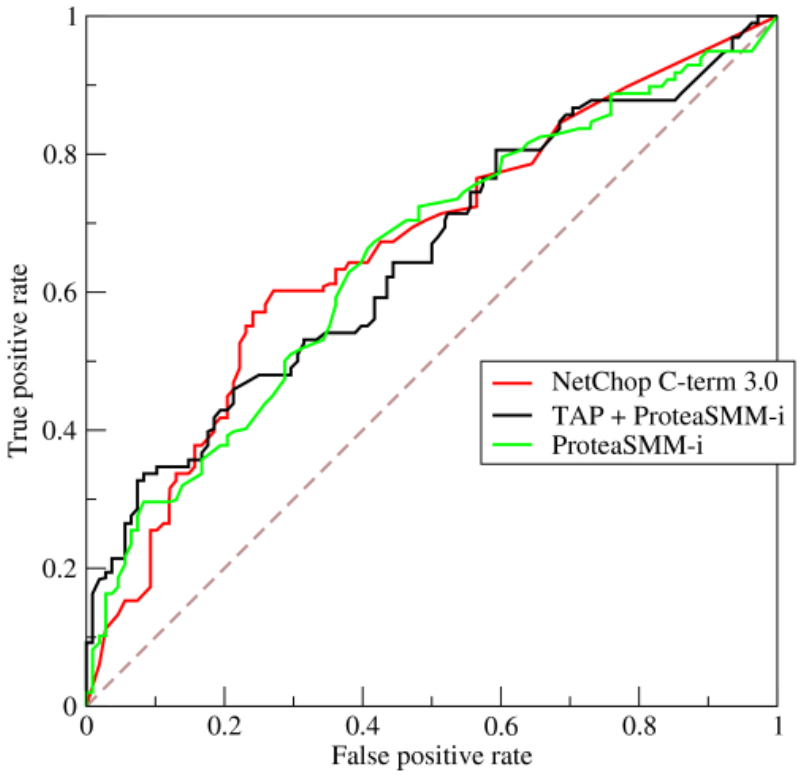
- Some obvious facts :

1. A ***perfect*** classifier would give us 100% success for true positives, with a 0% rate of false ones
2. A ***coin-flip*** classifier would achieve equal rates of each
3. Any classifier that is always positive hits all true ***and*** false positives
4. One that is always negative has ***no*** true or false positives

Image source: BOR, Wikipedia (CC ASA 3.0 license)

Tufts

Area Under ROC Curves (AUROC or AUC)



- The ROC curve can be very nuanced, and it is not always obvious from the curve itself how different algorithms measure up and compare
- A metric for comparing multiple curves is the area under them
  - A larger area means the curve gets a higher true positive success rate ***earlier*** (i.e., with fewer false positives) than one of smaller area

Image source: BOR, Wikipedia (CC ASA 3.0 license)

Probabilistic Classifiers

- The basic perceptron linear classifier assigns each data item to a specific class.
- Other approaches generate probability distributions over the data: they assign each data item a probability of being in a positive class.
  - A probability of 1.0 means the data item is *definitely* positive
  - A probability of 0.0 means the data item is *definitely* negative
  - A probability $0.0 < p < 1.0$ means the data item has *some chance* of being in *either* class
- **Question**: how can we turn the outputs of a probabilistic classifier *back into* a discrete $(1/0)$ classification?
  - One possibility is a threshold: pick a probability T such that everything assigned a probability $p \geq T$ is assigned positive, all else negative.

Log-Loss for Probabilistic Classification

- For any data item x_i (of N total), let y_i be the correct class label $(1/0)$, and let p_i be the probability assigned by the classifier that the data item is 1
- We can then define the logarithmic loss (log loss) for this classifier across the entire data set:

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N} y_i \log p_i + (1 - y_i) \log(1 - p_i)$$

- This measures cross entropy between the true distribution of labels in our data and the classifier's label distribution (that is, it measures the amount of extra noise introduced by the classifier relative to the true noisiness of the data set)

Log-Loss for Probabilistic Classification

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N} y_i \log p_i + (1 - y_i) \log(1 - p_i)$$

- If the true class of a data item is 1, then the log loss only sums up the *first* term in the right-hand equation.
  - The closer probability p_i is to 1 in this case. The closer loss is to 0.

- If the true class of a data-item is 0, then the log loss only sums up the *second* term in the right-hand equation.
  - The closer probability p_i is to 0 in this case. The closer loss is to 0.
  - Remember that by convention, we let $0 \log 0 = 0$

AUC for Probabilistic Classification

- If we are using a probabilistic classifier, then the area under the ROC curve for the classifier measures something else of real interest:

$$\text{AUC} = P(p_i > p_j \,|\, y_i = 1 \text{ AND } y_j = 0)$$

  - Here, again, let p_i is the probability assigned by the classifier that the data item is positive (1)

- This measures, for any given data items x_i and x_j, one positive and one negative, the chance that the classifier gives the positive one a higher probability than the negative one

A Problem Case for AUC

- Suppose we have data as shown, and two classifiers, C_1 and C_2 that assign probabilities as given in this table:

	y_1	C_1	C_2
x_1	0	0.10	0.15
x_2	0	0.20	0.25
x_3	0	0.30	0.35
x_4	0	0.45	0.50
x_5	1	0.60	0.55
x_6	1	0.75	0.65
x_7	1	0.80	0.70
x_8	1	0.95	0.85

- Although the classifiers differ in the values they assign to each data-point, they are ***both,*** in one sense, ***perfect***
  - There are threshold values for which each classifies every input correctly
  - In fact, for any threshold value ($0.50 < T \leq 0.55$), ***both*** will classify everything correctly

A Problem Case for AUC

- Varying threshold T does change the TPR and FPR of each classifier
  - However, each always has $\text{TPR} = 1.0$ or $\text{FPR} = 0.0$ (or both)
  - It is easy to verify that $\text{AUC} = 1.0$ (the same) for each classifier

	y_1	C_1	C_2
x_1	0	0.10	0.15
x_2	0	0.20	0.25
x_3	0	0.30	0.35
x_4	0	0.45	0.50
x_5	1	0.60	0.55
x_6	1	0.75	0.65
x_7	1	0.80	0.70
x_8	1	0.95	0.85

T	TPR_1	FPR_1	TPR_2	FPR_2
0.1	4/4	4/4	4/4	4/4
0.2	4/4	3/4	4/4	3/4
0.3	4/4	2/4	4/4	2/4
0.4	4/4	1/4	4/4	1/4
0.5	4/4	0/4	4/4	1/4
0.6	4/4	0/4	3/4	0/4
0.7	3/4	0/4	2/4	0/4
0.8	2/4	0/4	1/4	0/4
0.9	1/4	0/4	0/4	0/4
1.0	0/4	0/4	0/4	0/4

Choosing an Appropriate Measure

	y_1	C_1	C_2
x_1	0	0.10	0.15
x_2	0	0.20	0.25
x_3	0	0.30	0.35
x_4	0	0.45	0.50
x_5	1	0.60	0.55
x_6	1	0.75	0.65
x_7	1	0.80	0.70
x_8	1	0.95	0.85

- AUC is not a useful metric here, since it rates each classifier the same
- Instead, we can compare the log-loss, which is better (lower) for C_1 because it consistently outputs a probability that is ***closer*** to the correct value (i.e., higher for the 1's and lower for the 0's)

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N} y_i \log p_i + (1 - y_i)\log(1 - p_i)$$

$$\mathcal{L}(C_1) \approx 0.2945$$

$$\mathcal{L}(C_2) \approx 0.3902$$

Threshold Functions

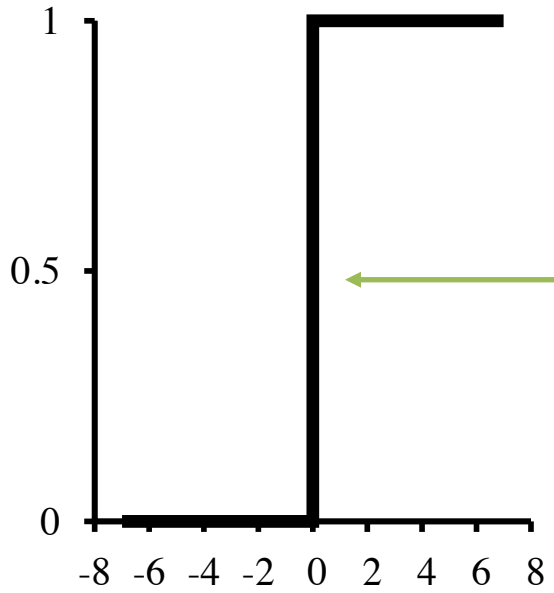
1. We have data-points with n features:
$$\mathbf{x} = (x_1, x_2, \ldots, x_n)$$
2. We have a linear function defined by $n+1$ weights:
$$\mathbf{w} = (w_0, w_1, w_2, \ldots, w_n)$$
3. We can write this linear function as:
$$\mathbf{w} \cdot \mathbf{x}$$
4. We can then find the linear boundary, where:
$$\mathbf{w} \cdot \mathbf{x} = 0$$
5. And use it to define our threshold between classes:
$$h_{\mathbf{w}} = \begin{cases} 1 & \mathbf{w} \cdot \mathbf{x} \geq 0 \\ 0 & \mathbf{w} \cdot \mathbf{x} < 0 \end{cases}$$

Outputs 1 and 0 here are ***arbitrary labels*** for one of two possible classes

Hard Thresholds are Hard!

$$h_{\mathbf{w}} = \begin{cases} 1 & \mathbf{w} \cdot \mathbf{x} \geq 0 \\ 0 & \mathbf{w} \cdot \mathbf{x} < 0 \end{cases}$$

- The hard threshold function used by the perceptron algorithm (among others) produces some conceptual and mathematical challenges
- Gives a yes/no answer everywhere, which can be tricky when our data isn't linearly separable

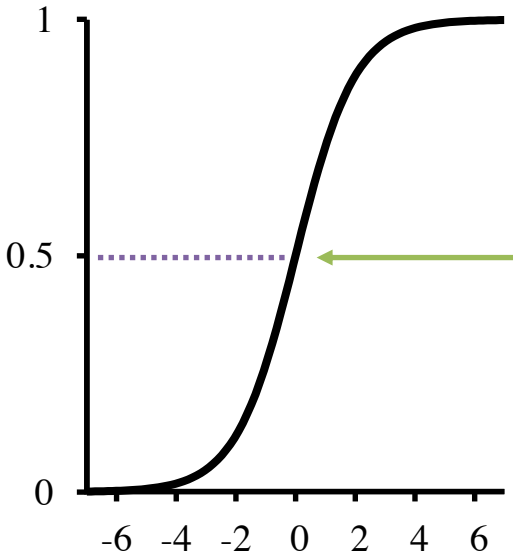


Function is discontinuous (non-differentiable) at $x = 0$

The Logistic Function

$$h_{\mathbf{w}}(\mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w} \cdot \mathbf{x}}}$$

- We can generate a smooth curve by instead using the logistic function as a threshold
- We can treat this value as a ***probability*** of belonging to one class or another

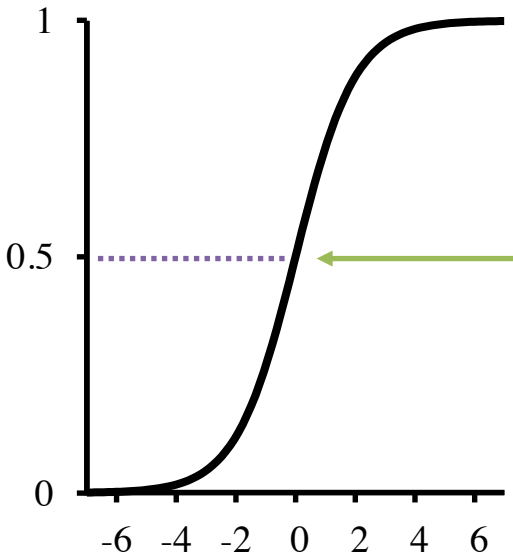


Probability function is 0.5 at $x = 0$

Using the Logistic for Classification

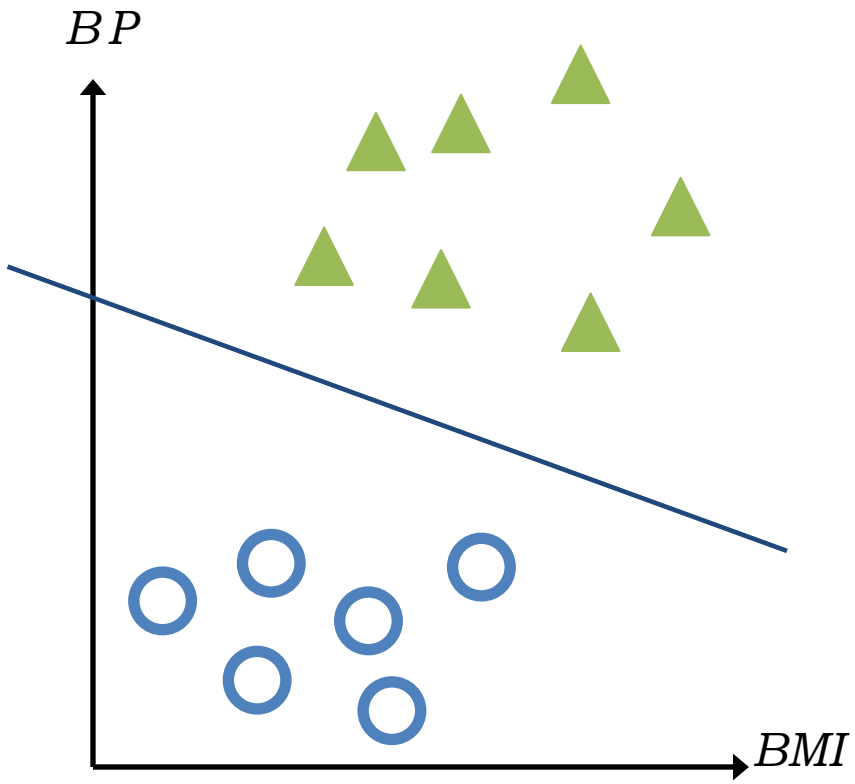
$$h_{\mathbf{w}}(\mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w} \cdot \mathbf{x}}}$$

