CS 457/557: Machine Learning

Lecture 04-*:
Linear Classification

University of Wisconsin
LA CROSSE

Prof. Jason Sauppe
jsauppe@uwlax.edu
https://cs.uwlax.edu/~jsauppe/

Fall 2024

Lecture 04-1a:
Motivating Classification

Quick Recap

Supervised Learning Problem

Given a *labeled* data set $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$ with $y = f(\mathbf{x})$ for some unknown function f, identify a **hypothesis function** h that approximates f well.

The type of learning problem depends on the **output values** y_i:

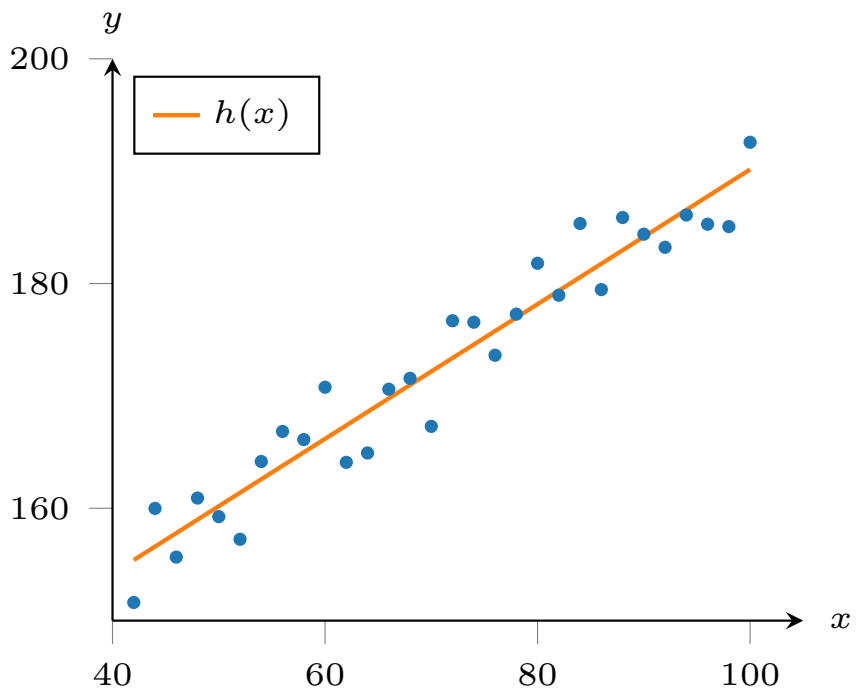
- If each y_i is **continuous-valued**, then it is a **regression** problem.
- If each y_i takes values from a **finite discrete set**, then it is a **classification** problem.

Canonical examples of classification:

- Spam or ham? (email classification)
- Fraudulent or legitimate transaction?
- Default on loan or not?

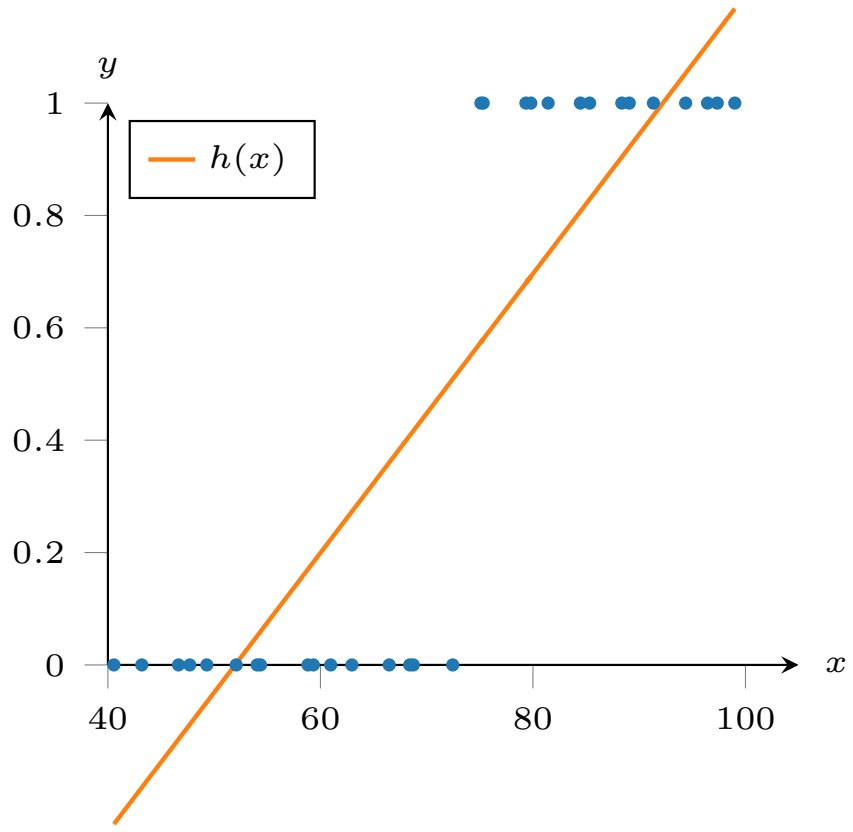
Comparing Regression and Classification

Regression with one input and one output:



We draw a line to **fit** the data.

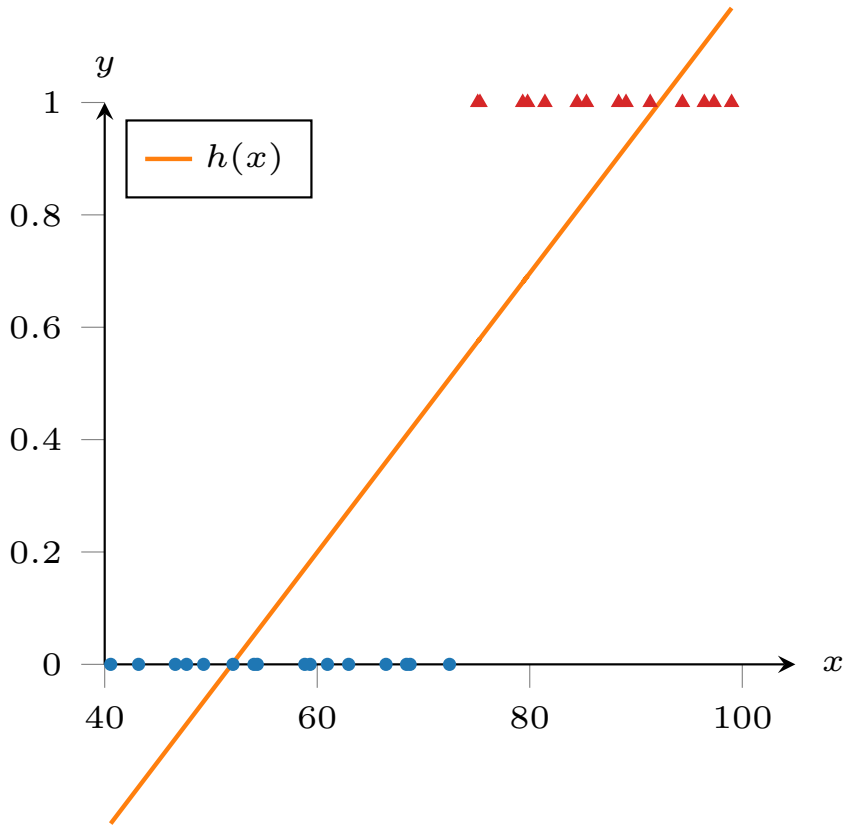
Classification with one input and one output:



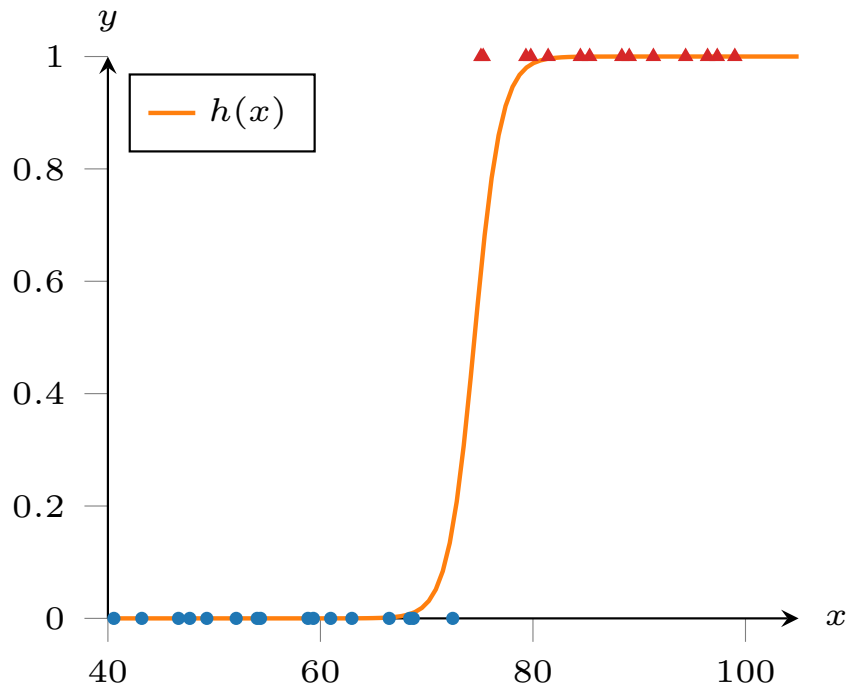
Drawing a line **doesn't quite work**...

Finding a Better Fit

Fitting a line to the data as in linear regression doesn't work:



What we'd like instead is something more like this:



Predicted values should be in $[0, 1]$.

Lecture 04-1b:
Simple Logistic Regression

Mapping to $[0, 1]$

Suppose we have a single predictor attribute x with binary target y.

Question: What hypothesis functions h should we consider?

If we make predictions with a simple linear function, we would have

$$\hat{y} = w_0 + w_1 x.$$

This produces predicted values in $\mathbb{R}$, which **is not** what we want...

To **constrain** the predicted values to be in $[0, 1]$, we can wrap the simple linear function inside another function

$$\hat{y} = \sigma\left(w_0 + w_1 x\right)$$

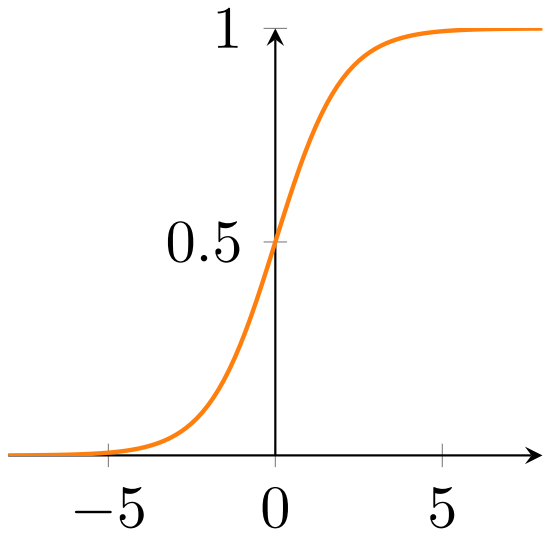
where $\sigma : \mathbb{R} \to [0, 1]$ is the **standard logistic function**.

Properties of the Logistic Function

The **standard logistic function** $\sigma : \mathbb{R} \to [0, 1]$ is defined as

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

for all $t \in \mathbb{R}$.



- σ is a **sigmoid** function
- Output is always in $[0, 1]$
- $\sigma(0) = 0.5$
- $\lim_{t \to \infty} \sigma(t) = 1$
- $\lim_{t \to -\infty} \sigma(t) = 0$
- σ is everywhere differentiable, with $\sigma'(t) = \sigma(t)(1 - \sigma(t))$

Simple Logistic Regression

Definition

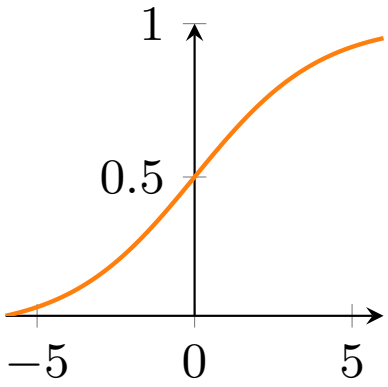
Simple logistic regression involves a single predictor attribute x and binary target attribute y and considers hypothesis functions of the form

$$h_{\mathbf{w}}(x) = \sigma\left(w_0 + w_1 x\right) = \frac{1}{1 + e^{-(w_0 + w_1 x)}}.$$

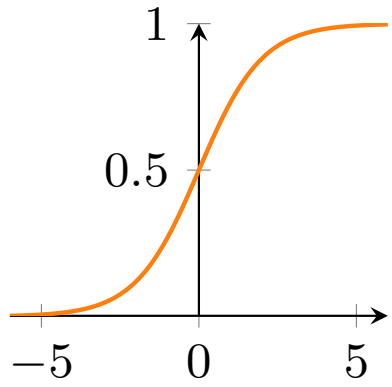
- The values returned by $h_{\mathbf{w}}(x)$ are in the interval $[0, 1]$ and can be interpreted as the **probability** of belonging to class 1 (sometimes called the "positive" class)
  - By **complementarity**, $1 - h_{\mathbf{w}}(x)$ represents the **probability** of belonging to class 0 (sometimes called the "negative" class)
- By itself, $h_{\mathbf{w}}$ is **not a classifier** (i.e., an algorithm that predicts labels), though we can produce labels with a little extra work (we'll see more on this shortly!)