- Treated as a probability, the logistic can still be used to *classify* data, where the class is the one that has highest probability overall, while also supplying a probability for that outcome



A "coin flip" where we have $x = 0$

Issues with Linear Classification



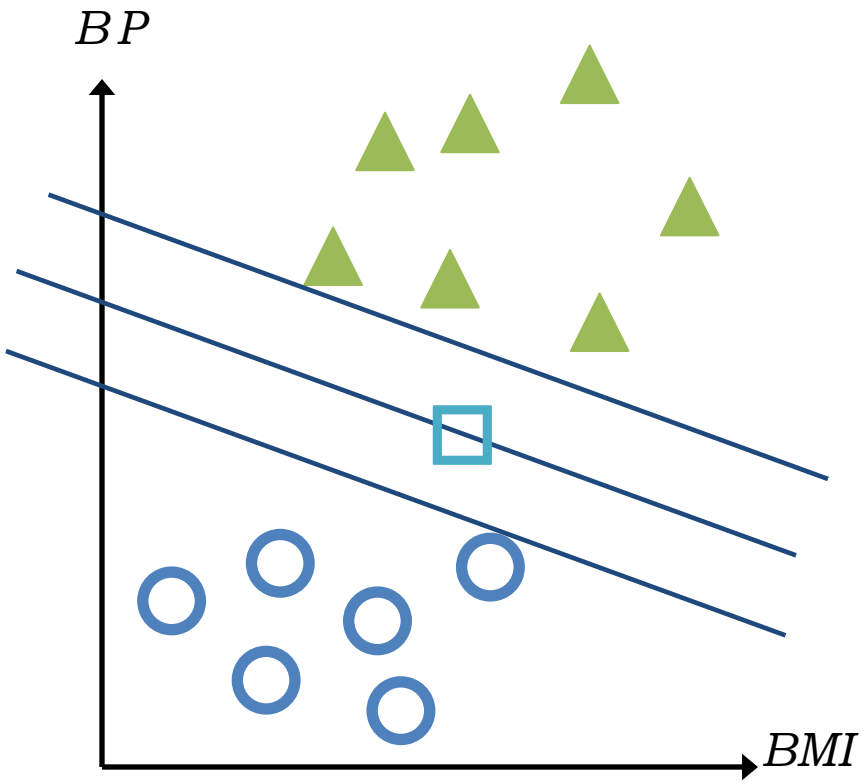
- Consider data about heart-attack risk, based on body mass index (BMI) and blood pressure (BP)
- Even assuming linearly separable training data, linear classification gives a hard cut-off that may not be appropriate

heart attack

don't know?

no heart attack

Issues with Linear Classification



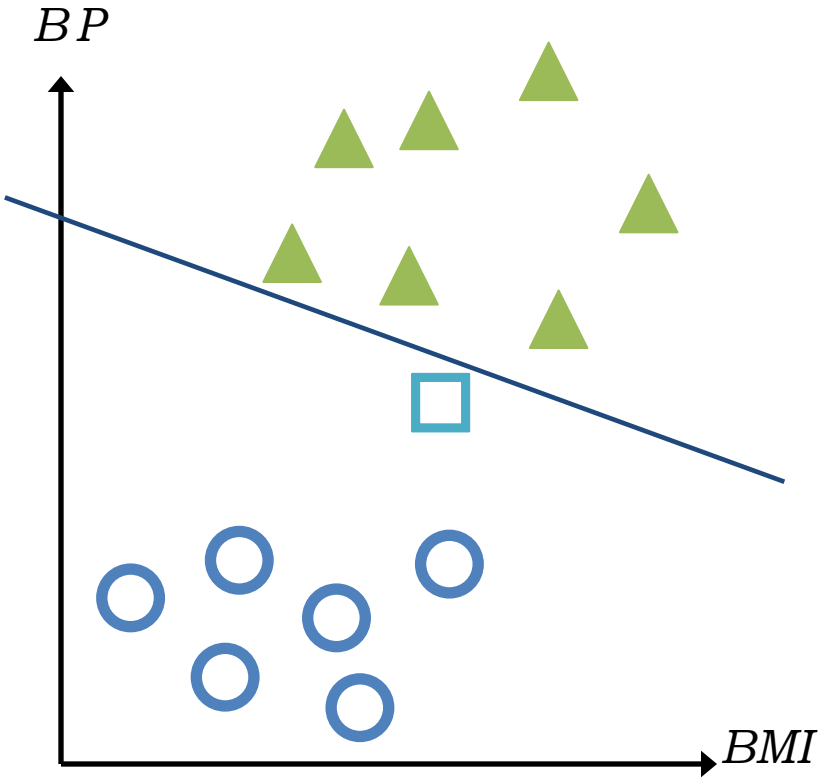
- Given that ***multiple*** possible lines can separate this data, how do we classify a new instance in the region between the training instances?

heart attack

no heart attack

don't know?

Issues with Linear Classification



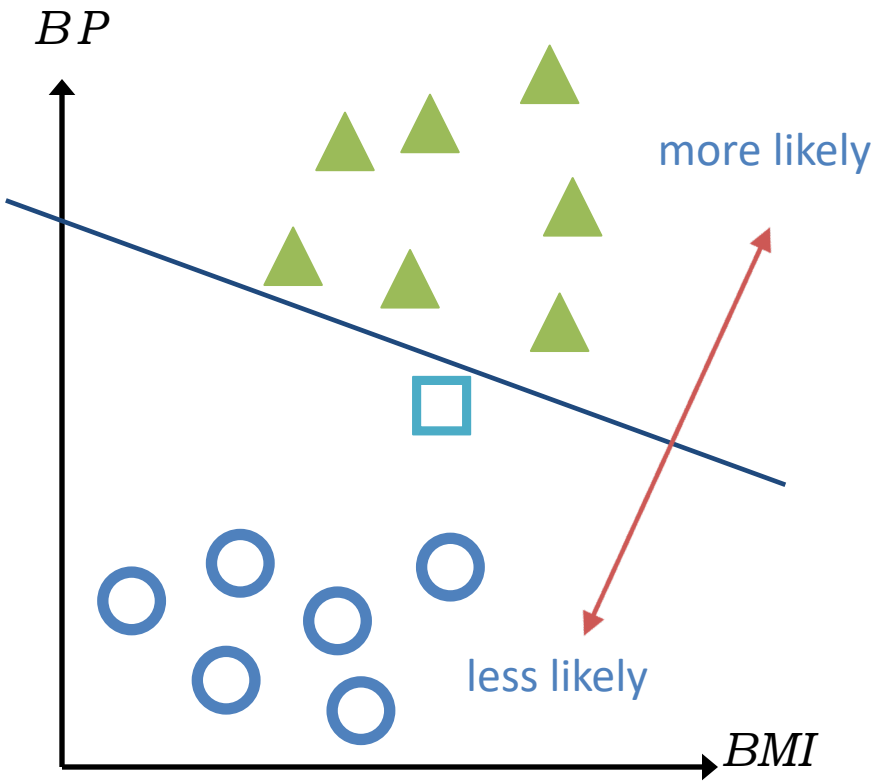
- Even if we did settle on some fixed line, what do we do with something that is *very close* to the separator?

heart attack

don't know?

no heart attack

Using Probabilistic Classification



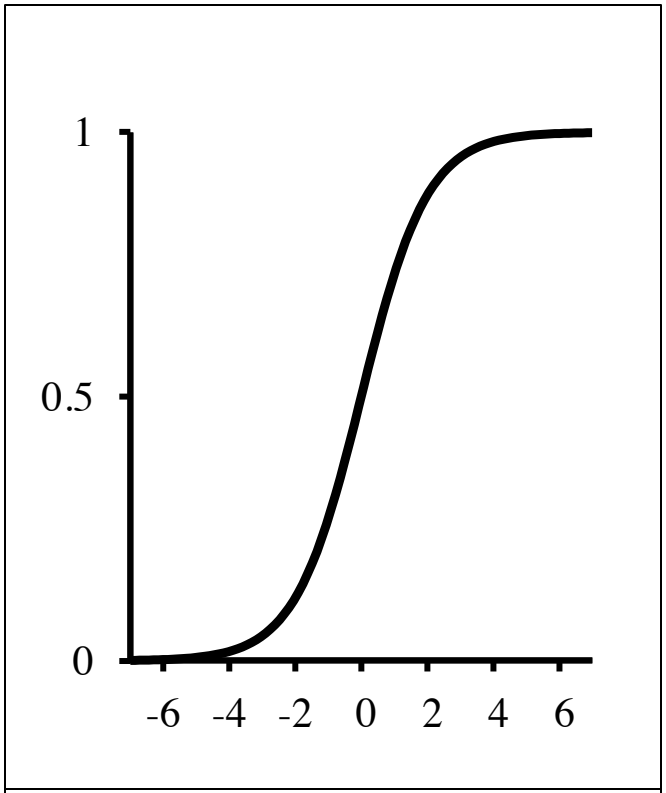
- Logistic regression also generates a linear separator (where the weight-function = 0), but now it is giving us an empirical distribution over the data-set
- A new data point close to the line still has ***some positive probability*** of being in the class on the other side of it

heart attack

don't know?

no heart attack

Properties of the Logistic Function



$$h_{\mathbf{w}}(\mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w} \cdot \mathbf{x}}}$$

- Also known as the Sigmoid, from the shape of its plot
- It always has a value in range:

$$0 \leq x \leq 1$$

- The function is everywhere differentiable, and has a *derivative* that is easy to calculate, which turns out to be useful for learning:

$$h'_{\mathbf{w}}(\mathbf{x}) = h_{\mathbf{w}}(\mathbf{x})(1 - h_{\mathbf{w}}(\mathbf{x}))$$

Logistic Regression

- In perceptron learning we update the weight vector in each case based upon a mis-classified instance, using the equation:

$$w_j \leftarrow w_j + \alpha(y_i - h_{\mathbf{w}}(\mathbf{x}_i)) \times x_{i,j}$$

- For the logistic, using the same loss function (squared error), we would do the same, but add an extra term:

$$w_j \leftarrow w_j + \alpha(y_i - h_{\mathbf{w}}(\mathbf{x}_i)) \times h_{\mathbf{w}}(\mathbf{x}_i)(1 - h_{\mathbf{w}}(\mathbf{x}_i)) \times x_{i,j}$$

The difference between what output ***should*** be, and what our weights make it

The derivative of the logistic

The jth feature-value

Gradient Descent for Logistic Regression, I

- We could then use the same approach as for linear classification, starting with some random (or uniform) weights and then:

1. Choose an input $\mathbf{x}_i$ from our data set that is wrongly classified.
2. Update vector of weights, $\mathbf{w} = (w_0, w_1, w_2, \ldots, w_n)$:

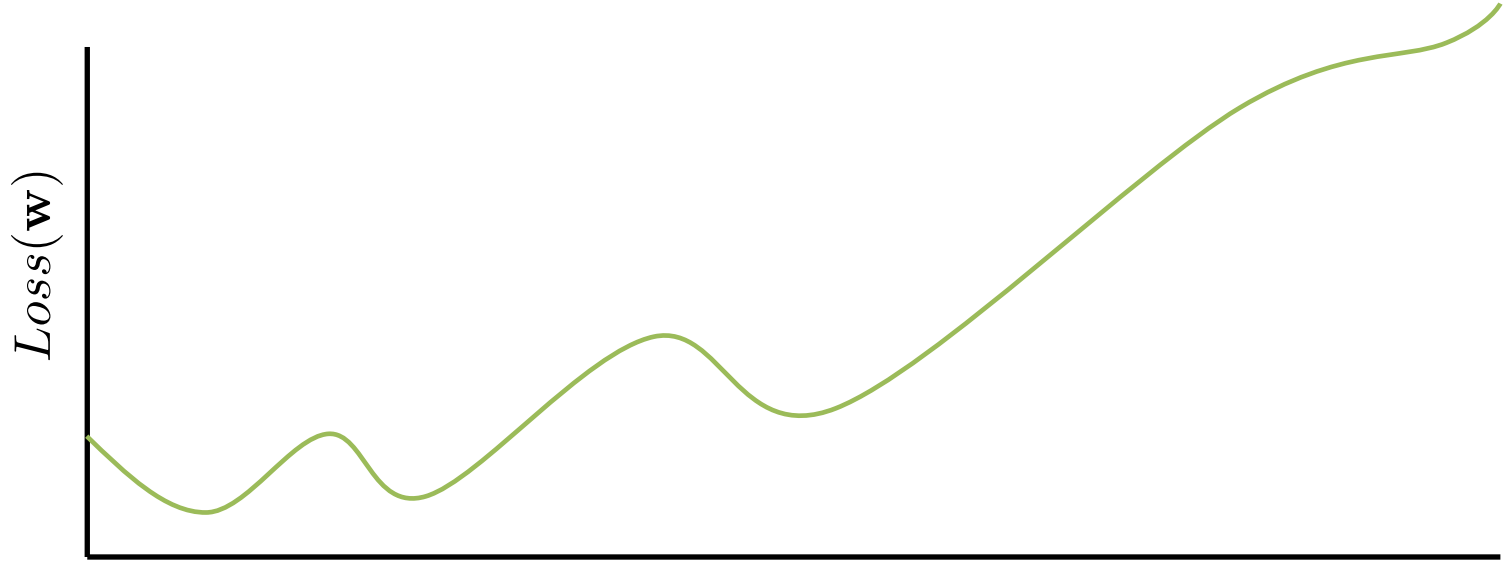
$$w_j \leftarrow w_j + \alpha(y_i - h_{\mathbf{w}}(\mathbf{x}_i)) \times h_{\mathbf{w}}(\mathbf{x}_i)(1 - h_{\mathbf{w}}(\mathbf{x}_i)) \times x_{i,j}$$

3. Repeat until weights no longer change; modify learning parameter α over time to guarantee this.

- Again, we make α smaller and smaller over time, and the algorithm converges as $\alpha \to 0$

A Problem: Local Minima

$$w_j \leftarrow w_j + \alpha(y_i - h_{\mathbf{w}}(\mathbf{x}_i)) \times h_{\mathbf{w}}(\mathbf{x}_i)(1 - h_{\mathbf{w}}(\mathbf{x}_i)) \times x_{i,j}$$



- While this sort of weight update ***does*** drive the error down, it has a flaw: squared-error loss for logistic regression is ***not*** convex
  - Gradient descent can get stuck in locally optimal solutions that aren't ideal
  - Solution: change the error function!

Gradient Descent for Logistic Regression, II

- We change the weight-update in our gradient descent approach:

1. Choose an input $\mathbf{x}_i$ from our data set that is wrongly classified.
2. Update vector of weights, $\mathbf{w} = (w_0, w_1, w_2, \ldots, w_n)$:

$$w_j \leftarrow w_j + \alpha\, x_{i,j}\, (y_i - h_{\mathbf{w}}(\mathbf{x}_i))$$

3. Repeat until weights no longer change; modify learning parameter α over time to guarantee this.

- Note: the update ***looks*** like the update for linear regression, but:
  1. It only uses a ***single*** incorrect data point, not the sum of ***all*** errors
  2. The hypothesis *h* is an application of the logistic function
- We can do this update because we ***don't*** use the squared-error loss but use a ***different*** loss function: logistic loss.

Gradient Descent for Logistic Regression

$$w_j \leftarrow w_j + \alpha\, x_{i,j}\,(y_i - h_{\mathbf{w}}(\mathbf{x}_i))$$

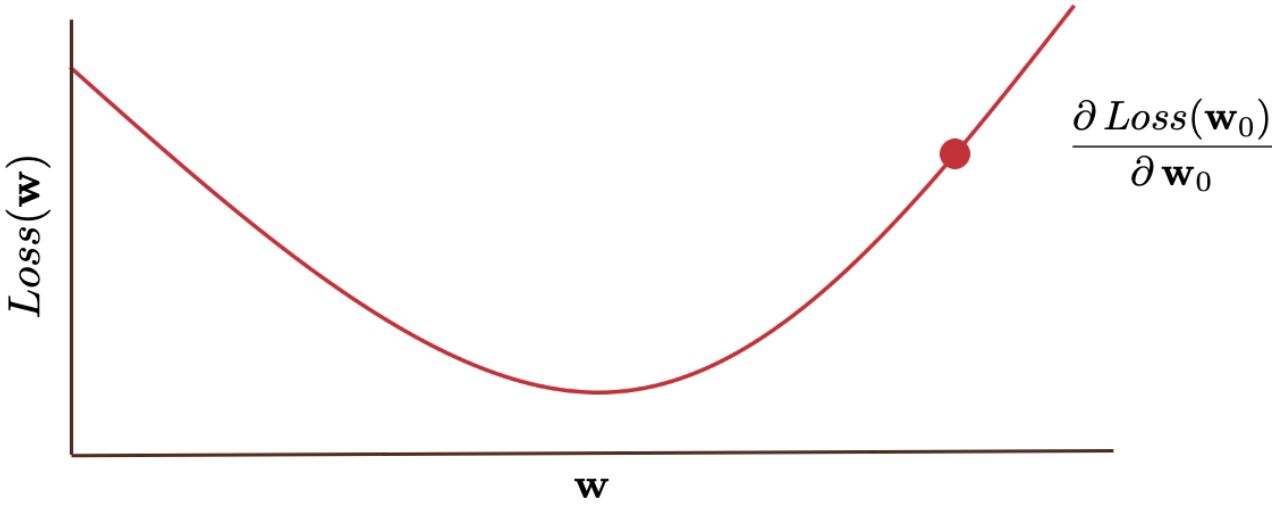
- The logistic update equation, via gradient descent, minimizes the log-loss (also known as logistic loss or binary cross entropy):

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N} y_i \log p_i + (1 - y_i)\log(1 - p_i)$$

  - For these purposes, we treat the output of the logistic as the probability we are interested in: $p_i \triangleq h_{\mathbf{w}}(\mathbf{x}_i)$

  - Over time, we drive the loss towards 0

Gradient Descent for Logistic Regression



$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N} y_i \log p_i + (1 - y_i)\log(1 - p_i)$$

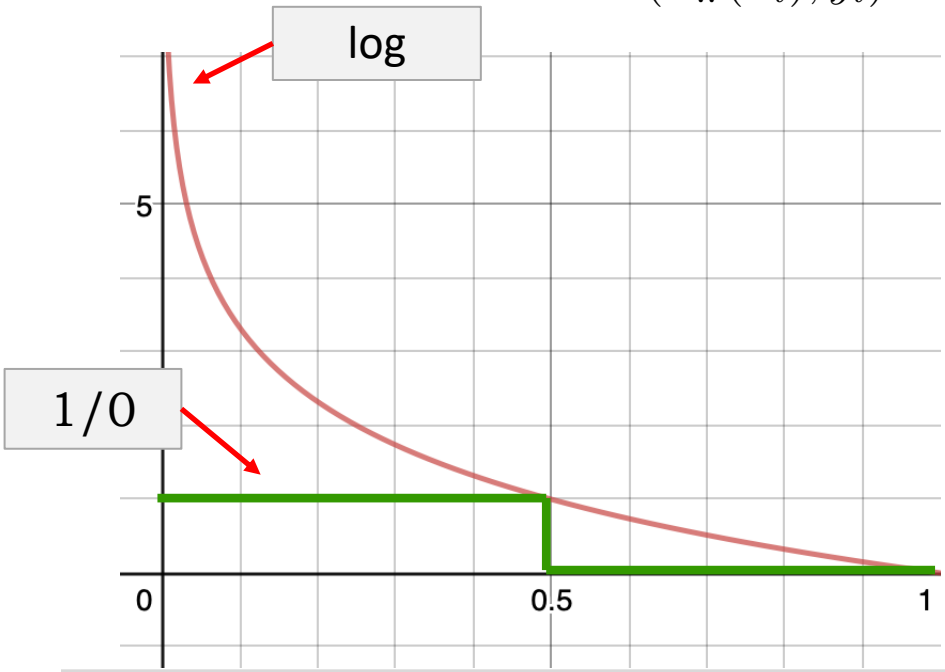
- The log-loss is a convex function
  - Like the losses used in linear regression and the perceptron
- This means that the gradient descent process (with suitable values of α) will converge upon a near-optimal solution

Logarithmic Loss vs. Threshold Error

- For an individual data element, the log loss is an upper bound on a threshold-based (1/0) loss:

$$\mathcal{E}(h_{\mathbf{w}}(\mathbf{x}_i), y_i) = \begin{cases} 0 & \text{if } h_{\mathbf{w}}(\mathbf{x}_i) = y_i \\ 1 & \text{if } h_{\mathbf{w}}(\mathbf{x}_i) \neq y_i \end{cases}$$

$$\mathcal{L}(h_{\mathbf{w}}(\mathbf{x}_i), y_i) = -[y_i \log h_{\mathbf{w}}(\mathbf{x}_i) + (1 - y_i) \log(1 - h_{\mathbf{w}}(\mathbf{x}_i))]$$



- This graph assumes:

1. True label is 1
2. Threshold used is 0.5 (i.e., $h_{\mathbf{w}}$ = 1 if probability assigned is $p \geq 0.5$)
3. Log base 2 is used

Linear vs. Logistic Regression

Linear Regression	Logistic Regression
A value $x \in \mathbb{R}$	A value $0 \leq x \leq 1$
Output value of an arbitrary function	Probability of belonging to a certain class
Tries to find line that best ***fits*** to the data	Tries to find separator that best ***divides*** the classes

Linear vs. Logistic Regression in Mathematical Terms

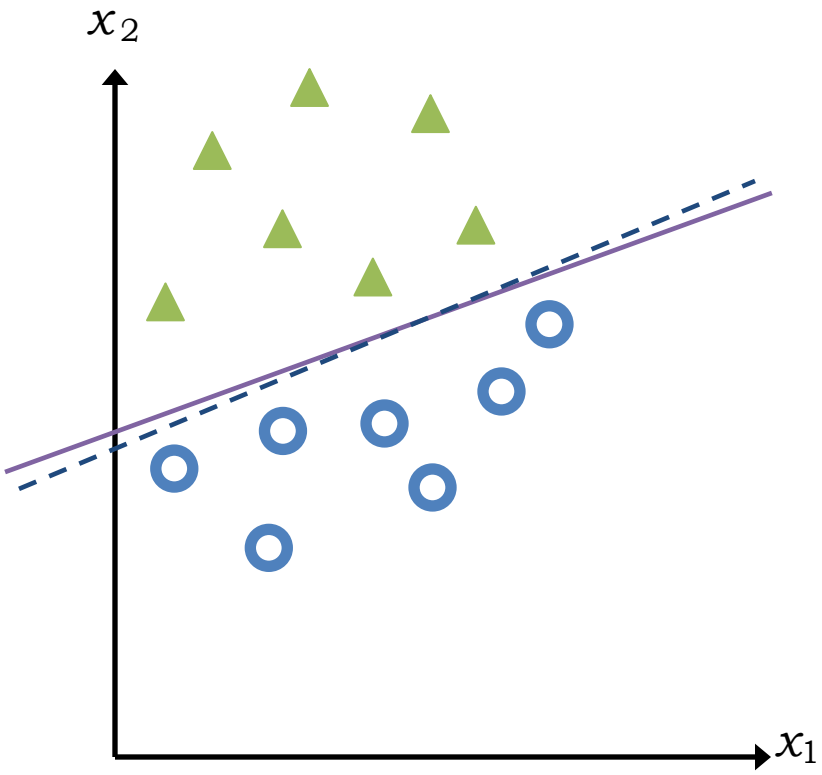
Linear	
Loss function	$Loss(\mathbf{w}) = \sum_{j=1}^{N} (y_j - h_{\mathbf{w}}(\mathbf{x}_j))^2$
Weight-update equation	$w_i \leftarrow w_i + \alpha \sum_j x_{j,i} \, (y_j - h_{\mathbf{w}}(\mathbf{x}_j))$

Logistic	
Loss function	$-\frac{1}{N} \sum_{j=1}^{N} [y_j \log h_{\mathbf{w}}(\mathbf{x}_j) + (1 - y_j) \log(1 - h_{\mathbf{w}}(\mathbf{x}_j))]$
Weight-update equation	$w_i \leftarrow w_i + \alpha \, x_{j,i} \, (y_j - h_{\mathbf{w}}(\mathbf{x}_j))$