Impact of Weights in Simple Logistic Regression

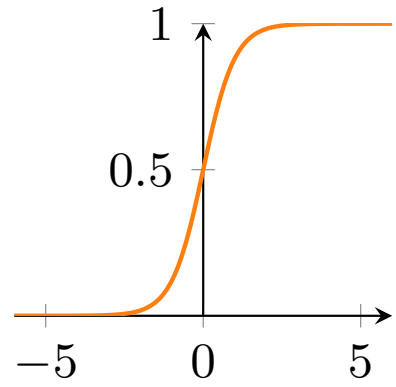
$w_0 = 0$, $w_1 = 0.5$:



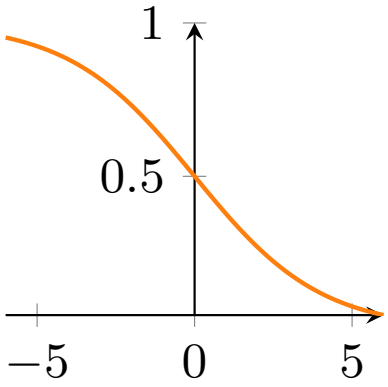
$w_0 = 0$, $w_1 = 1$:



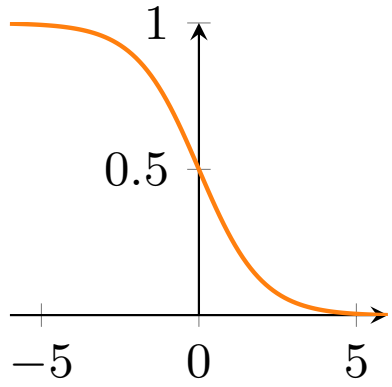
$w_0 = 0$, $w_1 = 2$:



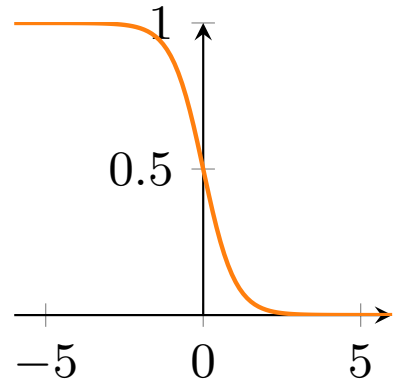
$w_0 = 0$, $w_1 = -0.5$:



$w_0 = 0$, $w_1 = -1$:



$w_0 = 0$, $w_1 = -2$:



Changing w_0 shifts left or right

Lecture 04-1c:
Logistic Regression

Logistic Regression with Multiple Predictor Attributes

Logistic regression as a supervised learning system:

- Data: $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$, with p input attributes
- Hypothesis space:

$$\mathcal{H} = \left\{ h : \mathbb{R}^p \to [0,1] \mid h_{\mathbf{w}}(\mathbf{x}) = \sigma\left(w_0 + \sum_{j=1}^{p} w_j x_j \right) \right\}$$

  where $\sigma(t) = \frac{1}{1+e^{-t}}$
- Learning algorithm:
  - Loss function: ???
  - Cost function: average loss (whatever that is)
  - Fitting method: Gradient descent (most likely)

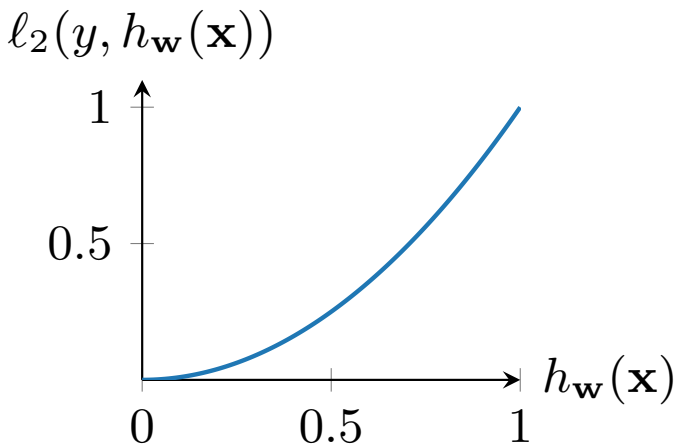
We need to figure out a few more details!

Loss Functions for Logistic Regression: Squared Loss

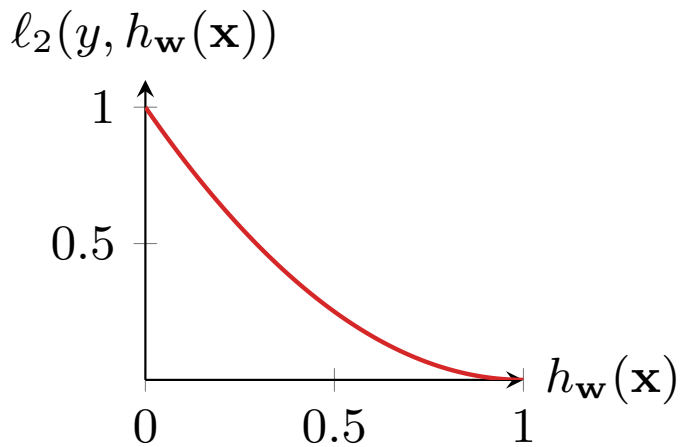
One option for the loss function in logistic regression is squared error:

$$\ell_2\left(y, h_{\mathbf{w}}(\mathbf{x})\right) = \left(y - h_{\mathbf{w}}(\mathbf{x})\right)^2$$

With $y = 0$, loss is $\left(-h_{\mathbf{w}}(\mathbf{x})\right)^2$



With $y = 1$, loss is $\left(1 - h_{\mathbf{w}}(\mathbf{x})\right)^2$



- $h_{\mathbf{w}}(\mathbf{x}) \in [0, 1]$, so loss is bounded between 0 and 1
- Yields a **nonconvex** cost function for finding $\mathbf{w}^*$
  (gradient descent is not guaranteed to converge to global minimum)

Loss Functions for Logistic Regression: Log-Loss

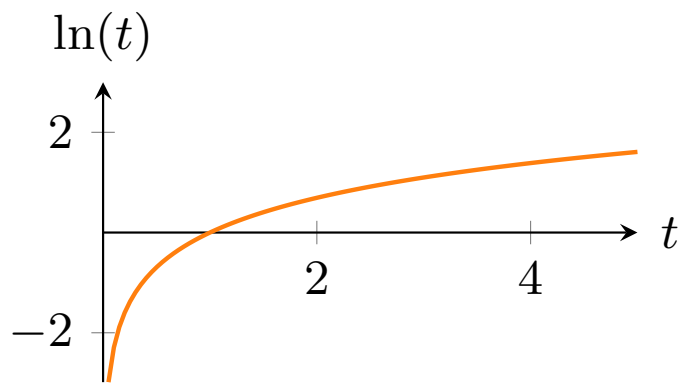
Another loss function is **log-loss** (**cross entropy loss**):

$$\ell_{\log}\left(y, h_{\mathbf{w}}(\mathbf{x})\right) = \begin{cases} -\log\left(h_{\mathbf{w}}(\mathbf{x})\right) & \text{if } y = 1 \\ -\log\left(1 - h_{\mathbf{w}}(\mathbf{x})\right) & \text{if } y = 0 \end{cases}$$
$$= -1\left[y \log\left(h_{\mathbf{w}}(\mathbf{x})\right) + (1-y)\log\left(1 - h_{\mathbf{w}}(\mathbf{x})\right)\right]$$

Recall: $\log_b a = c$ if and only if $a = b^c$.

Typically the **natural logarithm** is used here (changing base b just scales the loss).

- $\ln(0) = -\infty$ because $0 = e^{-\infty}$
- $\ln(1) = 0$ because $1 = e^0$
- $\ln(2) \approx 0.693$ because $2 \approx e^{0.693}$
- $\ln(2.718) \approx \ln(e) = 1$ because $e = e^1$

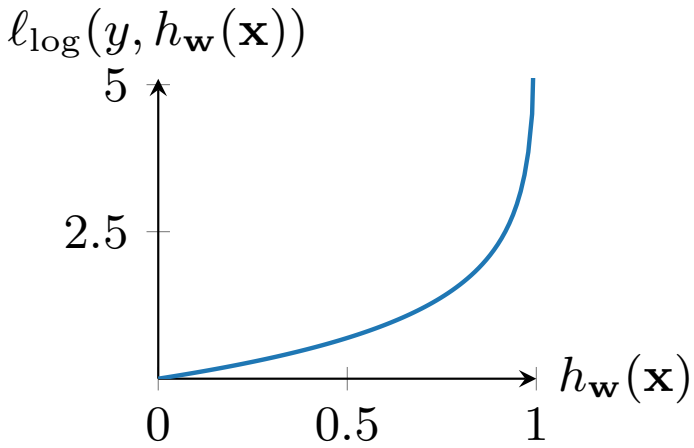


Loss Functions for Logistic Regression: Log-Loss

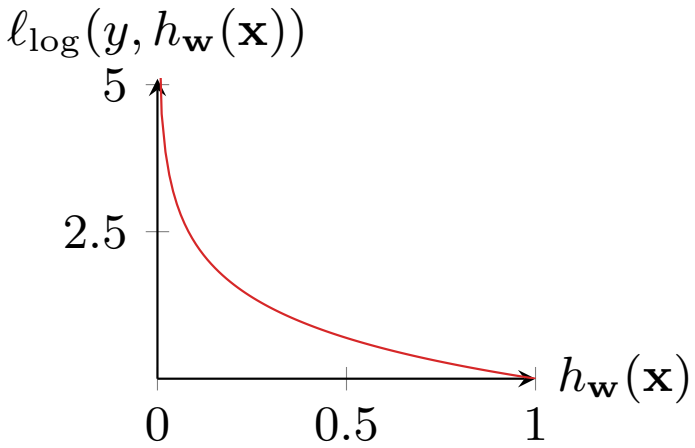
Another loss function is **log-loss** (**cross entropy loss**):

$$\ell_{\log}\left(y, h_{\mathbf{w}}(\mathbf{x})\right) = \begin{cases} -\log\left(h_{\mathbf{w}}(\mathbf{x})\right) & \text{if } y = 1 \\ -\log\left(1 - h_{\mathbf{w}}(\mathbf{x})\right) & \text{if } y = 0 \end{cases}$$
$$= -1\left[y \log\left(h_{\mathbf{w}}(\mathbf{x})\right) + (1-y)\log\left(1 - h_{\mathbf{w}}(\mathbf{x})\right)\right]$$

With $y = 0$, loss is $-\ln\left(1 - h_{\mathbf{w}}(\mathbf{x})\right)$:

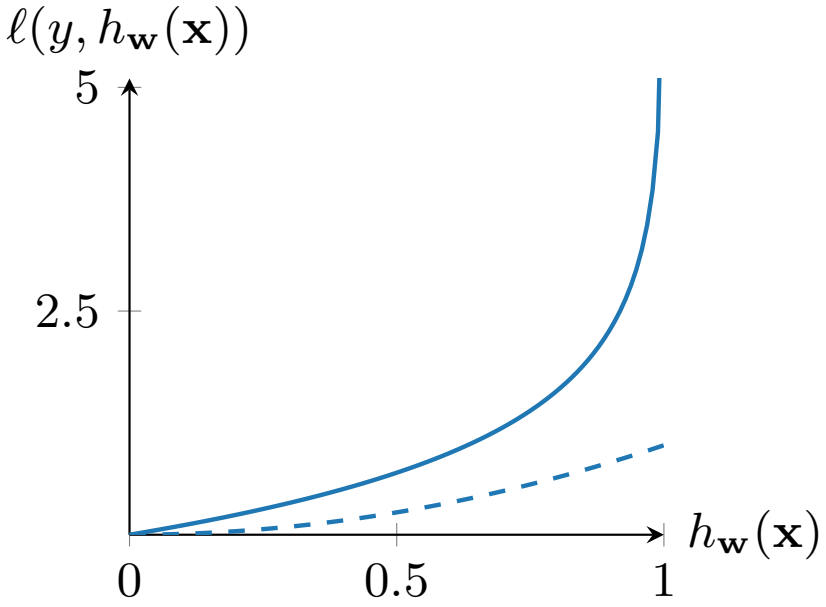


With $y = 1$, loss is $-\ln\left(h_{\mathbf{w}}(\mathbf{x})\right)$:

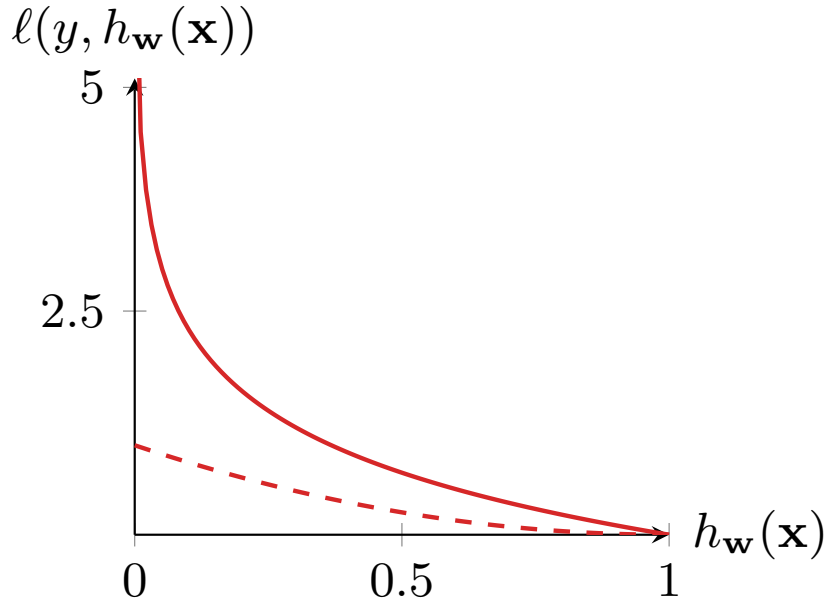


Comparing Squared Loss and Log-Loss

With $y = 0$:



With $y = 1$:



- **Log-loss** is shown as solid lines, **squared** loss is shown as dashed lines
- **Log-loss** penalizes wrong answers **more severely** than **squared loss**
- **Log-loss** leads to a **convex** cost function!
- **Log-loss** also has connections to maximum likelihood estimation

Logistic Regression Components, Revisited

Logistic regression as a supervised learning system:

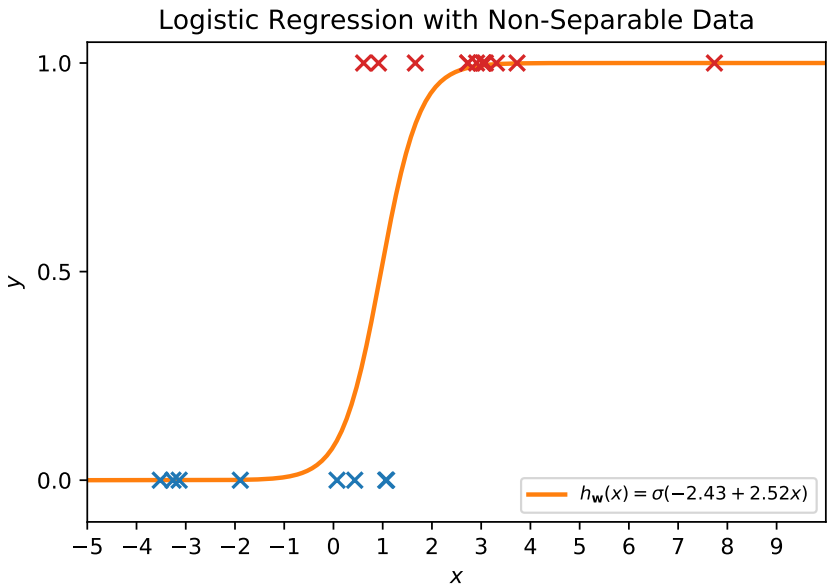
- Data: $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$, with p input attributes
- Hypothesis space:

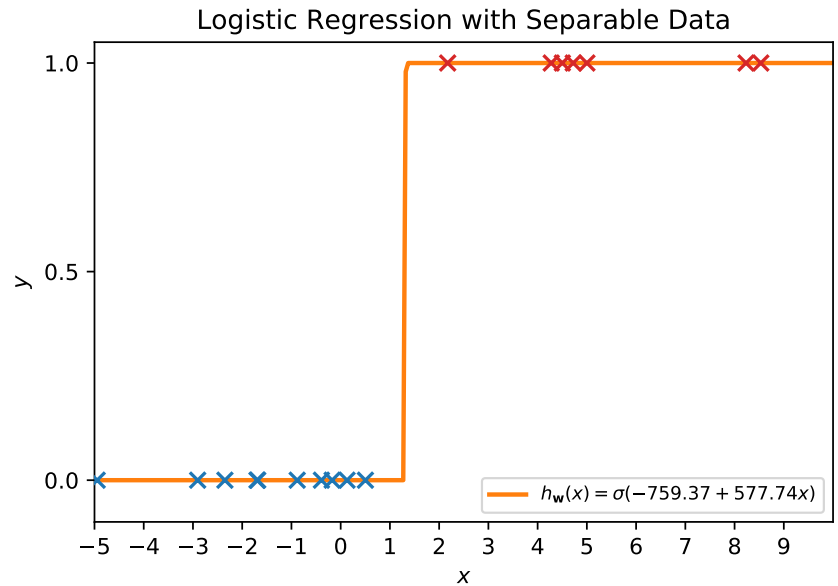
$$\mathcal{H} = \left\{ h : \mathbb{R}^p \to [0,1] \mid h(\mathbf{x}) = \sigma\left(w_0 + \sum_{j=1}^{p} w_j x_j \right) \right\}$$

  where $\sigma(t) = \frac{1}{1+e^{-t}}$
- Learning algorithm:
  - Loss function: **log-loss**
  - Cost function: average loss **with regularization**
  - Fitting method: **gradient descent**
    (we just need the partial derivatives of the cost function; obtaining them is left as an exercise!)

Why Regularization?

Using average log-loss as the cost function gives us a **convex** function, which we can **minimize** with gradient descent. There is one issue though...





With **separable** data, making the curve steeper always reduces the cost! Gradient descent can make the curve steeper by increasing the magnitude of the weights, so it will choose to do so unless we **penalize** large weights using **regularization**.

History of Logistic Regression (1838–1847)

- The logistic function and its name come from three papers by Pierre François Verhulst (right), a statistician and student of Alphonse Quételet (left)

- They were interested in modeling human population growth, which will tend to grow exponentially unless checked, but has an upper bound (equilibrium) at which it maxes out and stops growing

- The Sigmoid curve was a good fit for real population data for France, Belgium, and Russia up to the year 1833

History of Logistic Regression (20th C.)

- The logistic was re-discovered by Raymond Pearl (left) and Lowell Reed (right) in the 1920's
- They later discovered Verhulst's earlier work, and credited him, but his logistic terminology didn't really catch on until the work of others, after WWII
- Pearl and collaborators went on to apply the logistic curve to models of human and fruit fly populations, as well as to the growth of cantaloupes

- In the 40's and 50's, statisticians working to model bioassay (effects of medicines and other substances on living tissues) popularized the use of the logistic and its name
- Due to computational conveniences, this became more popular than other models

Lecture 04-2a:
Classification with Logistic Regression

Logistic Regression for Classification

In **logistic regression**, we consider hypothesis functions of the form

$$h_{\mathbf{w}}(\mathbf{x}) = \sigma\left(w_0 + \sum_{j=1}^{p} w_j x_j\right) = \frac{1}{1 + e^{-\left(w_0 + \sum_{j=1}^{p} w_j x_j\right)}}.$$

By itself $h_{\mathbf{w}}$ is **not a classifier**!

To create a **classifier** (i.e., an algorithm that produces labels), we add **hard thresholding** with a cutoff value $t \in [0, 1]$. Then for a vector of input attributes $\mathbf{x}$:

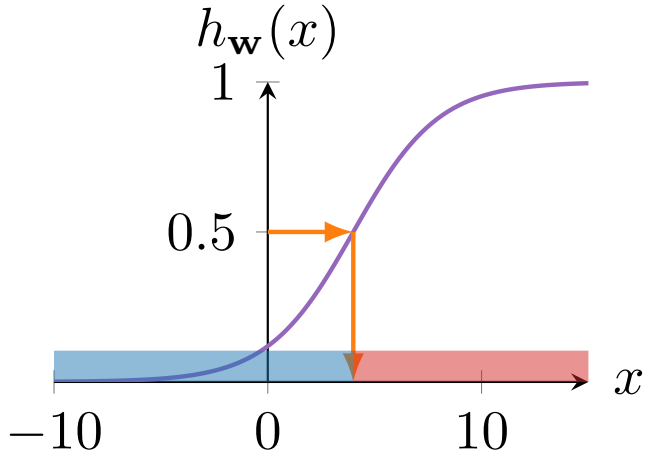
- The predicted label is 1 if $h_{\mathbf{w}}(\mathbf{x}) \geq t$
- The predicted label is 0 if $h_{\mathbf{w}}(\mathbf{x}) < t$

This is called a **classification rule** (**decision rule**).

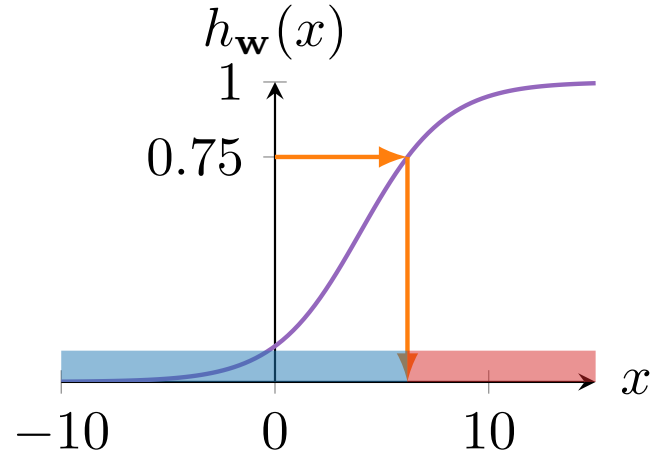
Classification with Thresholds

For logistic regression with a single input attribute x:

$w_0 = -2$, $w_1 = 0.5$ with $t = 0.5$:



$w_0 = -2$, $w_1 = 0.5$ with $t = 0.75$:



- **Different** values of t lead to **different** classifiers
- Rules can also decide not to classify points at all (e.g., if $h_{\mathbf{w}}(\mathbf{x})$ is close to 0.5)

Lecture 04-2b:
Decision Boundaries

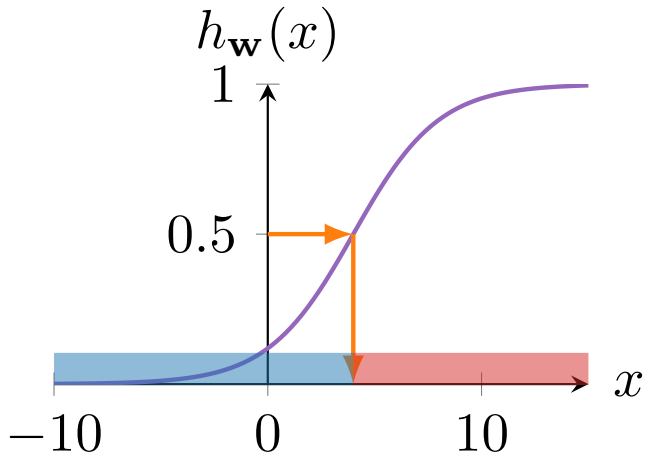
Decision Boundaries

Definition

The **decision boundary** of a classifier is the hyperplane (hypersurface) that partitions feature space into separate sets based on predicted label.

Example for logistic regression with one attribute:

$w_0 = -2$, $w_1 = 0.5$ with $t = 0.5$:



- Feature space is a one-dimensional space (i.e., a line, represented by the x-axis in the plot)
- The decision boundary is a zero-dimensional space (i.e., a point) on the x-axis where the predicted class changes from 0 (blue) to 1 (red)

Finding Decision Boundaries

For logistic regression with p attributes, learned weights $\mathbf{w}$, and a threshold of t, the decision boundary is given by all points $\mathbf{x} \in \mathbb{R}^p$ that satisfy $h_{\mathbf{w}}(\mathbf{x}) = t$. We can solve for these points as follows:

$$
\begin{aligned}
h_{\mathbf{w}}(\mathbf{x}) = t \quad &\Leftrightarrow \quad \frac{1}{1 + e^{-\left(w_0 + \sum_{j=1}^{p} w_j x_j\right)}} = t \\
&\Leftrightarrow \quad e^{-\left(w_0 + \sum_{j=1}^{p} w_j x_j\right)} = \frac{1-t}{t} && [\textit{rearranging}] \\
&\Leftrightarrow \quad -\left(w_0 + \sum_{j=1}^{p} w_j x_j\right) = \ln\left(\frac{1-t}{t}\right) && [\textit{taking natural log}] \\
&\Leftrightarrow \quad w_0 + \sum_{j=1}^{p} w_j x_j = \ln(t) - \ln(1-t) && [\textit{property of } \ln]
\end{aligned}
$$

Finding Decision Boundaries (continued)

From the previous slide, we saw that

$$h_{\mathbf{w}}(\mathbf{x}) = t \quad \Leftrightarrow \quad w_0 + \sum_{j=1}^{p} w_j x_j = \ln(t) - \ln(1-t).$$

For any choice of t, $\ln(t) - \ln(1-t)$ is a constant, so the above equation defines a **hyperplane**, which is the decision boundary of the classifier.

- For $t = 0.5$, the above equation simplifies to:

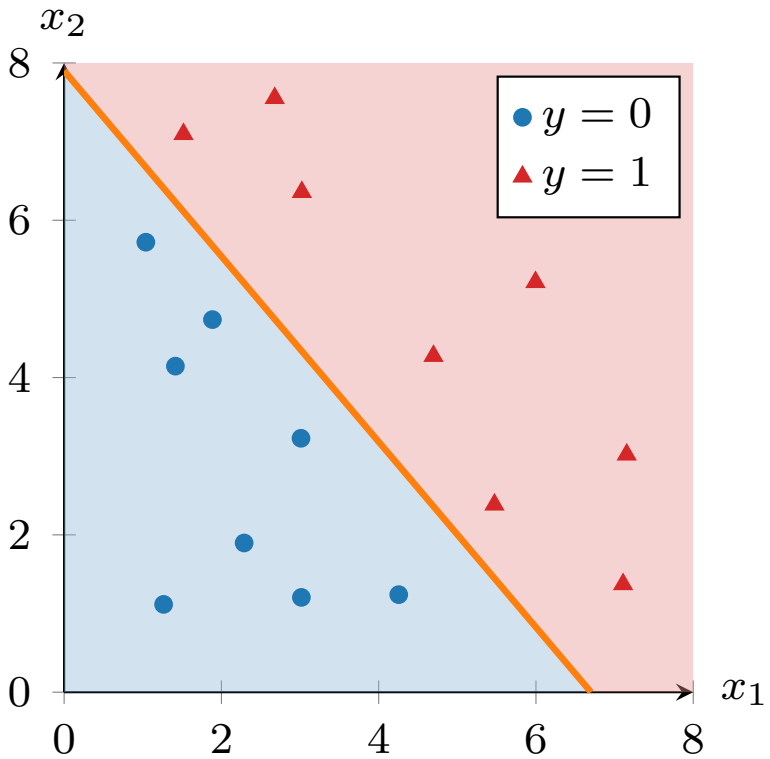
$$w_0 + \sum_{j=1}^{p} w_j x_j = 0.$$

- For $p = 2$ (i.e., two features), the decision boundary is the **line** given by

$$w_0 + w_1 x_1 + w_2 x_2 = 0.$$

Decision Boundaries with Two Input Attributes

With two features, a typical **decision boundary** is a **line** in feature space:



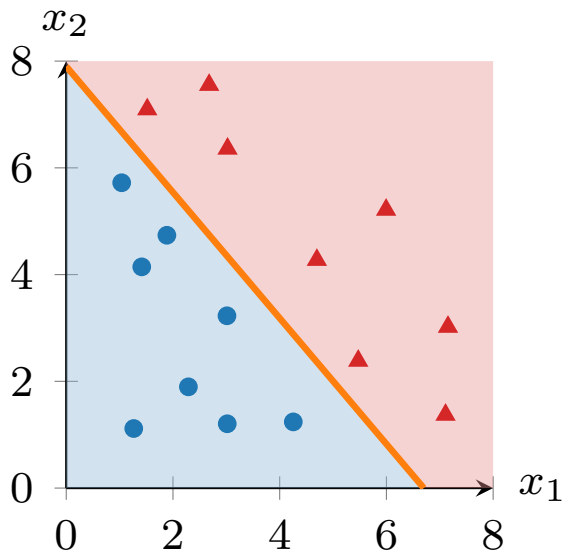
- Each data point $(\mathbf{x}_i, y_i)$ is drawn as an (x_{i1}, x_{i2}) pair, with marker color indicating y_i value
- The orange line represents all points $\mathbf{x} = (x_1, x_2)$ with $h_{\mathbf{w}}(\mathbf{x}) = t$

Linear Separability

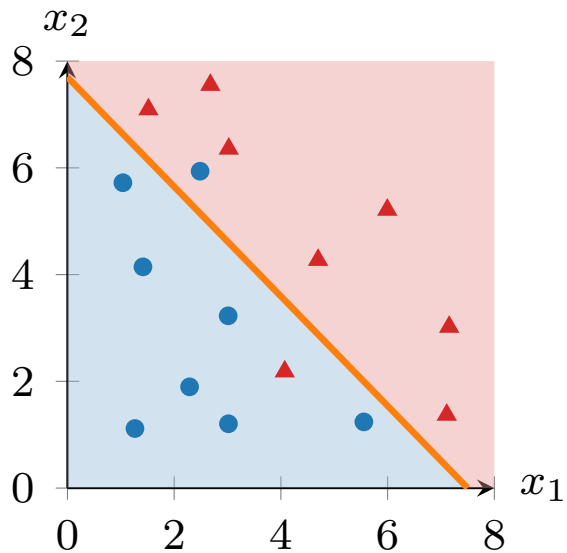
Definition

A data set $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$ with $\mathbf{x}_i \in \mathbb{R}^p$ and $y_i \in \{0, 1\}$ is **linearly separable** if there exists a hyperplane in $\mathbb{R}^p$ that separates the sets of points $\{\mathbf{x}_i \mid y_i = 0\}$ and $\{\mathbf{x}_i \mid y_i = 1\}$.