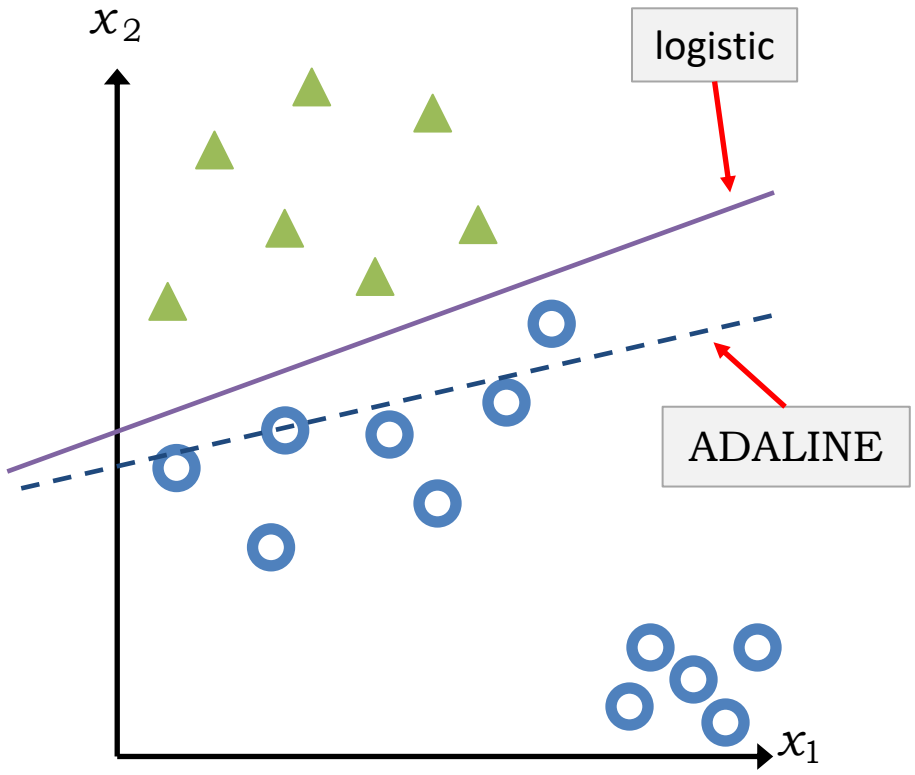
Another Approach: ADALINE classifiers

- Rather than a perceptron or logistic approach, what if we tried to use linear regression itself to build a classifier?
- For two classes, we could:
  1. Label data using two class-labels, $y \in \{+1, -1\}$
  2. Fit a linear regression to this data using squared loss (now measured as the difference between the linear value and the class-label, not some other real number) and the same weight-updates as before
  3. Classify data based upon whether the resulting linear function is ≥ 0 (in which case it is assigned $+1$) or not (-1)
- This is known as a least-squares or ADALINE (Adaptive Linear Neuron) classifier

Treatment of Outliers in Data



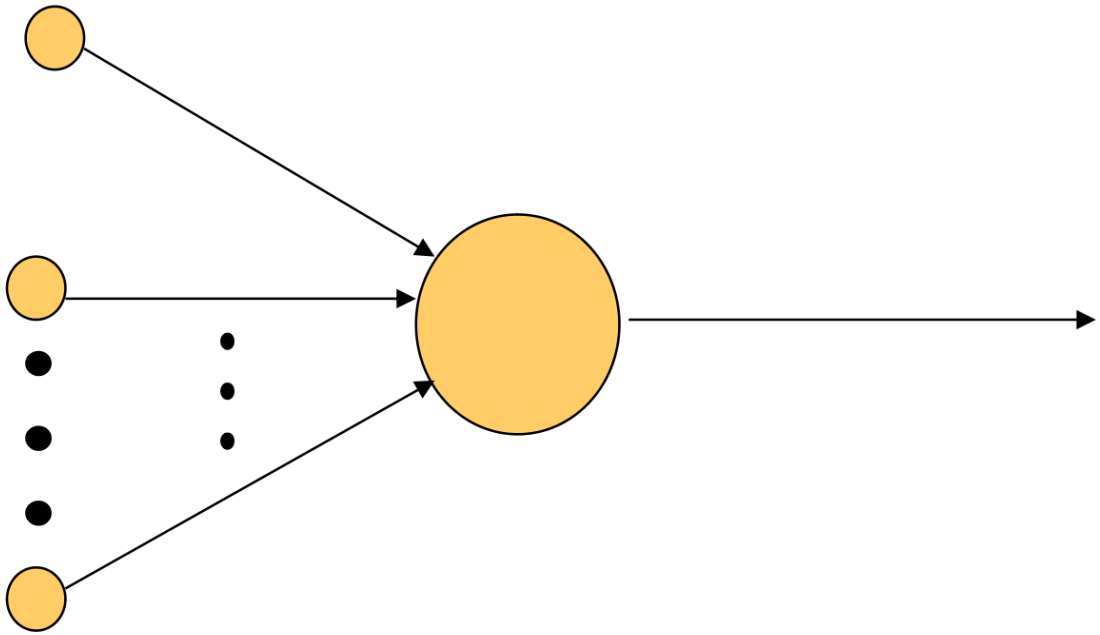
- Logistic regression (solid line) and ADALINE (dashed) give similar results on ***some*** data



- The ADALINE is skewed by outliers, however, as loss function sees them as "too correct"

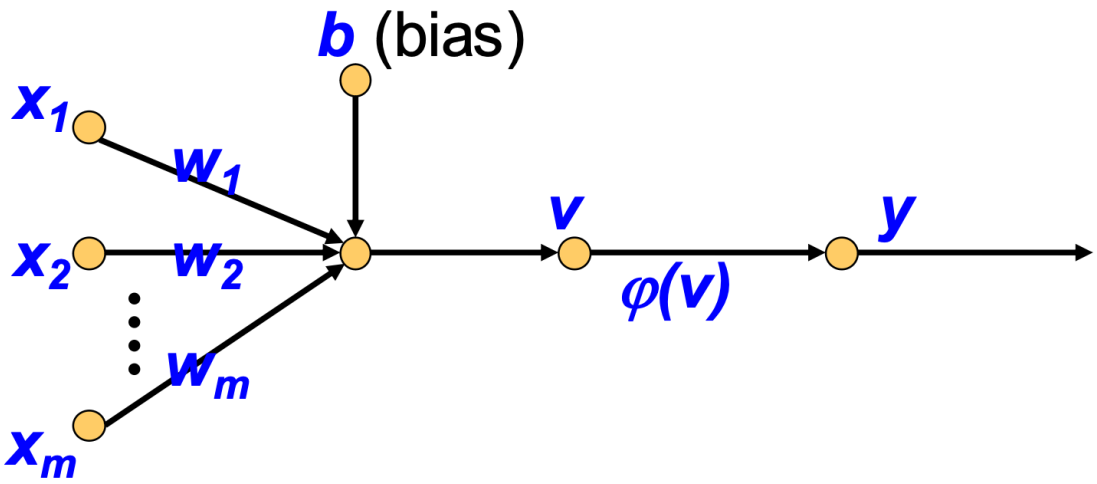
Architecture

- We consider the architecture: feed-forward NN with one layer
- It is sufficient to study single-layer preceptrons with just one neuron:



Perceptron: Neuron Model

- Uses a non-linear (McCulloch-Pitts) model of neuron:



- φ is the *sign* function:

$$\varphi(v) = \begin{cases} +1 & \text{IF } v >= 0 \\ -1 & \text{IF } v < 0 \end{cases}$$

Is the function *sign*(v)

Perceptron: Applications

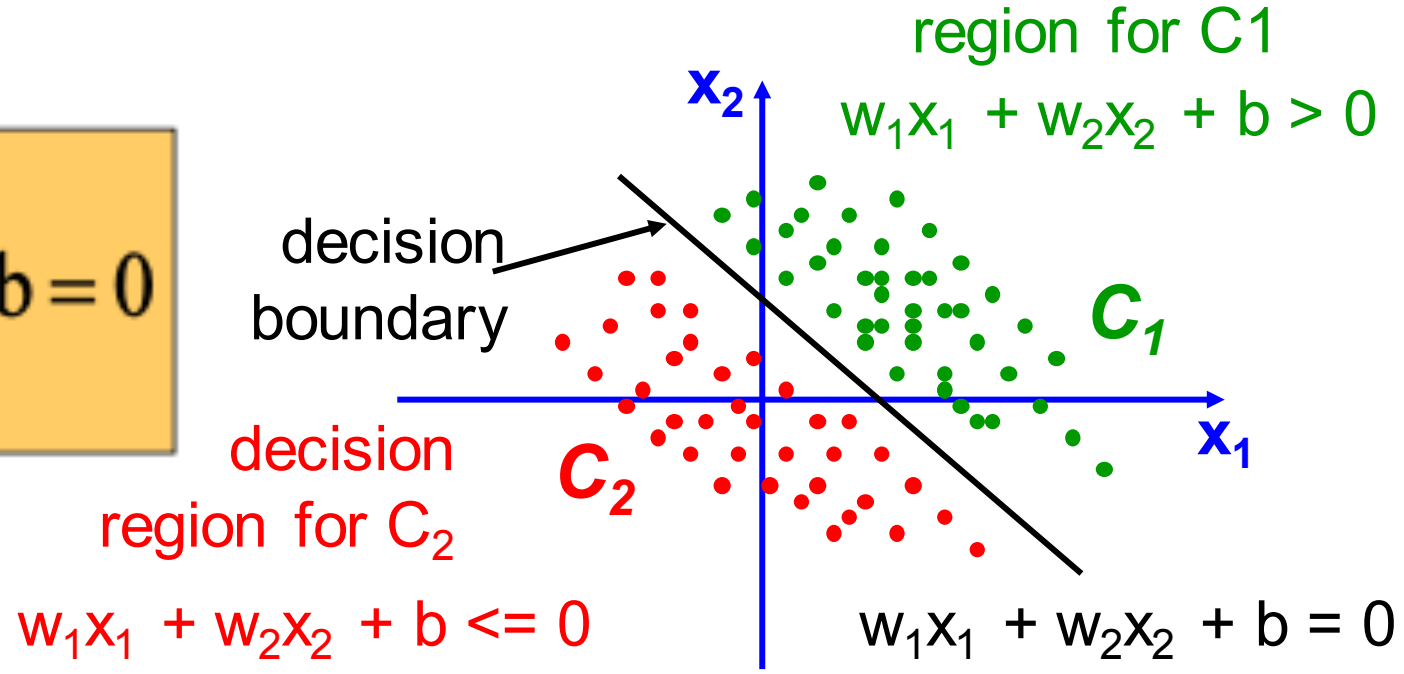
- The perceptron is used for classification:
- classify a set of examples correctly into one of the two classes C_1 and C_2:

If the output of the perceptron is +1, then the input is assigned to class C_1
If the output is -1, then the input is assigned to C_2

Perceptron: Classification

• The equation below describes a hyperplane in the input space. This hyperplane is used to separate the two classes C1 and C2

$$\sum_{i=1}^{m} w_i x_i + b = 0$$



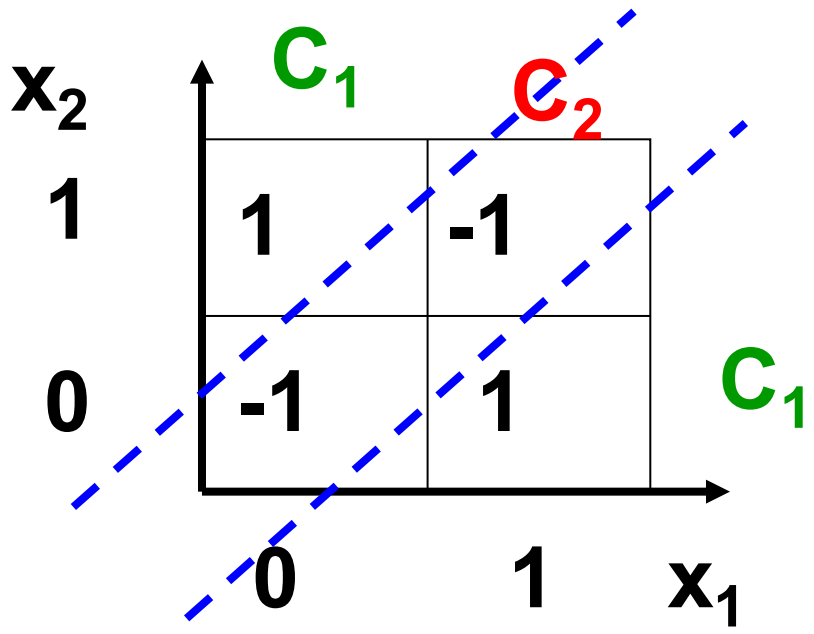
Perceptron: Limitations

- The perceptron can only model linearly separable functions.
- The perceptron can be used to model the following Boolean functions:
- AND
- OR
- COMPLEMENT
- But it cannot model the XOR. Why?