The data set below **is linearly separable**:

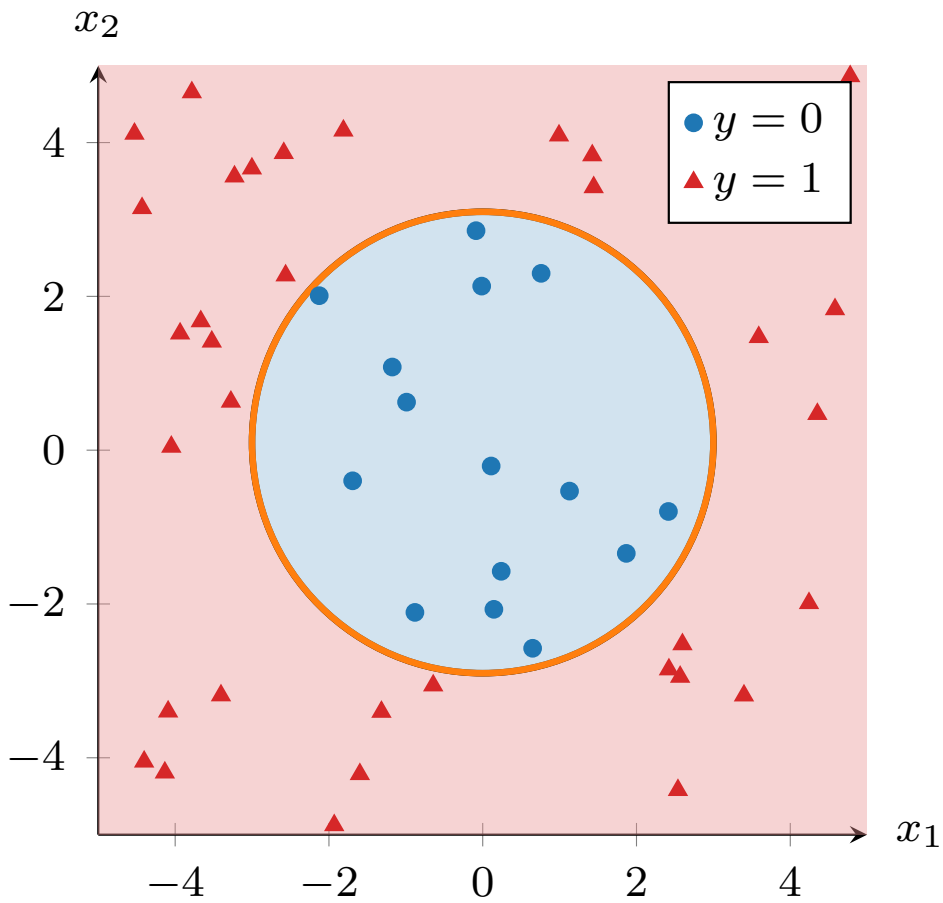


The data set below **is not linearly separable**:



Nonlinear Decision Boundaries

Sometimes a **nonlinear** decision boundary is needed:



- Here we consider hypothesis functions of the form

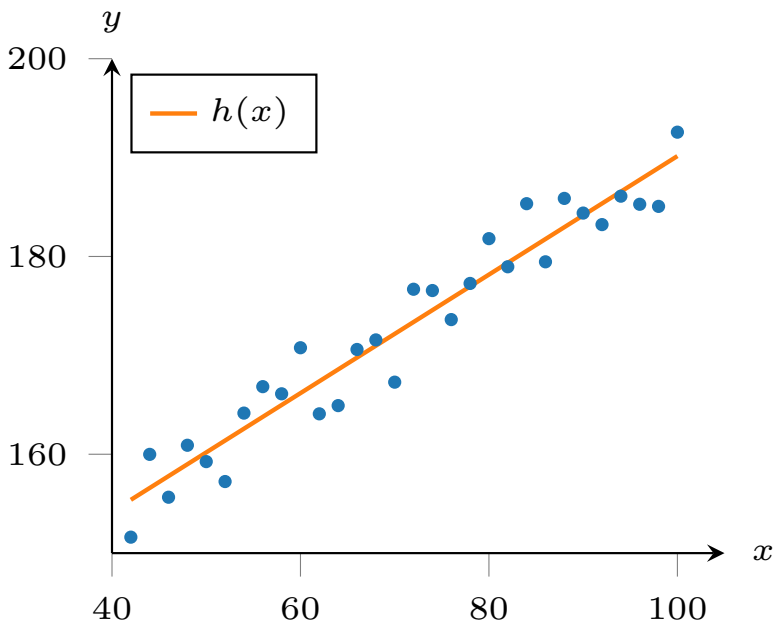
$$h_{\mathbf{w}}(\mathbf{x}) = \sigma\big(w_0 + w_1 x_1 + w_2 x_2 + w_3 x_1^2 + w_4 x_2^2\big)$$

- The orange circle represents all points $\mathbf{x} = (x_1, x_2)$ with $h_{\mathbf{w}}(\mathbf{x}) = 0.5$; because $h_{\mathbf{w}}$ includes **higher-order** terms, this yields a **nonlinear** decision boundary in feature space
- **Feature engineering** is also important in logistic regression

Lecture 04-3a:
Linear Classification

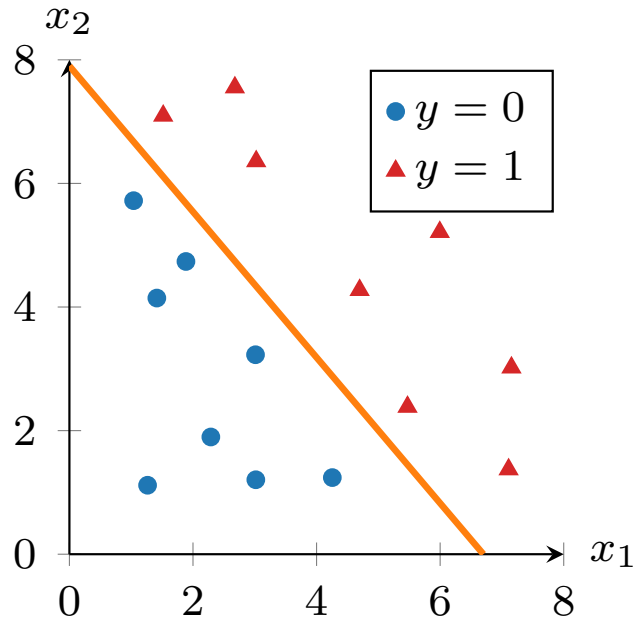
Drawing Lines

Linear regression:
Draw a line to **fit** the data!



- Line is drawn in **input-output space**

Linear classification:
Draw a line to **separate** the data!



- Line is drawn in **input (feature) space**

Linear Classification

This suggests another approach to **classification**: try to find a "line" in feature space that separates the classes!

For the following, we'll assume that we have a data set $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$ with $\mathbf{x}_i \in \mathbb{R}^p$ and $y_i \in \{0, 1\}$ (outputs 0 and 1 are *arbitrary labels* for the two possible classes).

- For $p = 1$, we want a **point** (threshold) that separates the classes
- For $p = 2$, we want a **line** that separates the classes
- For $p = 3$, we want a **plane** that separates the classes
- For $p > 3$, we want a **hyperplane** that separates the classes

Hyperplanes

Definition

A **hyperplane** in $\mathbb{R}^p$ is the set of points $\{\mathbf{x} \in \mathbb{R}^p \mid \mathbf{w} \cdot \mathbf{x} + b = 0\}$, where $\mathbf{w} \in \mathbb{R}^p$ is a **weight vector** and b is a **scalar intercept** (bias term).

The above definition uses the **vector dot product**, which is defined as:

$$\mathbf{w} \cdot \mathbf{x} = \sum_{j=1}^{p} w_j x_j.$$

(This provides a compact way to represent a weighted sum of features.)

- A hyperplane is parameterized by $\mathbf{w}$ and b
- $\mathbf{w} \cdot \mathbf{x}$ represents a linear combination of the features
- b represents an offset (just like w_0 from earlier)

Classification by Hyperplane

A hyperplane divides $\mathbb{R}^p$ into two half-spaces:

$$\{\mathbf{x} \in \mathbb{R}^p \mid \mathbf{w} \cdot \mathbf{x} + b \geq 0\} \quad \text{and} \quad \{\mathbf{x} \in \mathbb{R}^p \mid \mathbf{w} \cdot \mathbf{x} + b < 0\}$$

This leads to a natural **classifier** based on the hyperplane parameters $\mathbf{w}$ and b:

$$h_{\mathbf{w},b}(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b \geq 0 \\ 0 & \text{if } \mathbf{w} \cdot \mathbf{x} + b < 0 \end{cases}$$

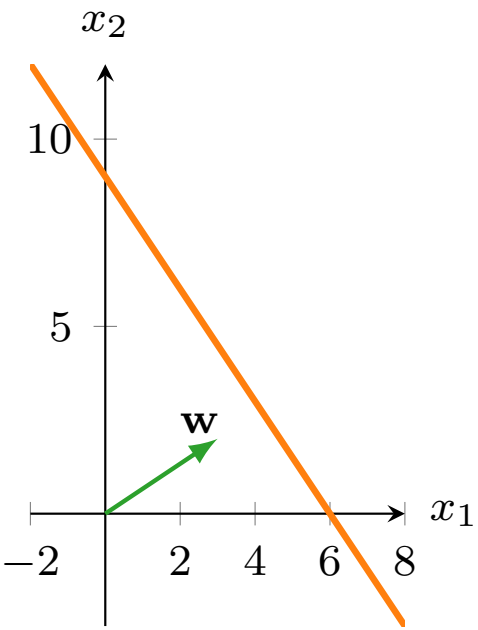
Definition (revisited)

A data set $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$ is **linearly separable** if there exist hyperplane parameters $\mathbf{w}$ and b such that $h_{\mathbf{w},b}(\mathbf{x}_i) = y_i$ for all $i \in \{1, 2, \ldots, n\}$.

Geometry of Hyperplanes

In $\mathbb{R}^2$, consider the hyperplane (line) given by $\mathbf{w} = (3, 2)$ and $b = -18$. The points on the hyperplane satisfy $w_1 x_1 + w_2 x_2 + b = 0$.

Let's plot!



Find x_1 intercept when $x_2 = 0$:

- $3x_1 + -18 = 0 \quad \Rightarrow \quad x_1 = 6$

Find x_2 intercept when $x_1 = 0$:

- $2x_2 + -18 = 0 \quad \Rightarrow \quad x_2 = 9$

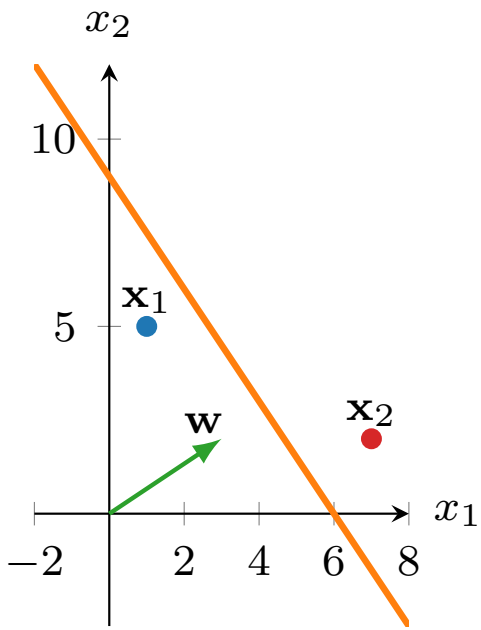
The parameter $\mathbf{w}$ defines a vector that is **orthogonal** (perpendicular) to the hyperplane.

- Changing $\mathbf{w}$ changes the orientation of the hyperplane
- The parameter b controls the offset of the hyperplane from the origin

Geometry of Hyperplanes (continued)

In $\mathbb{R}^2$, consider the hyperplane (line) given by $\mathbf{w} = (3, 2)$ and $b = -18$. Let's look at how various feature vectors would be classified.

The plot:



For feature vector $\mathbf{x}_1 = (1, 5)$, we have

$$\begin{aligned} w_1 x_{1,1} + w_2 x_{1,2} + b &= 3 \cdot 1 + 2 \cdot 5 + -18 \\ &= -5 < 0, \end{aligned}$$

so $h_{\mathbf{w},b}(\mathbf{x}_1) = 0$.

For feature vector $\mathbf{x}_2 = (7, 2)$, we have

$$\begin{aligned} w_1 x_{2,1} + w_2 x_{2,2} + b &= 3 \cdot 7 + 2 \cdot 2 + -18 \\ &= 7 \geq 0, \end{aligned}$$

so $h_{\mathbf{w},b}(\mathbf{x}_2) = 1$.

Lecture 04-3b:
Finding a Separating Hyperplane:
The Perceptron Algorithm

Perceptron Learning

The **perceptron algorithm** attempts to find a separating hyperplane for a given data set $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$ by iteratively updating the hyperplane parameters using the **perceptron learning rule**:

1. Start with $t = 0$ and initial weights $\mathbf{w}^{(0)}$ and $b^{(0)}$ (e.g., with each parameter drawn from $[-1, 1]$ uniformly)
2. Choose an input $\mathbf{x}_i$ that is **incorrectly classified**
3. Update parameters for iteration $t + 1$ using

$$b^{(t+1)} \leftarrow b^{(t)} + \alpha \left(y_i - h_{\mathbf{w}^{(t)}, b^{(t)}}(\mathbf{x}_i) \right)$$

$$w_k^{(t+1)} \leftarrow w_k^{(t)} + \alpha \left(y_i - h_{\mathbf{w}^{(t)}, b^{(t)}}(\mathbf{x}_i) \right) \cdot x_{ik} \quad \forall k \in \{1, 2, \ldots, p\}$$

4. Increment t and repeat Steps 2 and 3 until **no classification errors remain**

Understanding Weight Updates

The **perceptron learning rule** uses an **incorrectly classified** input $\mathbf{x}_i$ to update weights:

$$w_k^{(t+1)} \leftarrow w_k^{(t)} + \alpha \left(y_i - h_{\mathbf{w}^{(t)}, b^{(t)}}(\mathbf{x}_i) \right) \cdot x_{ik}$$

For convenience, we'll use $h^{(t)}$ to denote $h_{\mathbf{w}^{(t)}, b^{(t)}}$.

Understanding the update rule:

- If correct output should be **below** the boundary $(y_i = 0)$ but current weights placed it **above** the boundary $(h^{(t)}(\mathbf{x}_i) = 1)$, then **subtract** attribute value x_{ik} (scaled by α) from weight w_k
- If correct output should be **above** the boundary $(y_i = 1)$ but current weights placed it **below** the boundary $(h^{(t)}(\mathbf{x}_i) = 0)$, then **add** attribute value x_{ik} (scaled by α) to weight w_k

Perceptron Updates

The perceptron update rule shifts the weight vector **positively** or **negatively**, trying to get all data on the correct side of the hyperplane:

$$w_k^{(t+1)} \leftarrow w_k^{(t)} + \alpha \left(y_i - h^{(t)}(\mathbf{x}_i) \right) \cdot x_{ik}$$

Example: At iteration t, we have $\mathbf{w}^{(t)} = (-2.5, 0.6)$, $b^{(t)} = 0.2$, and we examine $\mathbf{x}_i = (0.5, 0.4)$ with $y_i = 1$:

$$\mathbf{w}^{(t)} \cdot \mathbf{x}_i + b^{(t)} = (-2.5 \cdot 0.5) + (0.6 \cdot 0.4) + 0.2 = -0.81,$$

$$\text{and } -0.81 < 0, \text{ so } h^{(t)}(\mathbf{x}_i) = 0.$$

This means $\mathbf{x}_i$ is **misclassified**. With $\alpha = 1$, the parameters get updated using:

$$b^{(t+1)} \leftarrow \left(b^{(t)} + (1 - 0) \right) = 0.2 + 1 = 1.2$$

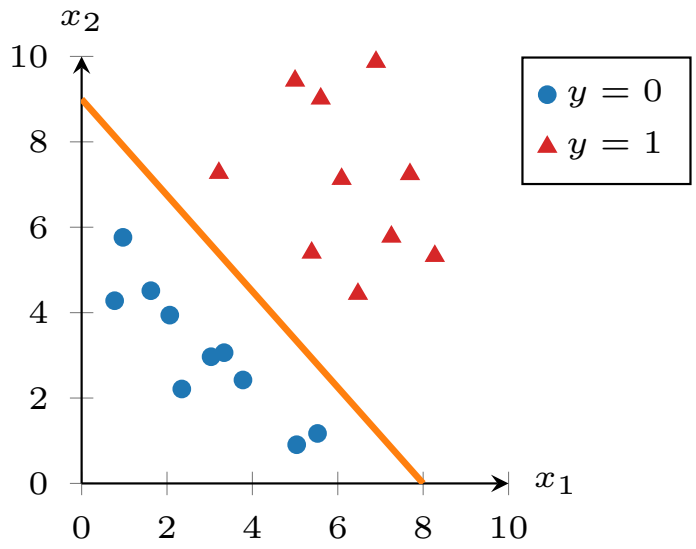
$$w_1^{(t+1)} \leftarrow \left(w_1^{(t)} + (1 - 0) \cdot x_{i,1} \right) = (-2.5 + 1 \cdot 0.5) = -2.0$$

$$w_2^{(t+1)} \leftarrow \left(w_2^{(t)} + (1 - 0) \cdot x_{i,2} \right) = (0.6 + 1 \cdot 0.4) = 1.0$$

Then $\mathbf{w}^{(t+1)} \cdot \mathbf{x}_i + b^{(t)} = (-2.0 \cdot 0.5) + (1.0 \cdot 0.4) + 1.2 = 0.6$, making $h^{(t+1)}(\mathbf{x}_i) = 1$.

Perceptron Termination and Linear Separability

The **perceptron algorithm** stops when no data point is misclassified.



For a **linearly separable** data set, the perceptron algorithm is **guaranteed to converge**, meaning that it will find a separating hyperplane (one of potentially many).

However, not all data sets are linearly separable...

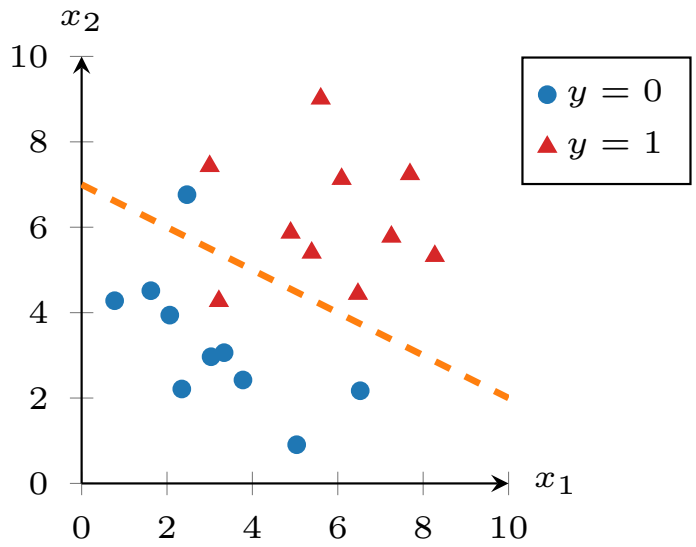
Definition

A data set $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$ is **linearly separable** if there exist hyperplane parameters $\mathbf{w}$ and b such that $h_{\mathbf{w},b}(\mathbf{x}_i) = y_i$ for all $i \in \{1, 2, \ldots, n\}$.

Lecture 04-3c:
Linearly Inseparable Data

Linearly Inseparable Data

The **perceptron algorithm** stops when no data point is misclassified.



For a **linearly inseparable** data set, the perceptron algorithm will **never terminate**!

- No matter how we adjust the weights, some points will **always** be classified incorrectly

We'll need to make a few modifications to the basic **perceptron algorithm** in order to handle the possibility of linearly inseparable data.

Modifying Perceptron Learning

To handle linearly inseparable data, we modify the perceptron algorithm:

1. Start with $t = 0$ and initial weights $\mathbf{w}^{(0)}$ and $b^{(0)}$
2. Choose an input $\mathbf{x}_i$ that is **incorrectly classified**
3. Update parameters for iteration $t + 1$ using

$$b^{(t+1)} \leftarrow b^{(t)} + \alpha \left(y_i - h^{(t)}(\mathbf{x}_i) \right)$$

$$w_k^{(t+1)} \leftarrow w_k^{(t)} + \alpha \left(y_i - h^{(t)}(\mathbf{x}_i) \right) \cdot x_{ik} \quad \forall k \in \{1, 2, \ldots, p\}$$

4. Increment t and repeat Steps 2 and 3 until ~~no classification errors remain~~ weights no longer change

To ensure that the weights eventually stop changing, we decrease α over time. As $\alpha \to 0$, the weights will stop changing, and the algorithm will stop. To get to a **least-error** hyperplane, this decrease needs to happen **slowly**; e.g., $\alpha^{(t)} \leftarrow \frac{1000}{1000 + t}$.

Modifying Perceptron Learning

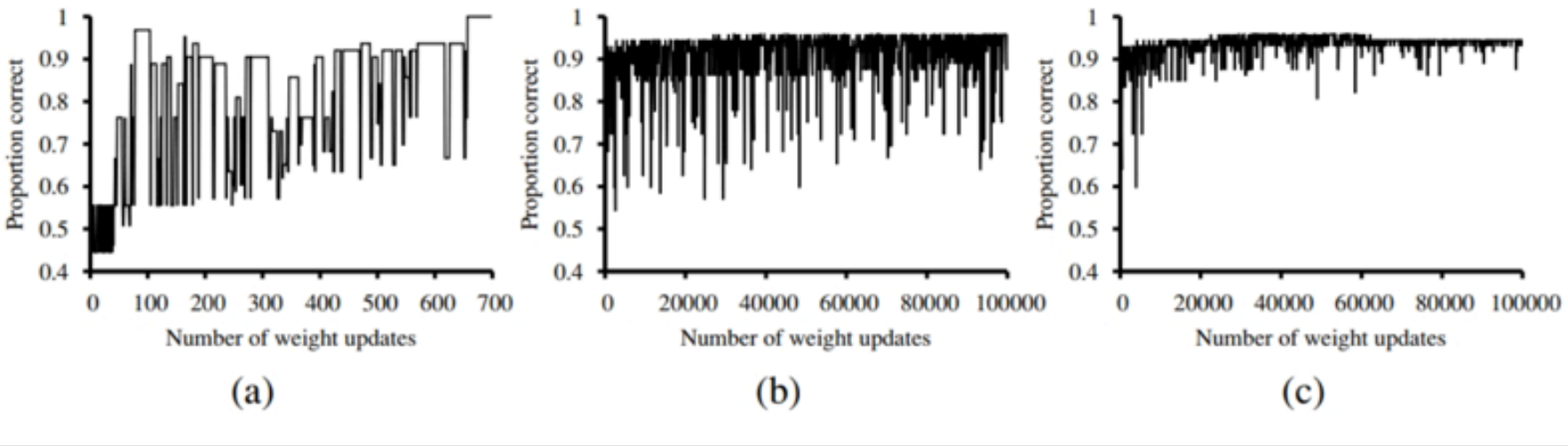


Figure 18.16 (a) Plot of total training-set accuracy vs. number of iterations through the training set for the perceptron learning rule, given the earthquake/explosion data in Figure 18.15(a). (b) The same plot for the noisy, non-separable data in Figure 18.15(b); note the change in scale of the x-axis. (c) The same plot as in (b), with a learning rate schedule $\alpha(t) = 1000/(1000 + t)$.

Lecture 04-3d:
Perceptron Loss and Cost Functions

Deriving Perceptron Weight Updates

The perceptron learning rule uses an **incorrectly classified** example $\mathbf{x}_i$ to update each weight w_k as

$$w_k^{(t+1)} \leftarrow w_k^{(t)} + \alpha \left(y_i - h^{(t)}(\mathbf{x}_i) \right) \cdot x_{ik}.$$

Natural question: Where does this come from?

We've seen something like this before! With some minor reformulation, the weight update rule for **stochastic gradient descent** for multiple linear regression is

$$w_k^{(t+1)} \leftarrow w_k^{(t)} + \alpha \left(y_i - h^{(t)}(\mathbf{x}_i) \right) \cdot 2x_{ik},$$

where $\mathbf{x}_i$ is the example that is used to approximate the gradient at iteration t.

Deriving Perceptron Updates (continued)

So it looks like the perceptron algorithm is doing an SGD-like update for the weights. We'll investigate, but first some observations:

1. The perceptron update rule

$$w_k^{(t+1)} \leftarrow w_k^{(t)} + \alpha \left(y_i - h^{(t)}(\mathbf{x}_i) \right) \cdot x_{ik}$$

   can also be applied to a **correctly classified** example $\mathbf{x}_i$, with the result being no change in weight because $y_i = h^{(t)}(\mathbf{x}_i)$.

2. The bias term b, which has update rule given by

$$b^{(t+1)} \leftarrow b^{(t)} + \alpha \left(y_i - h^{(t)}(\mathbf{x}_i) \right),$$

   can be treated like the other weights by **augmenting** each feature vector $\mathbf{x}_i$ with a **zeroth** attribute x_{i0} having value 1.

So we'll focus on the update process for a generic weight w_k using a training example $\mathbf{x}_i$ (which may or may not be correctly classified).

Deriving Perceptron Updates (continued)

Perceptron weight update:

$$w_k^{(t+1)} \leftarrow w_k^{(t)} + \alpha \left(y_i - h^{(t)}(\mathbf{x}_i) \right) \cdot x_{ik}$$

Generic gradient descent update:

$$w_k^{(t+1)} \leftarrow w_k^{(t)} - \alpha \frac{\partial}{\partial w_k} J(\mathbf{w})$$

Let's assume that the **perceptron algorithm** is using some cost function $J(\mathbf{w}, b)$ and following a (stochastic) gradient descent-like update using training example $\mathbf{x}_i$. Then we should have

$$\frac{\partial}{\partial w_k} J(\mathbf{w}, b) = -x_{ik} \cdot (y_i - h(\mathbf{x}_i))$$

which means that the cost function $J(\mathbf{w}, b)$ needs to include a term of the form

$$[-x_{ik} \cdot (y_i - h(\mathbf{x}_i))] \cdot w_k$$

for each weight w_k.

Deriving the Perceptron Cost Function

Using augmented features $x_{i0} = 1$, we can put everything together and rewrite the (SGD) cost function as:

$$\begin{aligned} J(\mathbf{w}, b) &= \sum_{j=0}^{p} \left(-x_{ij} \cdot (y_i - h(\mathbf{x}_i))\right) \cdot w_j \\ &= \sum_{j=0}^{p} -w_j \cdot x_{ij} \cdot (y_i - h(\mathbf{x}_i)) \\ &= -\left(\mathbf{w} \cdot \mathbf{x}_i + b\right) \cdot (y_i - h(\mathbf{x}_i)) \end{aligned}$$

In a full-batch gradient descent update using **all** training examples, the cost function would be

$$J(\mathbf{w}, b) = \frac{1}{n} \sum_{i=1}^{n} -\left(\mathbf{w} \cdot \mathbf{x}_i + b\right) \cdot (y_i - h(\mathbf{x}_i)) \,.$$

Understanding the Perceptron Cost Function

After some work, we see that the **perceptron algorithm** is using **stochastic gradient descent** to minimize the cost function

$$J(\mathbf{w}, b) = \frac{1}{n} \sum_{i=1}^{n} - (\mathbf{w} \cdot \mathbf{x}_i + b) \cdot (y_i - h_{\mathbf{w},b}(\mathbf{x}_i)) \, .$$

(Here we add subscripts $\mathbf{w}$ and b back to h to emphasize dependence; also, recall that $h_{\mathbf{w},b}(\mathbf{x}) = 1$ if $\mathbf{w} \cdot \mathbf{x} + b \geq 0$, and 0 otherwise.)