Perceptron: Limitations

The XOR is not linearly separable

- It is impossible to separate the classes C_1 and C_2 with only one line



Perceptron: Learning Algorithm

- **Variables and parameters**

$\mathbf{x}(n)$ = input vector

$= [+1, x_1(n), x_2(n), \ldots, x_m(n)]^T$

$\mathbf{w}(n)$ = weight vector

$= [b(n), w_1(n), w_2(n), \ldots, w_m(n)]^T$

$b(n)$ = bias

$y(n)$ = actual response

$d(n)$ = desired response

η = learning rate parameter

The fixed-increment learning algorithm

- Initialization: set $\mathbf{w}(0) = 0$
- Activation: activate perceptron by applying input example (vector $\mathbf{x}(n)$ and desired response $d(n)$)
- Compute actual response of perceptron:

$$y(n) = \text{sgn}[\mathbf{w}^T(n)\mathbf{x}(n)]$$

- Adapt weight vector: if $d(n)$ and $y(n)$ are different then

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \eta[d(n)-y(n)]\mathbf{x}(n)$$

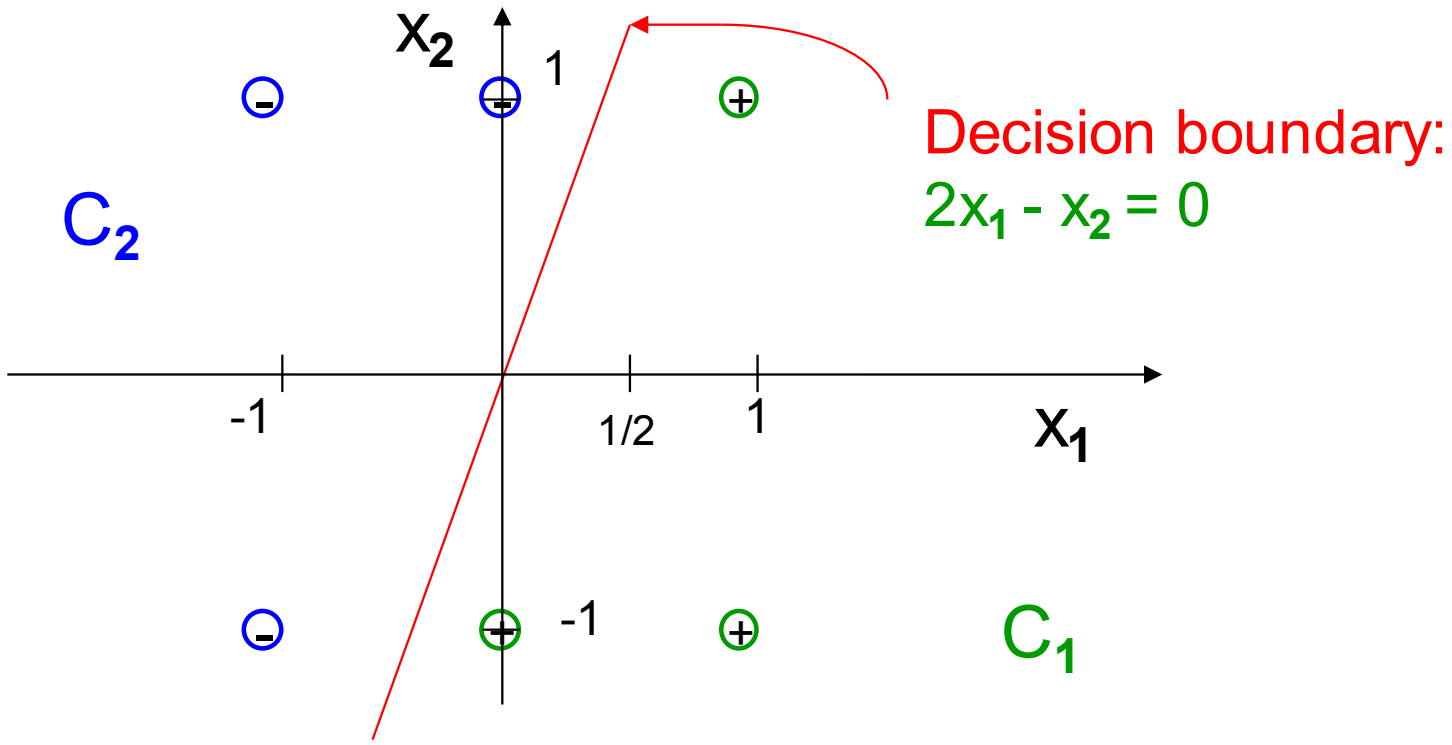
$$\text{Where } d(n) = \begin{cases} +1 & \text{if } \mathbf{x}(n) \in C_1 \\ -1 & \text{if } \mathbf{x}(n) \in C_2 \end{cases}$$

- Continuation: increment time step n by 1 and go to Activation step

Example

Consider a training set C**1** ∪ C**2**, where:
C**1** = {(1,1), (1, -1), (0, -1)} elements of class 1 C**2** = {(-1,-1), (-1,1), (0,1)} elements of class -1
Use the perceptron learning algorithm to classify these examples.
• **w**(0) = [1, 0, 0]**T** = 1

Example



Convergence of the learning algorithm

Suppose datasets C_1 and C_2 are linearly separable. The perceptron convergence algorithm converges after n_0 iterations, with $n_0 \leq n_{max}$ on the training set $C_1 \cap C_2$.

Proof:

- suppose $\boldsymbol{x} \in C_1 \rightarrow \text{output} = 1$ and $\boldsymbol{x} \in C_2 \rightarrow \text{output} = -1$.
- For simplicity assume **w**(1) = 0, η = 1.
- Suppose perceptron incorrectly classifies **x**(1) ... **x**(n) ... $\in C_1$.

Then $w^T(k)x(k) \leq 0$.

→ Error correction rule:

$$\left.\begin{aligned} \mathbf{w}(2) &= \mathbf{w}(1) + \mathbf{x}(1) \\ \mathbf{w}(3) &= \mathbf{w}(2) + \mathbf{x}(2) \\ &\vdots \\ \mathbf{w}(n+1) &= \mathbf{w}(n) + \mathbf{x}(n). \end{aligned}\right\} \Rightarrow \mathbf{w}(n+1) = \mathbf{x}(1) + \ldots + \mathbf{x}(n)$$

Adaline: Adaptive Linear Element

- Adaline: uses a linear neuron model and the Least-Mean-Square (LMS) learning algorithm

  The idea: try to minimize the square error, which is a function of the weights

$$E(\mathrm{w}(\mathrm{n})) = \frac{1}{2}\mathrm{e}^2(\mathrm{n})$$

$$\mathrm{e}(\mathrm{n}) = \mathrm{d}(\mathrm{n}) - \sum_{\mathrm{j}=0}^{\mathrm{m}} \mathrm{x}_{\mathrm{j}}(\mathrm{n})\mathrm{w}_{\mathrm{j}}(\mathrm{n})$$

- We can find the minimum of the error function *E* by means of the Steepest descent method

Steepest Descent Method

- start with an arbitrary point
- find a direction in which E is decreasing most rapidly

$$-(\text{gradient of } E(\mathrm{w})) = -\left[\frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_m}\right]$$

- make a small step in that direction

$$\mathrm{w}(\mathrm{n}+1) = \mathrm{w}(\mathrm{n}) - \eta(\text{gradient of } \mathrm{E}(\mathrm{n}))$$

Least Mean Square Algorithm (Widrow-Hoff Algorithm)

- Approximation of gradient(E)

$$\frac{\partial E(\mathrm{w(n)})}{\partial \mathrm{w(n)}} = \mathrm{e(n)} \frac{\partial \mathrm{e(n)}}{\partial \mathrm{w(n)}}$$
$$= \mathrm{e(n)}[-\mathrm{x(n)}^{\mathrm{T}}]$$

- Update rule for the weights becomes:

$$\mathrm{w(n+1)} = \mathrm{w(n)} + \eta \mathrm{x(n)e(n)}$$

Summary of LMS Algorithm

Training sample: input signal vector **x(n)**

desired response **d(n)**

User selected parameter $\eta > 0$

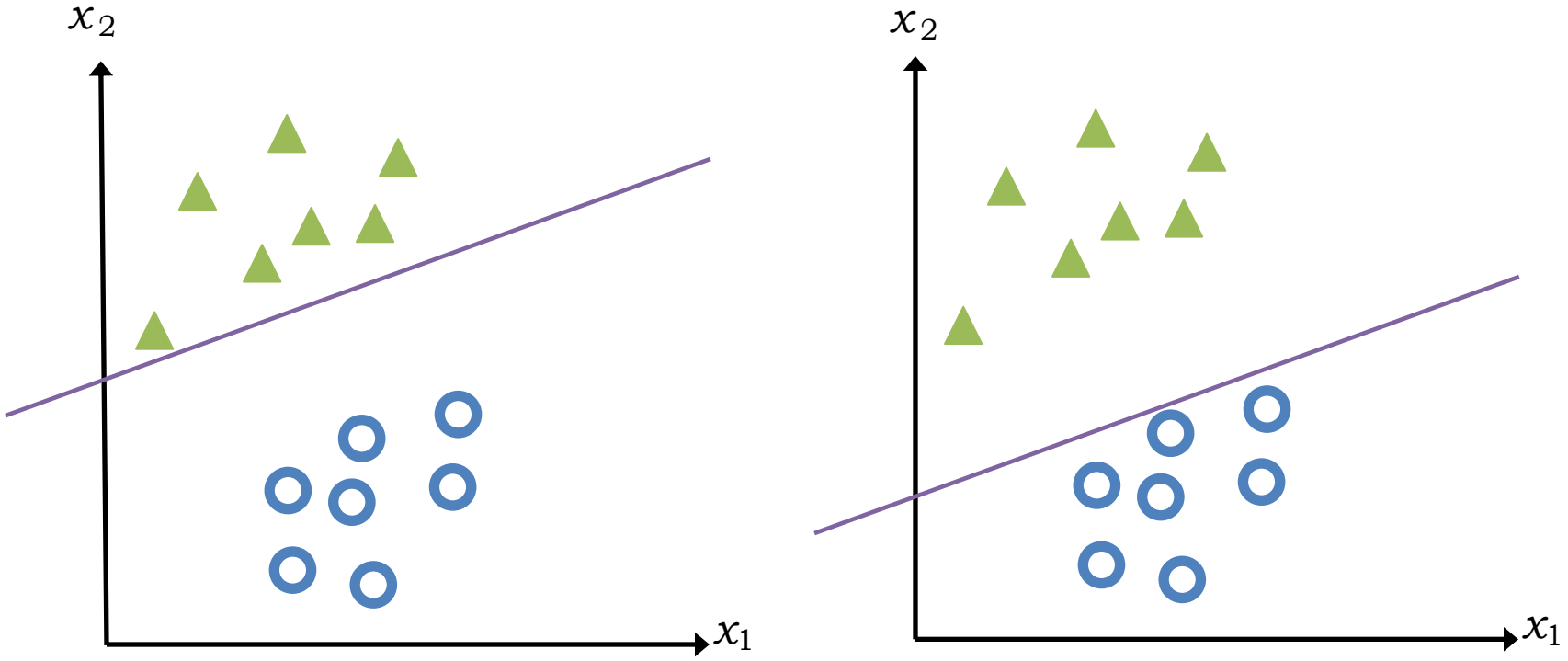
Initialization **set $\hat{w}(1) = 0$**

Computation **for n = 1, 2, … compute**

$$e(n) = d(n) - \hat{w}^T(n)x(n)$$

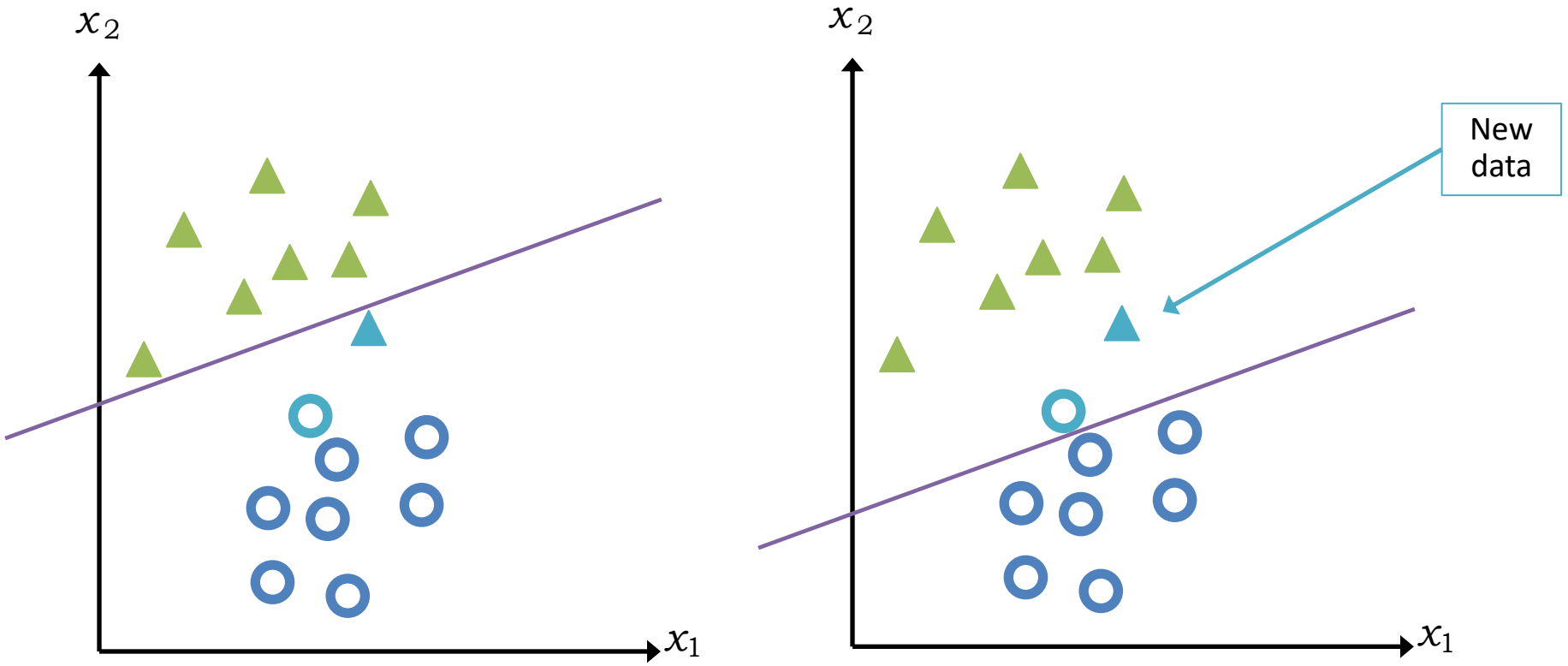
$$\hat{w}(n+1) = \hat{w}(n) + \eta\, x(n)e(n)$$