So for any training example $\mathbf{x}_i$:

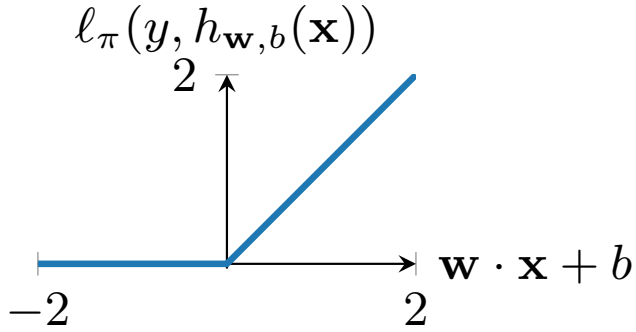
- If $y_i = h_{\mathbf{w},b}(\mathbf{x}_i)$, then $\mathbf{x}_i$ is **correctly classified** and contributes nothing to the cost
- If $y_i \neq h_{\mathbf{w},b}(\mathbf{x}_i)$, then $\mathbf{x}_i$ is **incorrectly classified**;
  - If $y_i = 0$, then $\mathbf{x}_i$ contributes $(\mathbf{w} \cdot \mathbf{x}_i + b)$ to the cost (with scaling)
  - If $y_i = 1$, then $\mathbf{x}_i$ contributes $-(\mathbf{w} \cdot \mathbf{x}_i + b)$ to the cost (with scaling)

The Perceptron Loss Function

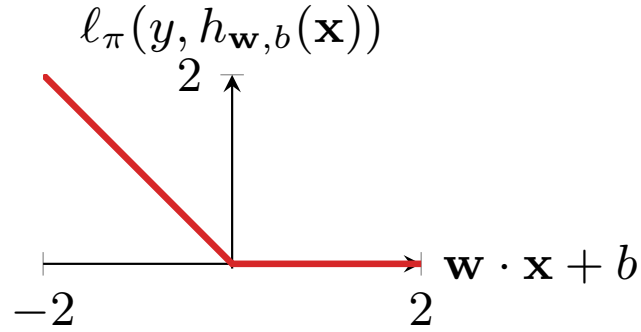
Interpreting the perceptron cost function $J(\mathbf{w}, b)$ as the average loss over the training set gives us the **perceptron loss function**

$$\begin{aligned}\ell_\pi\left(y, h_{\mathbf{w},b}(\mathbf{x})\right) &= -\left(\mathbf{w}\cdot\mathbf{x}+b\right)\cdot\left(y - h_{\mathbf{w},b}(\mathbf{x})\right)\\ &= \max\left\{0, (1-2y)\cdot(\mathbf{w}\cdot\mathbf{x}+b)\right\}.\end{aligned}$$

With $y = 0$, loss is $\max\{0, \mathbf{w}\cdot\mathbf{x}+b\}$



With $y = 1$, loss is $\max\{0, -(\mathbf{w}\cdot\mathbf{x}+b)\}$



Perceptron loss is also called **hinge loss** after the shape of the loss function. The further away an incorrectly classified point is from the boundary, the larger the loss.

The Perceptron Algorithm as a Supervised Learning System

We can view the **perceptron algorithm** as a supervised learning system:

- Data: $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$, with p input attributes
- Hypothesis space:

$$\mathcal{H} = \{h : \mathbb{R}^p \to \{0, 1\} \mid h_{\mathbf{w},b}(\mathbf{x}) = \text{Threshold}\,(\mathbf{w} \cdot \mathbf{x} + b)\}$$

  where $\text{Threshold}(t) = 1$ if $t \geq 0$ and 0 otherwise
- Learning algorithm:
  - Loss function: Hinge loss
  - Cost function: Average loss
  - Fitting method: Stochastic gradient descent

The History of the Perceptron

Frank Rosenblatt
1928–1969

Rosenblatt's perceptron played an important role in the history of machine learning. Initially, Rosenblatt simulated the perceptron on an IBM 704 computer at Cornell in 1957, but by the early 1960s he had built special-purpose hardware that provided a direct, parallel implementation of perceptron learning. Many of his ideas were encapsulated in "Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms" published in 1962. Rosenblatt's work was criticized by Marvin Minksy, whose objections were published in the book "Perceptrons", co-authored with Seymour Papert. This book was widely misinterpreted at the time as showing that neural networks were fatally flawed and could only learn solutions for linearly separable problems. In fact, it only proved such limitations in the case of single-layer networks such as the perceptron and merely conjectured (incorrectly) that they applied to more general network models. Unfortunately, however, this book contributed to the substantial decline in research funding for neural computing, a situation that was not reversed until the mid-1980s. Today, there are many hundreds, if not thousands, of applications of neural networks in widespread use, with examples in areas such as handwriting recognition and information retrieval being used routinely by millions of people.

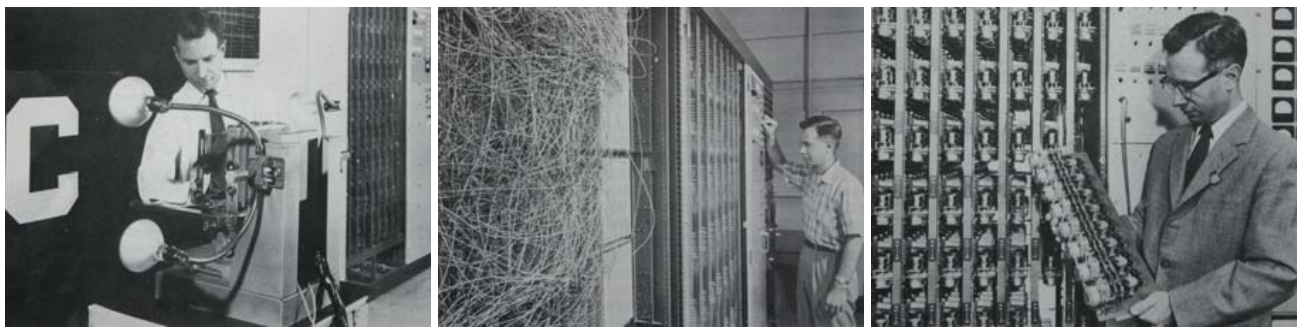


Figure 4.8 Illustration of the Mark 1 perceptron hardware. The photograph on the left shows how the inputs were obtained using a simple camera system in which an input scene, in this case a printed character, was illuminated by powerful lights, and an image focussed onto a 20×20 array of cadmium sulphide photocells, giving a primitive 400 pixel image. The perceptron also had a patch board, shown in the middle photograph, which allowed different configurations of input features to be tried. Often these were wired up at random to demonstrate the ability of the perceptron to learn without the need for precise wiring, in contrast to a modern digital computer. The photograph on the right shows one of the racks of adaptive weights. Each weight was implemented using a rotary variable resistor, also called a potentiometer, driven by an electric motor thereby allowing the value of the weight to be adjusted automatically by the learning algorithm.

From: C. Bishop, *Pattern Recognition and Machine Learning*. Springer (2006).

Lecture 04-4a:
Evaluating Classifiers

Model Evaluation

In a **regression** problem, we typically use **mean squared error** (MSE) to evaluate the **quality** of a hypothesis function h on a data set S:

$$\text{MSE}(h, S) = \frac{1}{|S|} \sum_{(\mathbf{x},y)\in S} (y - h(\mathbf{x}))^2 .$$

- During **learning** (model fitting), we find a hypothesis h that minimizes the MSE on the **training set**
- During **model selection**, we evaluate h's ability to generalize by computing its MSE on the **validation set**
  (or we use cross-validation instead)

Other **metrics** (methods of scoring) are available for regression, but MSE is the most common.

Question: How do we evaluate the **quality** of a **classifier**?

Evaluating a Classifier

In **logistic regression**, we used average **log-loss** (with regularization) for the cost function during **training**:

$$\frac{1}{n}\sum_{i=1}^{n} -\left(y_i \log\left(h(\mathbf{x}_i)\right) + (1-y_i)\log\left(1-h(\mathbf{x}_i)\right)\right).$$

Log-loss works well for evaluating hypothesis functions that return **probabilities** in the range $[0, 1]$.

However, log-loss is **not useful** for evaluating a **classifier** which predicts **labels** in $\{0, 1\}$ because a single **wrong** prediction results in an **infinite cost**.

Instead, we'll need some other metrics to evaluate classifiers.

Classifier Performance Metrics

Definition

Two metrics for evaluating a classifier are:

$$\textbf{accuracy} = \frac{\#\text{ of points classified } \textbf{correctly}}{\#\text{ of points total}}$$

$$\textbf{error} = \frac{\#\text{ of points classified } \textbf{incorrectly}}{\#\text{ of points total}} = 1 - \text{accuracy}$$

Accuracy provides a very **simple** approach to measuring performance, but it doesn't provide as much insight into mistakes that might get made.

- The modified **perceptron algorithm** searches for a **least-error** hyperplane (i.e., one that misclassifies the smallest number of points)

Misclassification Errors

		Truth (y)	
		1	0
Prediction $(\hat{y})$	1	**TP**: True positives	**FP**: False positives
	0	**FN**: False negatives	**TN**: True negatives

$$\textbf{accuracy} = \frac{\textbf{TP} + \textbf{TN}}{\textbf{TP} + \textbf{TN} + \textbf{FP} + \textbf{FN}} = \frac{\textbf{TP} + \textbf{TN}}{n}$$

$$\textbf{precision} = \frac{\textbf{TP}}{\textbf{TP} + \textbf{FP}}$$

(of points that were **predicted** positive, what proportion actually were positive?)

$$\textbf{recall} = \frac{\textbf{TP}}{\textbf{TP} + \textbf{FN}}$$

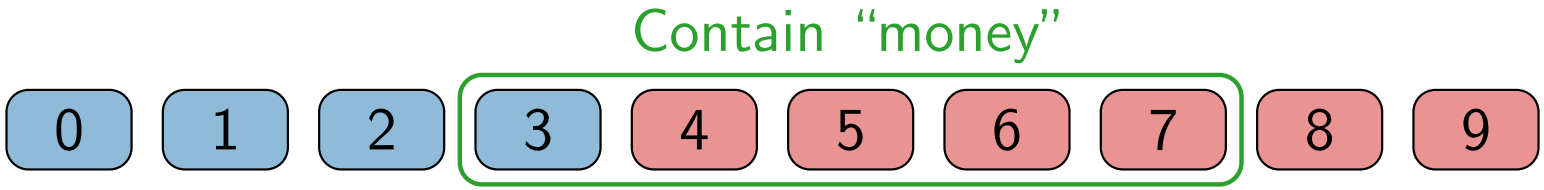
(of points that were **actually** positive, what proportion were predicted correctly?)

Confusion Matrices

Definition

A **confusion matrix** is a table containing counts of true positives, true negatives, false positives, and false negatives for a classifier.

Example: Consider the problem of classifying email as **spam** (1) or **ham** (0). Assume we have 10 emails, 6 of which are spam; additionally, 5 emails contain the word "money", only one of which is legitimate.



We have three classifiers:

- Classifier 1: nothing is spam
- Classifier 2: everything is spam
- Classifier 3: emails containing "money" are spam

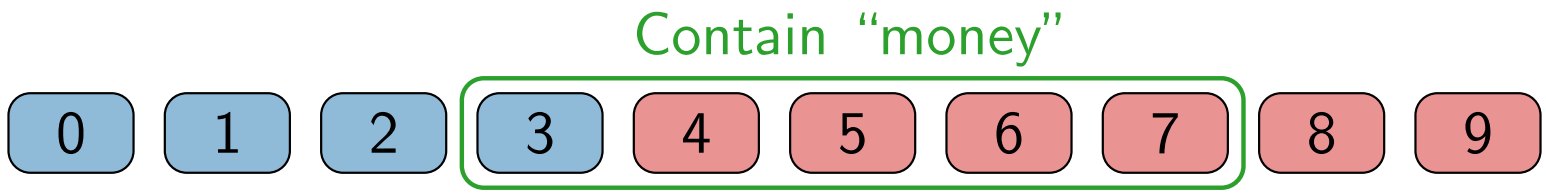
Example Continued: Confusion Matrices

Let's construct confusion matrices for each of the three classifiers.

Classifiers:

- Clf. 1: nothing is spam
- Clf. 2: everything is spam
- Clf. 3: emails containing "money" are spam

		y	
		1	0
$\hat{y}$	1	TP	FP
	0	FN	TN



Clf. 1		y	
		1	0
$\hat{y}$	1	0	0
	0	6	4

Clf. 2		y	
		1	0
$\hat{y}$	1	6	4
	0	0	0

Clf. 3		y	
		1	0
$\hat{y}$	1	4	1
	0	2	3

Example Continued: Precision & Recall

		Clf. 1	Clf. 2	Clf. 3
Accuracy	$\frac{\text{TP}+\text{TN}}{n}$	$\frac{0+4}{10} = 0.4$	$\frac{6+0}{10} = 0.6$	$\frac{4+3}{10} = 0.7$
Precision	$\frac{\text{TP}}{\text{TP}+\text{FP}}$	$\frac{0}{0+0} = \text{undef}$	$\frac{6}{6+4} = 0.6$	$\frac{4}{4+1} = 0.8$
Recall	$\frac{\text{TP}}{\text{TP}+\text{FN}}$	$\frac{0}{0+6} = 0$	$\frac{6}{6+0} = 1.00$	$\frac{4}{4+2} \approx 0.67$

Clf. 1		y	
		1	0
$\hat{y}$	1	0	0
	0	6	4

Clf. 2		y	
		1	0
$\hat{y}$	1	6	4
	0	0	0

Clf. 3		y	
		1	0
$\hat{y}$	1	4	1
	0	2	3

Lecture 04-4b:
(Optional) Evaluation using Curves

Precision & Recall for a Threshold Classifier

For a soft threshold classifier (e.g., logistic regression), **precision** and **recall** **depend** on the cutoff value t used for classification:

$$\hat{y} = \begin{cases} 1 & \text{if } h(\mathbf{x}) \geq t \\ 0 & \text{otherwise} \end{cases}$$

- If $t = 0$: Everything is spam — precision is **low**, but recall is **1.0**
- If $t = 1$: Nothing is spam — precision is **high**, but recall is **low**

Precision and recall are **inversely related**: **improving** one generally **decreases** the other.

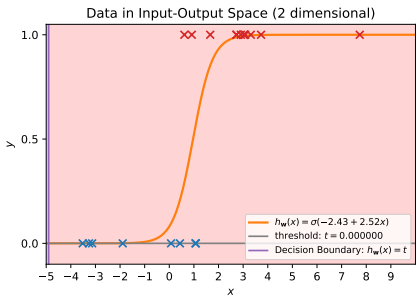
Precision penalizes **false positives**; **Recall** penalizes **false negatives**.

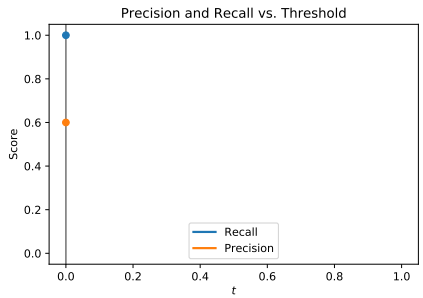
- Criminal trial: avoid false positives (want high precision)
- Disease screening: avoid false negatives (want high recall)

Precision & Recall vs. Threshold

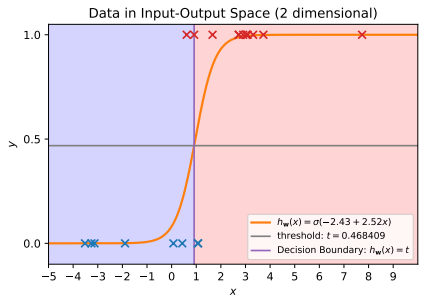
A soft threshold classifier is really a **family** of classifiers defined by the choice of cutoff value t. How does each classifier in the family do?