Data Separation



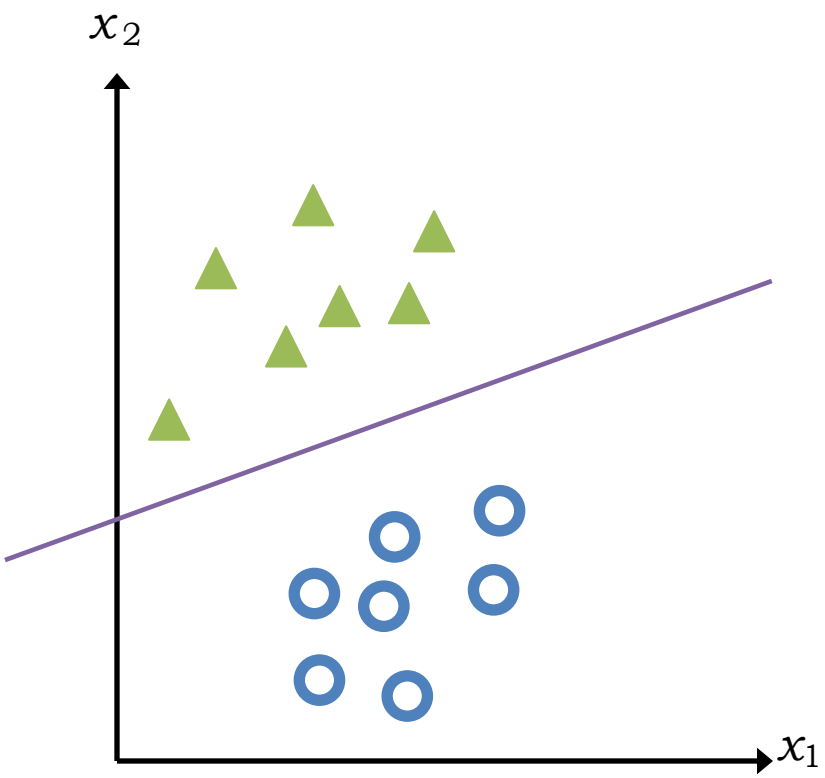
- Linear classification with a perceptron or logistic function looks for a dividing line in the data (or a plane or other linearly defined structure)
  - Often ***multiple*** lines are possible.
  - Essentially, the algorithms are ***indifferent***: they don't care which line we pick.

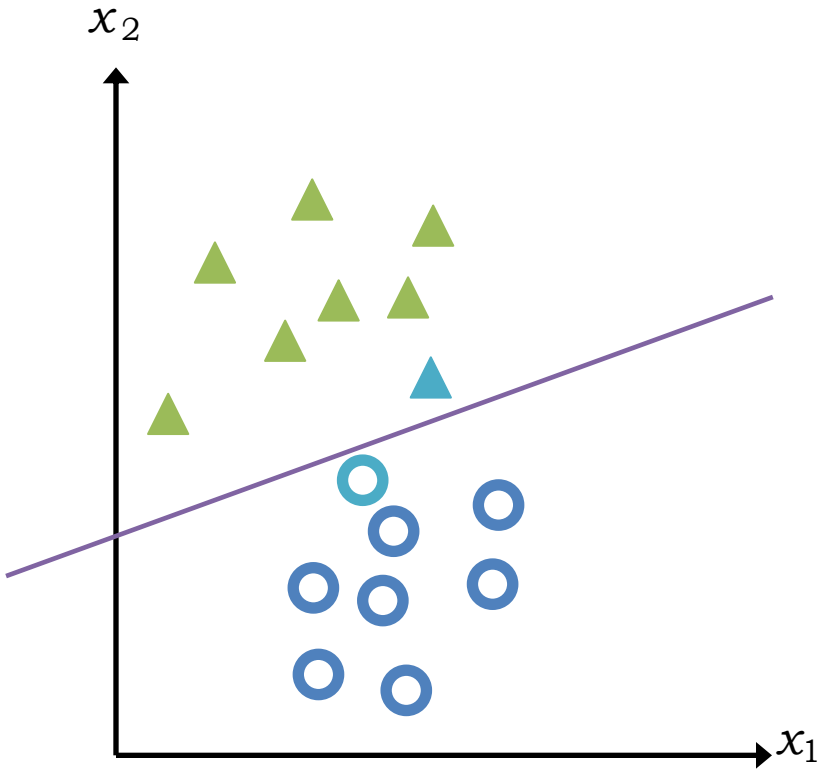
"Fragile" Separation



- As more data comes in, these classifiers may start to fail
  - A separator that is too close to one cluster or the other now makes mistakes
  - This may happen even if new data follows the same distribution seen in the training set

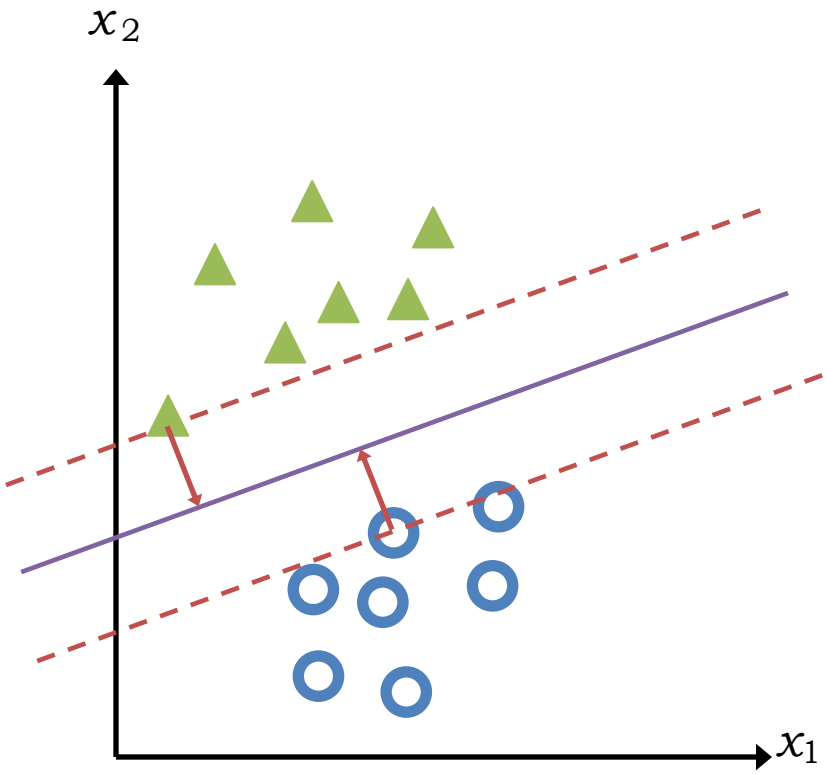
“Robust” Separation





- What we want is a large margin separator: a separation that has the largest distance possible from each part of our data-set
- This will often give much better performance when used on new data

Large Margin Separation



This is sometimes called the "widest road" approach

A support vector machine (SVM) is a technique that finds this road

The points that define the edges of the road are knows as the support vectors

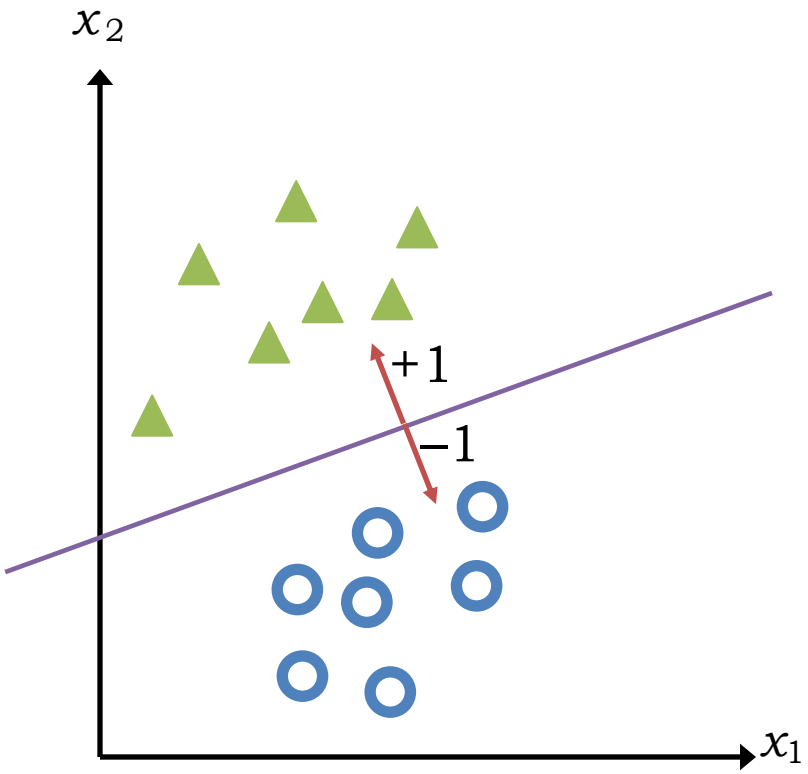
- A new learning problem: ***find*** the separator with the largest margin
- This will be measured from the data points, on opposite sides, that are ***closest together***

Linear Classifiers and SVMs

Linear	
Weight equation	$\mathbf{w} \cdot \mathbf{x} = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n$
Threshold function	$h_{\mathbf{w}} = \begin{cases} 1 & \mathbf{w} \cdot \mathbf{x} \geq 0 \\ 0 & \mathbf{w} \cdot \mathbf{x} < 0 \end{cases}$

SVM	
Weight equation	$\mathbf{w} \cdot \mathbf{x} + b = (w_1 x_1 + w_2 x_2 + \cdots + w_n x_n) + b$
Threshold function	$h_{\mathbf{w}} = \begin{cases} +1 & \mathbf{w} \cdot \mathbf{x} \geq 0 \\ -1 & \mathbf{w} \cdot \mathbf{x} < 0 \end{cases}$

Large Margin Separation



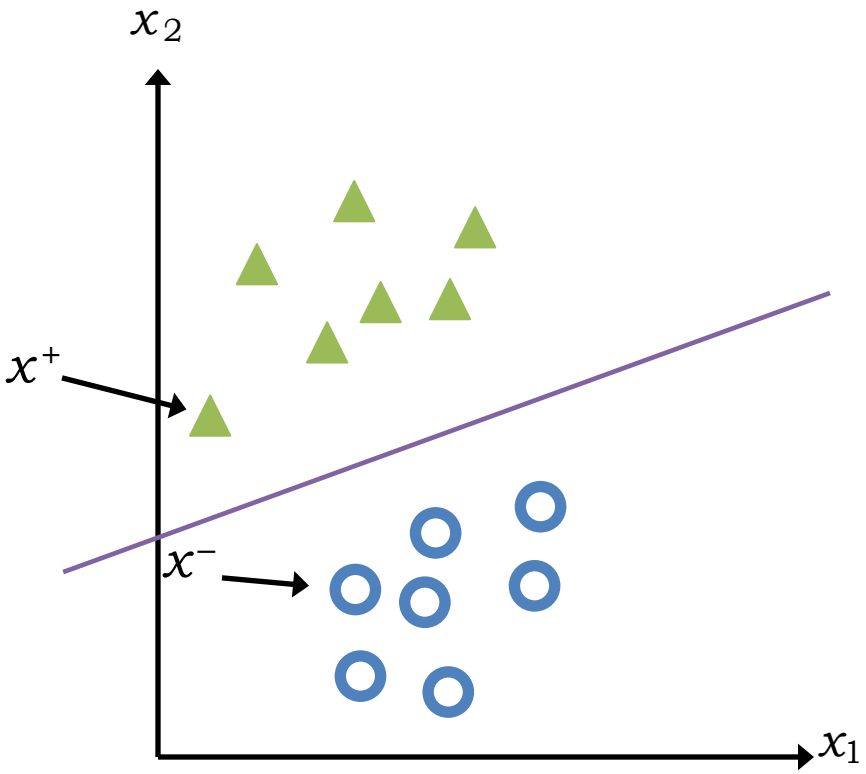
$$\mathbf{w} \cdot \mathbf{x} + b = 0$$

A key difference: the SVM is going to do this ***without*** learning and remembering weight vector $\mathbf{w}$.

Instead, it will use features of the ***data-items themselves***.

- Like a linear classifier, the SVM separates at the line where its learned vector of weights is zero

Mathematics of SVMs



$$\mathbf{w} \cdot \mathbf{x}^+ + b = +1$$
$$\mathbf{w} \cdot \mathbf{x}^- + b = -1$$

$$\mathbf{w} \cdot (\mathbf{x}^+ - \mathbf{x}^-) = 2$$
$$\frac{\mathbf{w}}{||\mathbf{w}||} \cdot (\mathbf{x}^+ - \mathbf{x}^-) = \frac{2}{||\mathbf{w}||}$$

$$||\mathbf{w}|| = \sqrt{w_1^2 + w_2^2 + \cdots + w_n^2}$$

- It turns out that the weight vector $\mathbf{w}$ for the largest margin separator has some essential properties relative to the closest data points on each side (x^+ and x^-)

Mathematics of SVMs

- Through the magic of mathematics (Lagrangian multipliers, to be specific), we can derive a quadratic programming problem

1. We start with our data set:

$$\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)\} \qquad [\forall i,\ y_i \in \{+1, -1\}]$$

2. We then solve a constrained optimization problem:

$$W(\alpha) = \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i\, \alpha_j\, y_i\, y_j (\mathbf{x}_i \cdot \mathbf{x}_j)$$

$$\forall i,\ \alpha_i \geq 0$$

$$\sum_i \alpha_i\, y_i = 0$$

The goal: based on ***known*** values ($\mathbf{x}_i$, y_i) ***find*** the values we ***don't know*** (α_i) that:

1. Will maximize value of margin $W(\alpha)$
2. Satisfy the two numerical constraints

Mathematics of SVMs

- Although complex, a constrained optimization problem like this can be algorithmically solved to get the α_i values we want:

$$W(\alpha) = \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \, \alpha_j \, y_i \, y_j (\mathbf{x}_i \cdot \mathbf{x}_j)$$

$$\forall i, \; \alpha_i \geq 0$$

$$\sum_i \alpha_i \, y_i = 0$$

A note about notation: these equations involve two different, necessary products:

1. The usual application of ***weights*** to ***points***:
$$\mathbf{w} \cdot \mathbf{x}_i = w_1 x_{i,1} + w_2 x_{i,2} + \cdots + w_n x_{i,n}$$
2. Products of ***points*** and ***other points***:
$$\mathbf{x}_i \cdot \mathbf{x}_j = x_{i,1} x_{j,1} + x_{i,2} x_{j,2} + \cdots + x_{i,n} x_{j,n}$$

- Once done, we can find the weight vector and bias term if we want:

$$\mathbf{w} = \sum_i \alpha_i \, y_i \, \mathbf{x}_i \qquad b = -\frac{1}{2} \left(\max_{i \,|\, y_i = -1} \mathbf{w} \cdot \mathbf{x}_i + \min_{j \,|\, y_j = +1} \mathbf{w} \cdot \mathbf{x}_j \right)$$

The Dual Formulation

- It turns out that we don't need to use the weights at all
- Instead, we can use the α_i values ***directly***:

$$\mathbf{w} \cdot \mathbf{x}_i + b = \sum_j \alpha_j y_j (\mathbf{x}_i \cdot \mathbf{x}_j) + b$$

What we ***usually*** look for in a parametric method: the weights, $\mathbf{w}$, and offset, b, defining the classifier

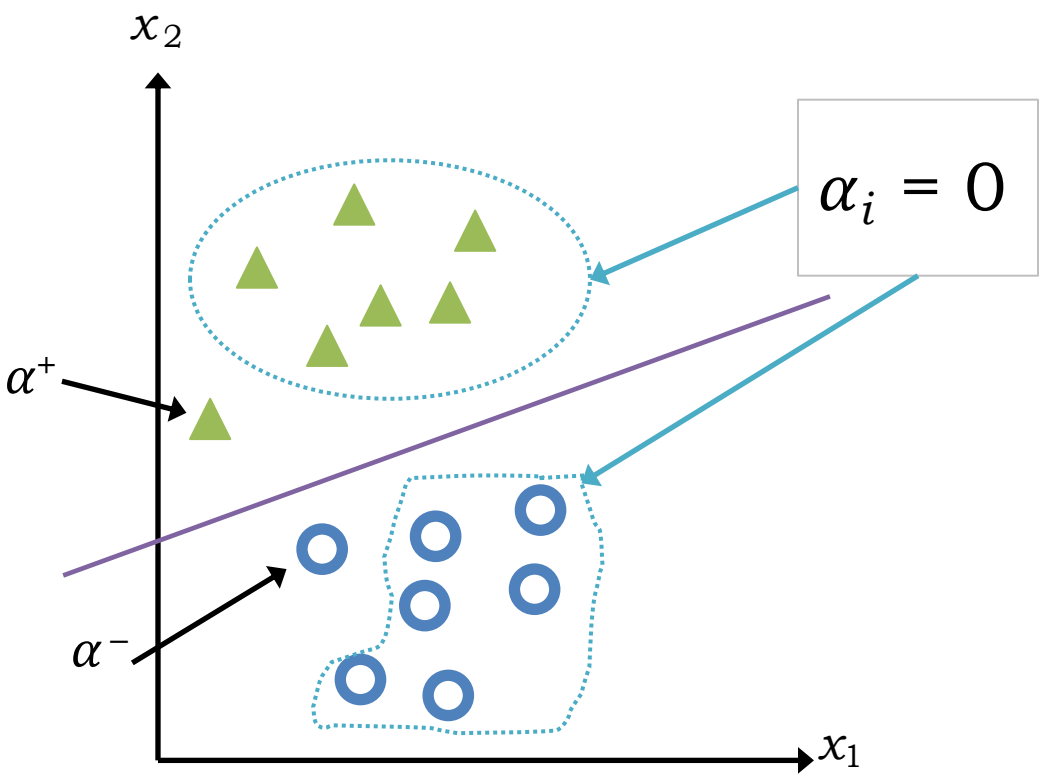
What we can use ***instead***: we compute an ***equivalent*** result based upon the α parameters, the outputs y, and products between data points themselves (along with the standard offset)

The Dual Formulation

$$\mathbf{w} \cdot \mathbf{x}_i + b = \sum_j \alpha_j y_j (\mathbf{x}_i \cdot \mathbf{x}_j) + b$$

- Now, if we had to sum over ***every*** data point as on the right-hand side of this equation, this would look very bad for a large data-set
- It turns out that these α_i values have a special property, however, that makes it feasible to use them as part of our classification function…

Sparseness of SVMs



So, when we do the calculation:

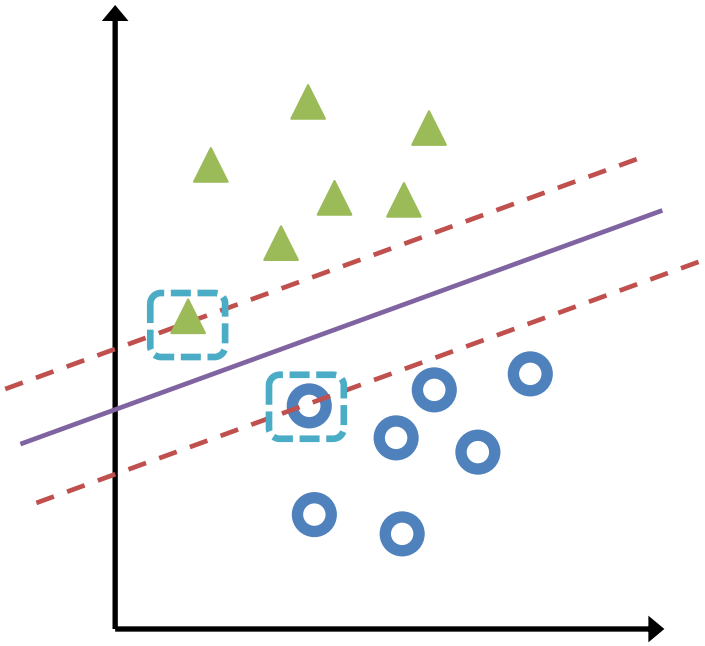
$$\sum_j \alpha_j y_j (\mathbf{x}_i \cdot \mathbf{x}_j) + b$$

We only have to sum over points $\mathbf{x}_j$ that are in the set of support vectors, ***ignoring*** all others, since the related α values are all 0.

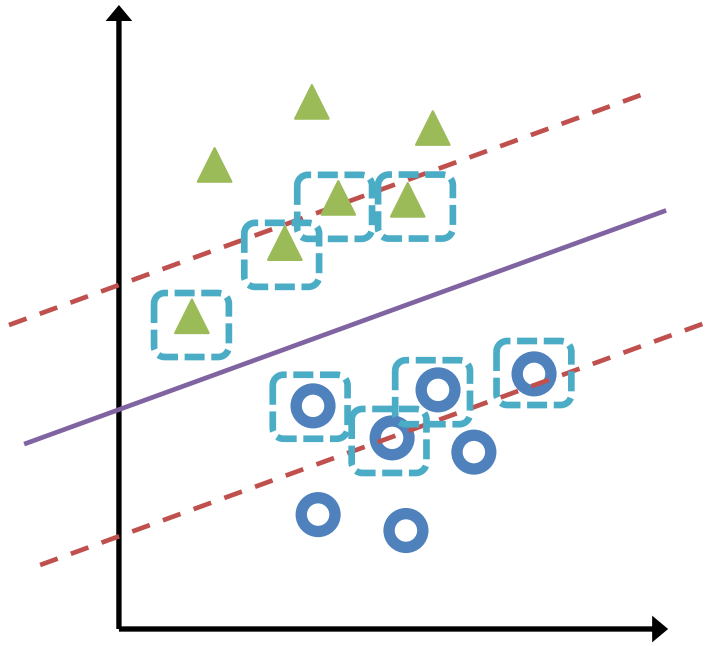
Thus, an SVM need only remember and use the values for a few support vectors, not those for all the rest of the data.

- The α_i values are 0 ***everywhere except*** at the support vectors (the points closest to the separator)

Hard and Soft Margins

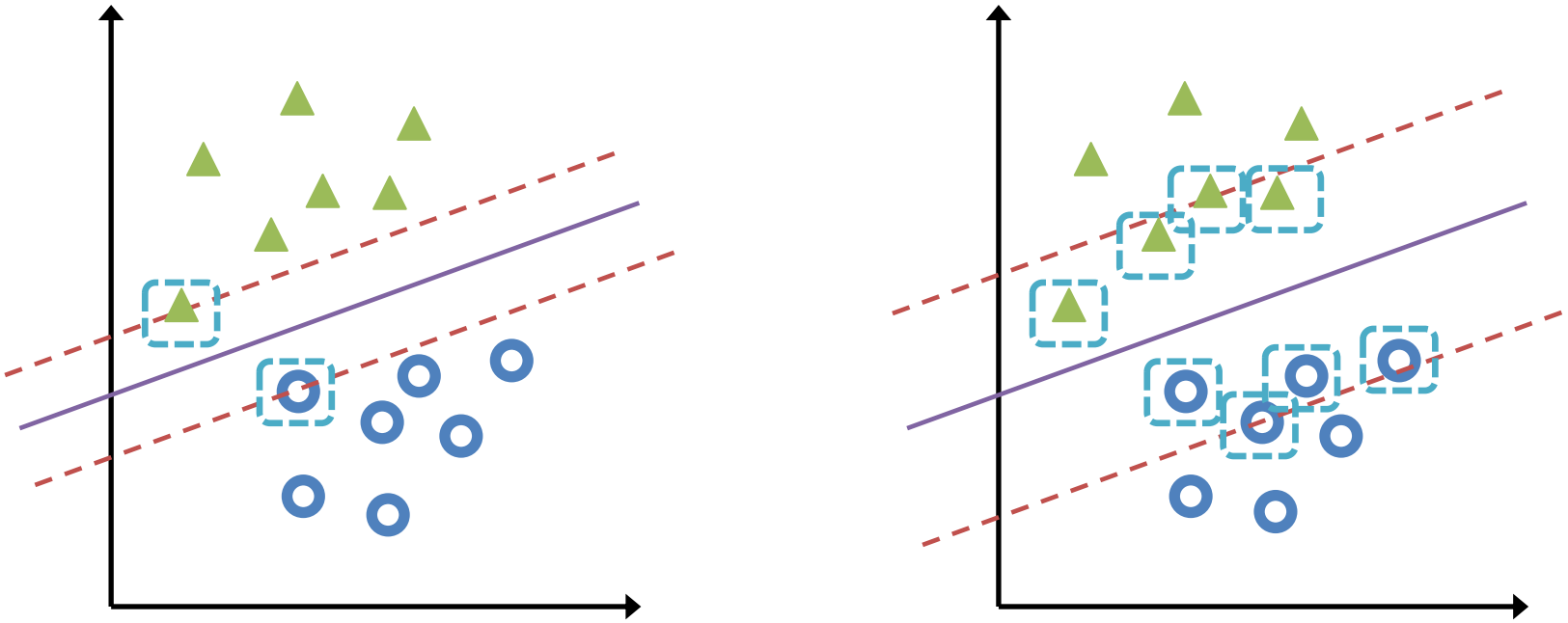


- We have slightly simplified one detail of how most SVMs work
- It is ***not always true*** that the support vectors lie on the margins, with nothing else in between them
- This is only true in the hard-margin case