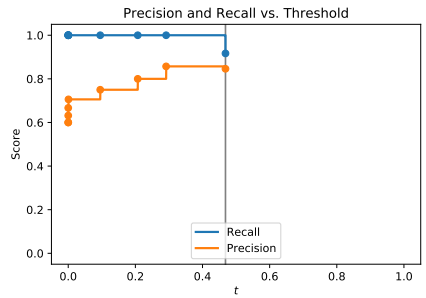
For $t = 0.0$, precision $= 0.6$, recall $= 1.0$:



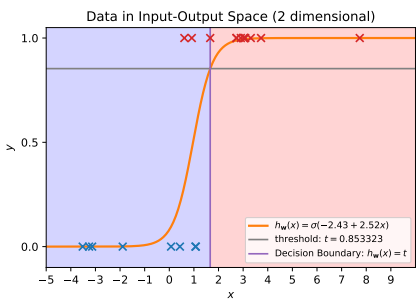


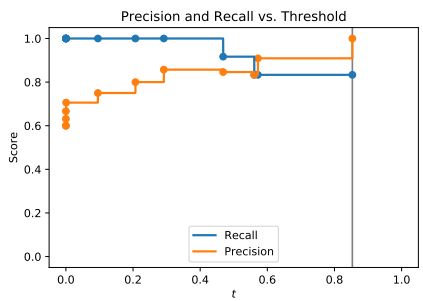
For $t \approx 0.468$, precision ≈ 0.85, recall ≈ 0.92:



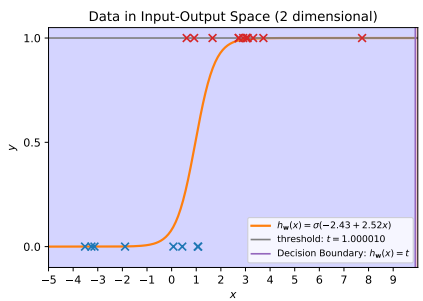


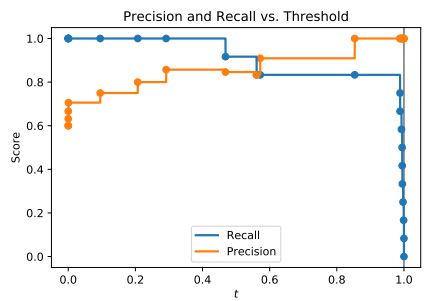
For $t \approx 0.853$, precision $= 1.0$, recall ≈ 0.83:





For $t > 1.0$, precision $=$ undef., recall $= 0.0$:

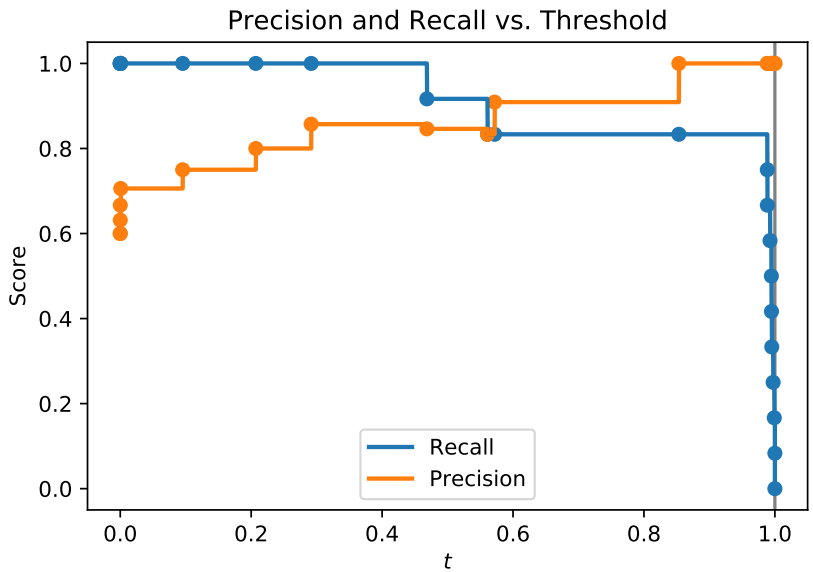


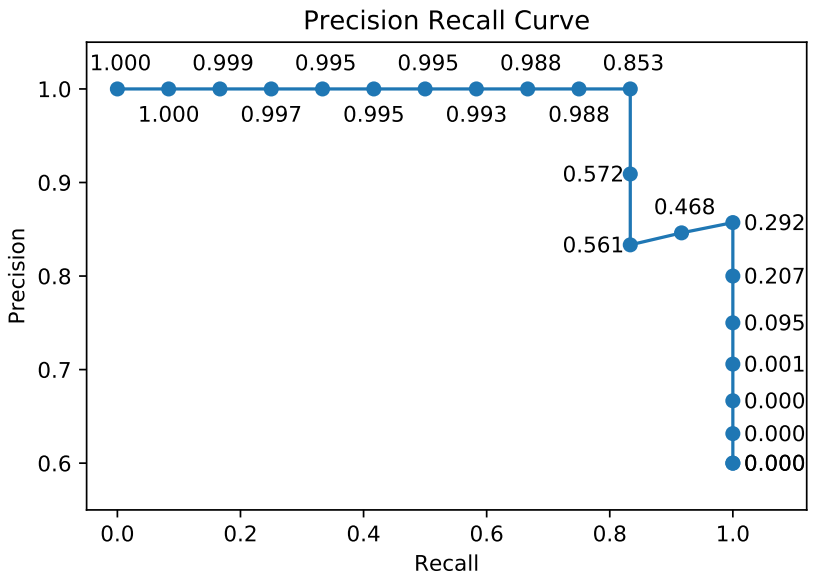


For each value of t, we compute the precision and recall of the resulting classifier. Then we plot the scores (shown on the right).

Precision Recall Curve

For any cutoff value t, we can also plot the associated (precision, recall) pairs as points in two-dimensional space:





- **Precision-recall curve** is **parameterized** by cutoff value t
- Area under the precision-recall curve gives us a metric for a (soft) classifier that is **independent** of cutoff value

Other Metrics for Evaluating Classifiers

The **F-score** (F1-score) balances precision and recall in a **single** number:

$$F = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

False Positive Rate (FPR):

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

True Positive Rate (TPR):

$$\textbf{TPR} = \frac{\textbf{TP}}{\textbf{TP} + \textbf{FN}} = \textbf{recall}$$

		y	
		1	0
$\hat{y}$	1	TP	FP
	0	FN	TN

ROC Curves

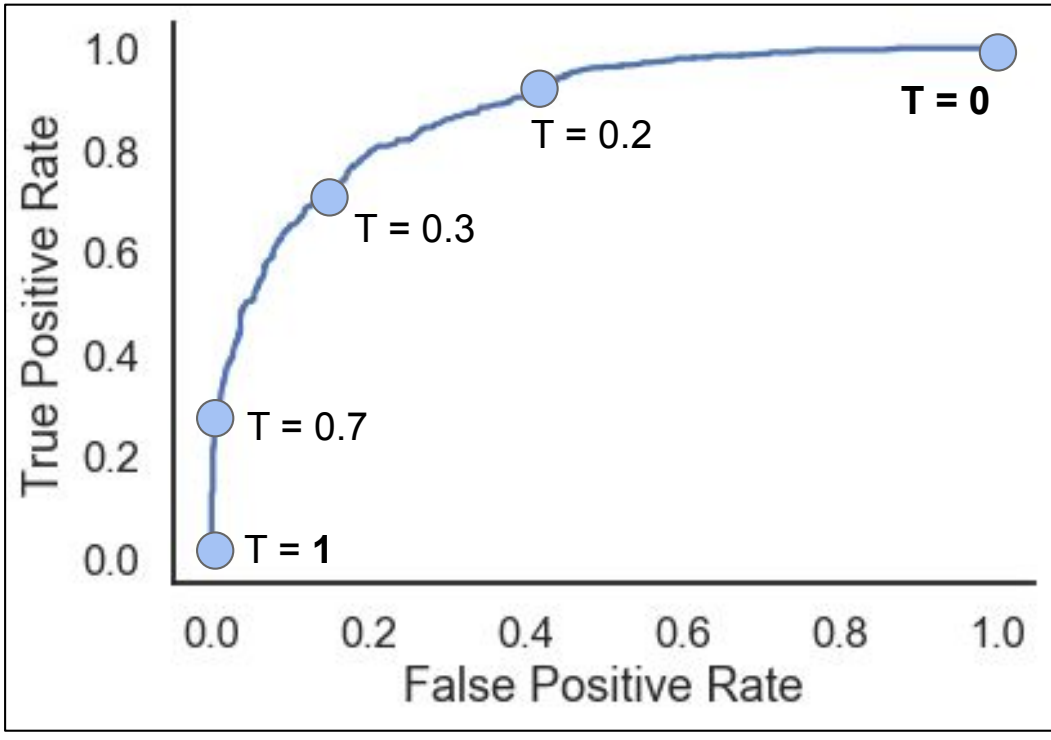
Decreasing the decision threshold:

- Increases TPR (good).
- Increases FPR (bad).

A “ROC curve” shows this tradeoff.

- X-axis: False positive rate.
- Y-axis: True positive rate.

What are the Ts in the bottom left and top right?



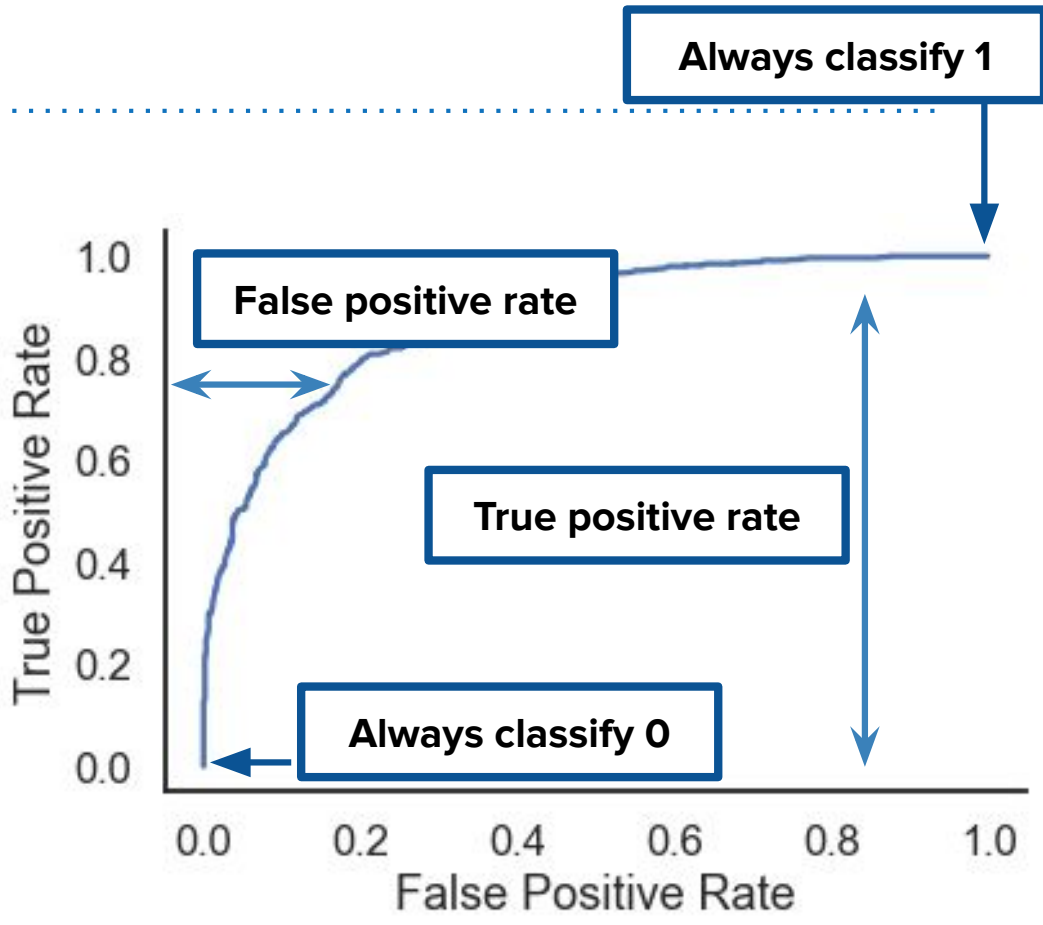
ROC Curves

Decreasing the decision threshold:

- Increases TPR (good).
- Increases FPR (bad).

A “ROC curve” shows this tradeoff.

- X-axis: False positive rate.
- Y-axis: True positive rate.

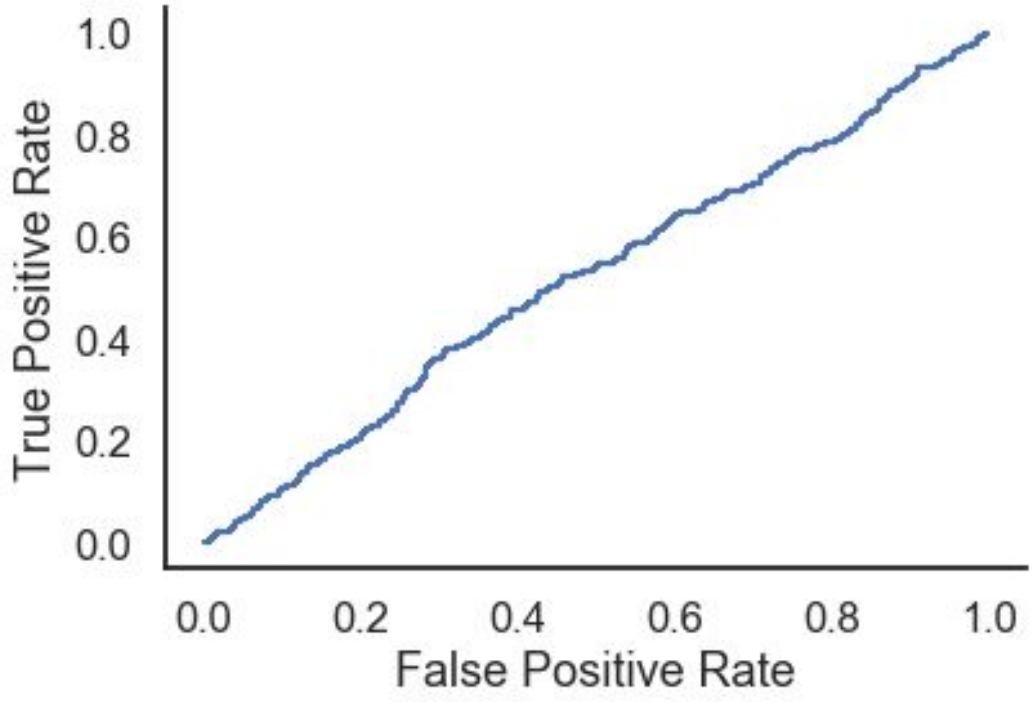


What Good Are ROC Curves?