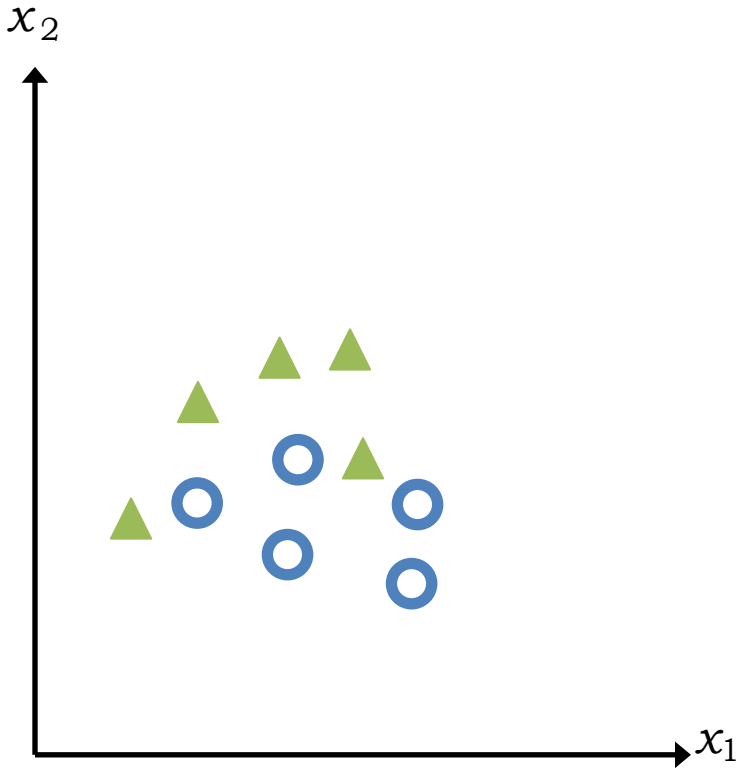
- SVMs can have soft margins instead (and usually do) to deal with noisier data
- We weaken the requirement that no points lie between the margins
  - ***All points*** within the margins then become the support vectors for classification

Hard and Soft Margins



- In upcoming exercises, we will see how to vary margin strength in `sklearn`
- SVM models come with a regularization parameter (`C`, as in the case of classifiers) that can be:
  1. ***Increased*** to enforce a ***harder*** margin
  2. ***Decreased*** to allow a ***softer*** margin

Another Nice Trick

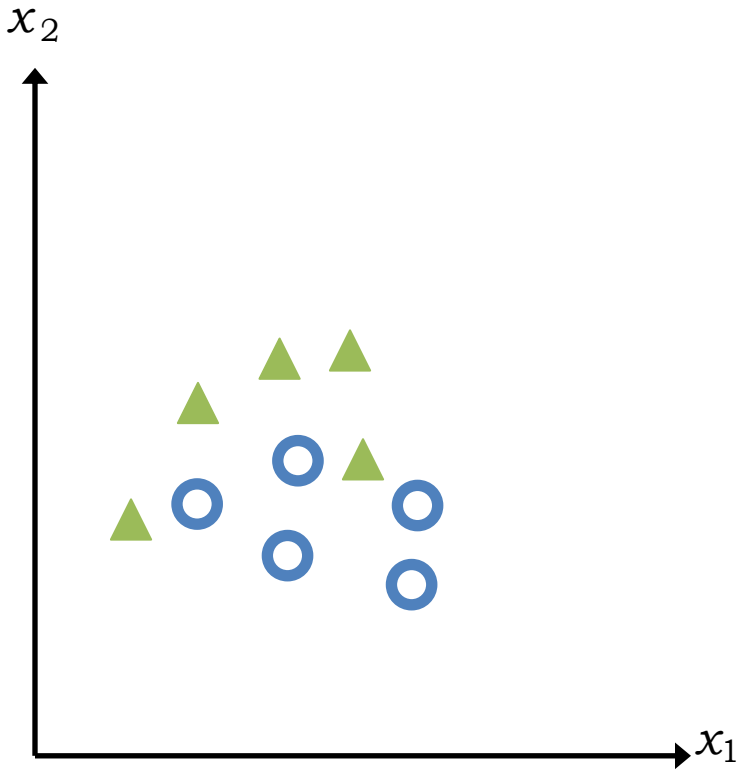


$$\sum_j \alpha_j y_j (\mathbf{x}_i \cdot \mathbf{x}_j) + b$$

Using a kernel"trick", we can find a function that ***transforms*** the data into another form, where it is actually possible to separate it in a linear manner

- The dual formulation uses dot products of data-points with each other (instead of with weights)
- Combining this with the idea of a kernel will allow us to deal with data that is not linearly separable.

Transforming Non-Separable Data



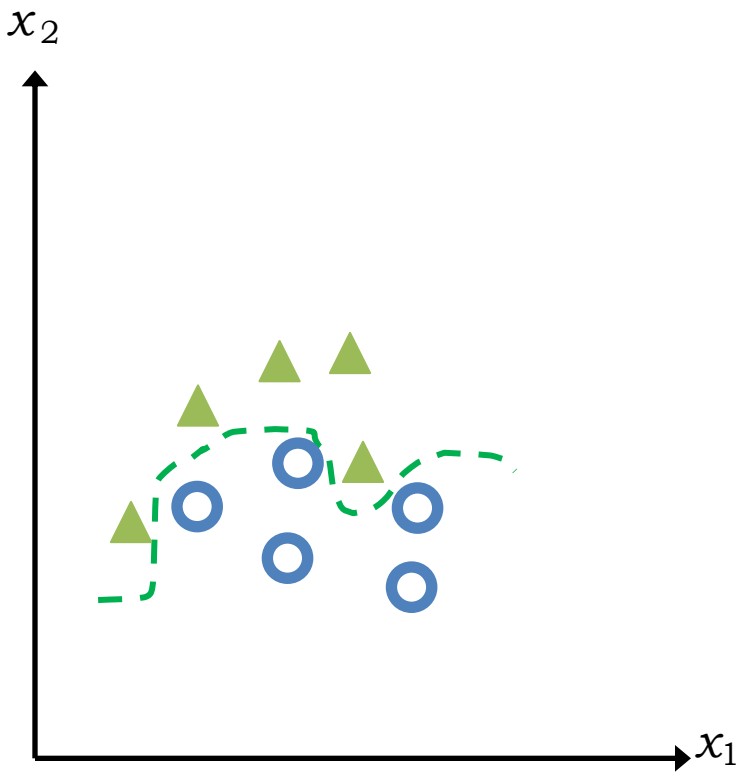
A transformation function:

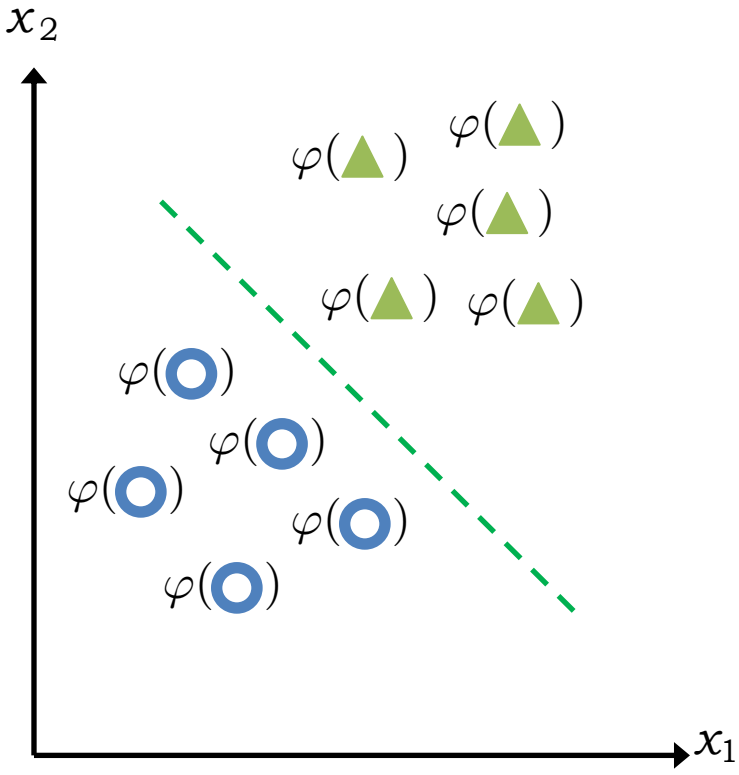
$$\varphi(\mathbf{x}) \qquad \varphi : \mathbb{R}^n \rightarrow \mathbb{R}^m$$

maps data-vectors to new vectors, of either the same dimensionality ($m = n$) or a different one ($m \neq n$)

- If data that is not linearly separable, we can ***transform*** it
  - We ***change*** features used to represent our data
  - As usual, we ***don't care*** what the data feature are, so long as we can get classification to work

Transforming Non-Separable Data

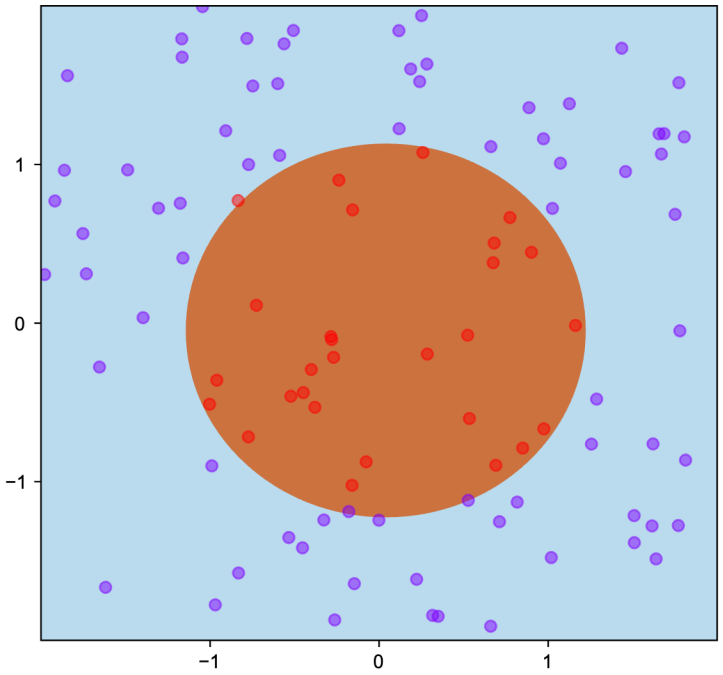


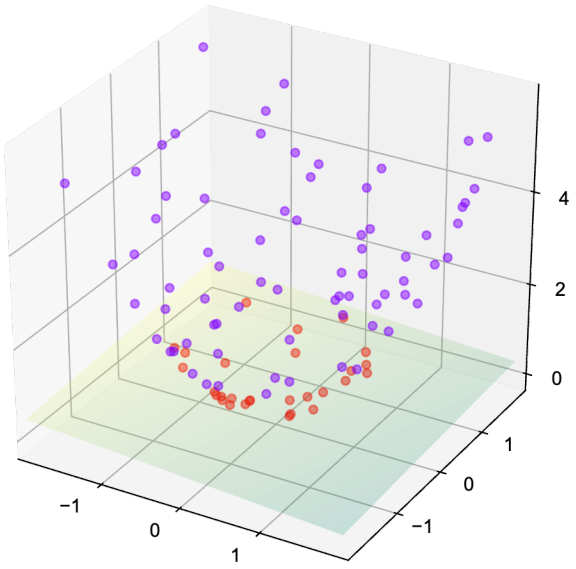


$\varphi(\mathbf{x})$ $\quad$ $\varphi : \mathbb{R}^n \rightarrow \mathbb{R}^m$

$$\sum_j \alpha_j y_j (\varphi(\mathbf{x}_i) \cdot \varphi(\mathbf{x}_j)) + b$$

The “Kernel Trick”





$$\varphi(x_1, x_2) = (x_1^2, x_2^2, \sqrt{2}x_1x_2)$$

Simplifying the Transformation Function

- We can derive a simpler (2-dimensional) equation, ***equivalent to*** the cross-product needed when doing SVM computations in the transformed (3-dimensional) space:

$$\varphi(\mathbf{x}) \cdot \varphi(\mathbf{z}) = (x_1^2,\ x_2^2,\ \sqrt{2}x_1x_2) \cdot (z_1^2,\ z_2^2,\ \sqrt{2}z_1z_2)$$

Needed

$$= x_1^2z_1^2 + x_2^2z_2^2 + \sqrt{2}x_1x_2\sqrt{2}z_1z_2$$

$$= x_1^2z_1^2 + x_2^2z_2^2 + 2x_1x_2z_1z_2$$

10 multiplications
2 additions

$$= (x_1z_1 + x_2z_2)^2$$

3 multiplications
1 addition

$$= (\mathbf{x} \cdot \mathbf{z})^2$$

Used instead

The Kernel Function

$$k(\mathbf{x}, \mathbf{z}) = \varphi(\mathbf{x}) \cdot \varphi(\mathbf{z}) = (\mathbf{x} \cdot \mathbf{z})^2$$

- This final function (right side) is what the SVM will use to compute dot products in its equations
- This is called the kernel function
- To make SVMs useful, we look for a kernel that:

1. Separates the data usefully
2. It is relatively efficient to calculate

Another Reason to Use Kernel Functions

$$\mathbf{w} \cdot \mathbf{x}_i + b = \sum_j \alpha_j y_j (\varphi(\mathbf{x}_i) \cdot \varphi(\mathbf{x}_j)) + b$$

- The SVM formulation generally uses dot products of data points (perhaps run through some kernel) rather than the standard product of features and weights.
- We have cases where the kernel-data approach is ***possible***, but the weights-based one is ***not.***
  - Some useful kernels that are easy to compute correspond to weight equations applied to ***very high dimensional*** transforms of the original data.
  - In some common cases, equivalent weight-data vectors are ***infinite-***dimensional and cannot be used in the computation.