ROC Curves provide a visual picture of the underlying quality of the regression model.

Example: Random Predictor.

- Assigns random probability between 0 and 1 to any prediction.
- Resulting ROC curve is a diagonal line.



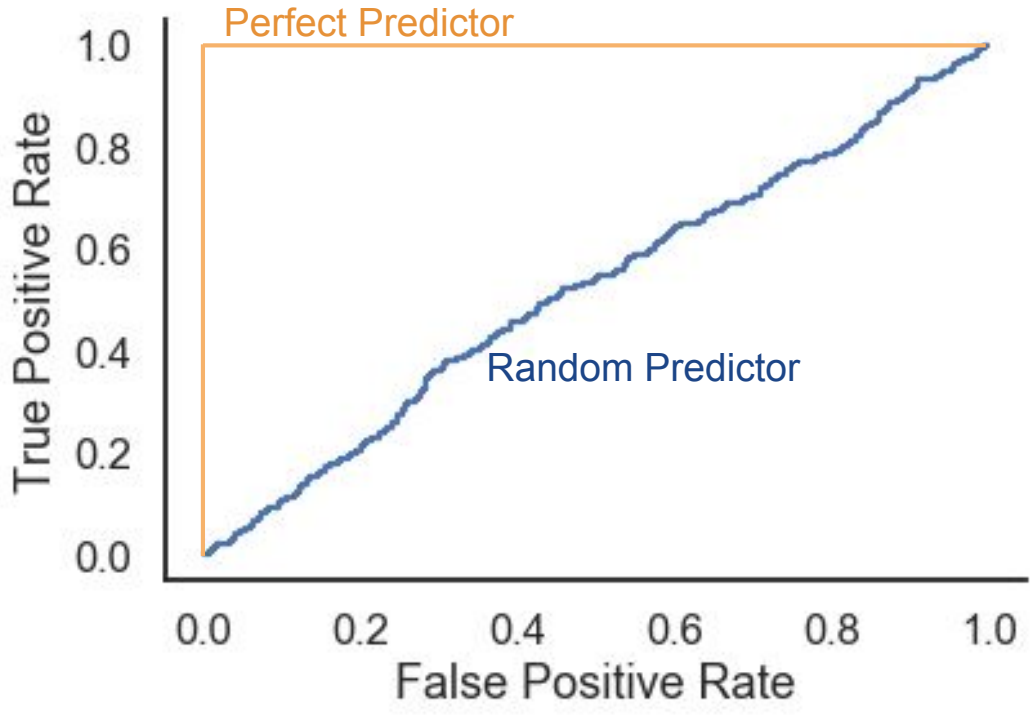
What Good Are ROC Curves?

ROC Curves provide a visual picture of the underlying quality of the regression model.

Example: Perfect Predictor.

- Gives probability 1 to class 1 and probability to class 0.
- Resulting ROC curve is a sharp curve.

The better a model, the more it will resemble the perfect predictor.



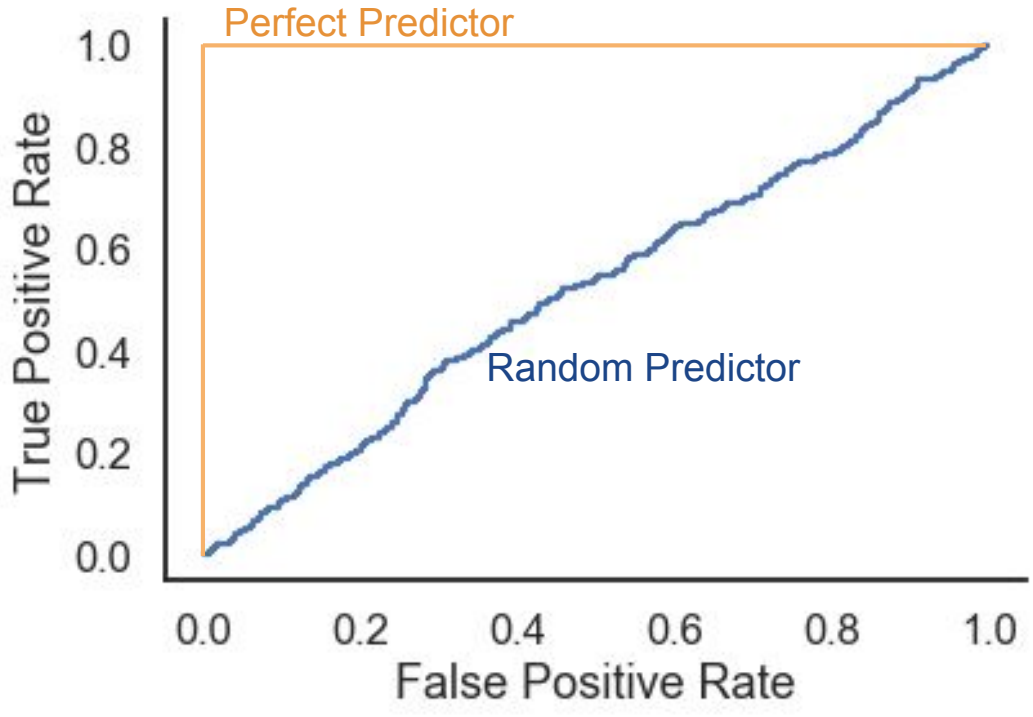
Area under ROC Curve

Can also compute the area under the ROC curve (a.k.a. **AUC**).

- Perfect Predictor: 1
- Random Predictor: 0.5
- Your predictor: Somewhere in [0.5, 1].

The **AUC** provides a dimensionless quantity that characterizes our underlying regression model.

- Dimensionless means it has no units (unlike MSE, mean cross entropy loss, etc.)



It is possible to build a predictor with AUC < 0.5, but that means it does worse than just randomly guessing.

Lecture 04-4c:
(Optional) Additional Notes on Classification

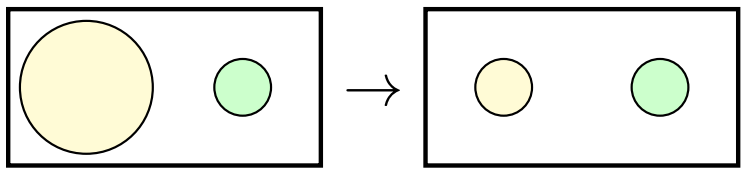
Unbalanced Classes

A significant issue in **classification** is having **unbalanced** training classes. Some examples:

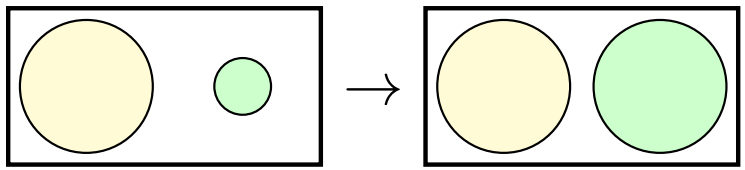
- Detecting fraud among legitimate transactions
- Detecting terrorists among general population

Some remedies:

- Subsample:



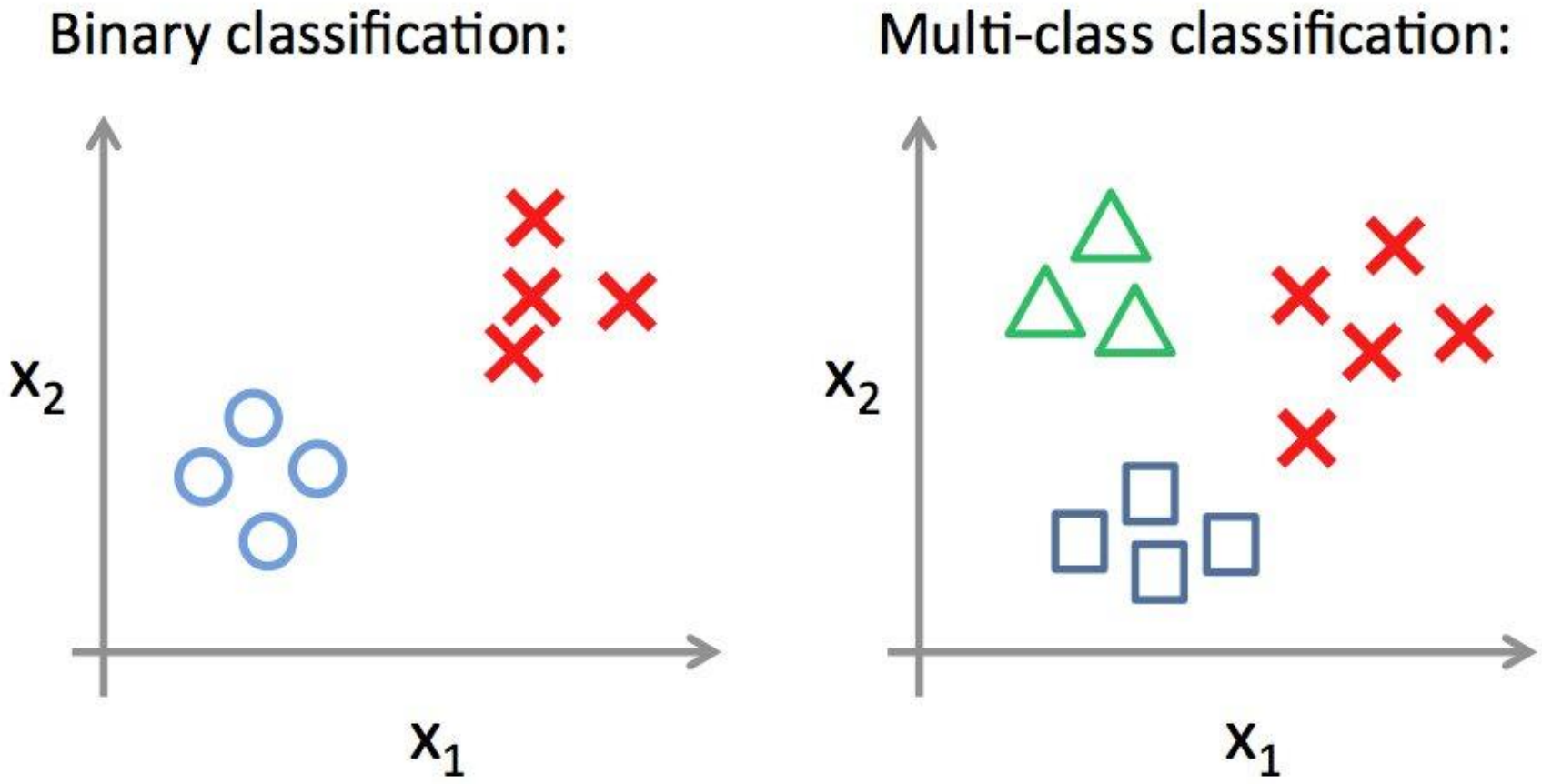
- Resample:



- Weighting: $\sum_{i=1}^{n} \ell\left(y_i, h(\mathbf{x}_i)\right) \rightarrow \sum_{i:y_i=1} c_1 \cdot \ell\left(y_i, h(\mathbf{x}_i)\right) + \sum_{i:y_i=0} c_0 \cdot \ell\left(y_i, h(\mathbf{x}_i)\right)$

Multi-Class Classification

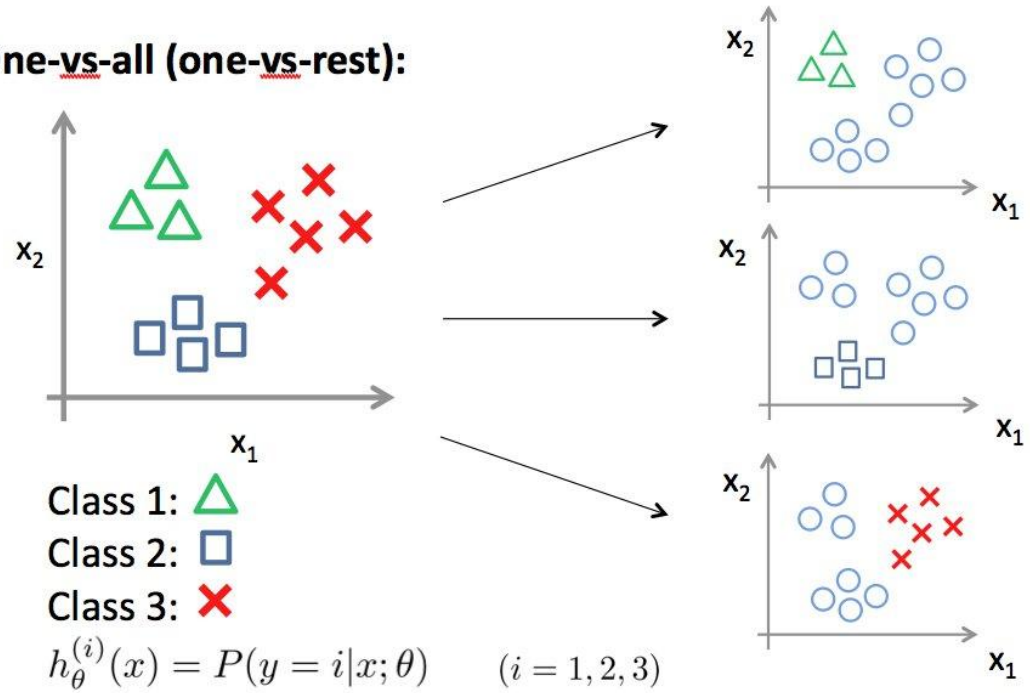
So far we have seen **classification** when the output variable is **binary**.
What about if y is discrete with **more than two** categories (classes)?



One Versus Rest Classifiers

For **multi-class classification** with K classes, we can build K binary classifiers, one for each class:

$$h_{\mathbf{w}^{(1)}}(\mathbf{x}) = P(Y = 1|\mathbf{X} = \mathbf{x})$$
$$h_{\mathbf{w}^{(2)}}(\mathbf{x}) = P(Y = 2|\mathbf{X} = \mathbf{x})$$
$$\vdots$$
$$h_{\mathbf{w}^{(K)}}(\mathbf{x}) = P(Y = K|\mathbf{X} = \mathbf{x})$$



We then classify points according to the highest-probability class:

$$\hat{y} = \arg\max_k \{h_{\mathbf{w}^{(k)}}(\mathbf{x})\}$$

Evaluating Multi-Class Classifiers

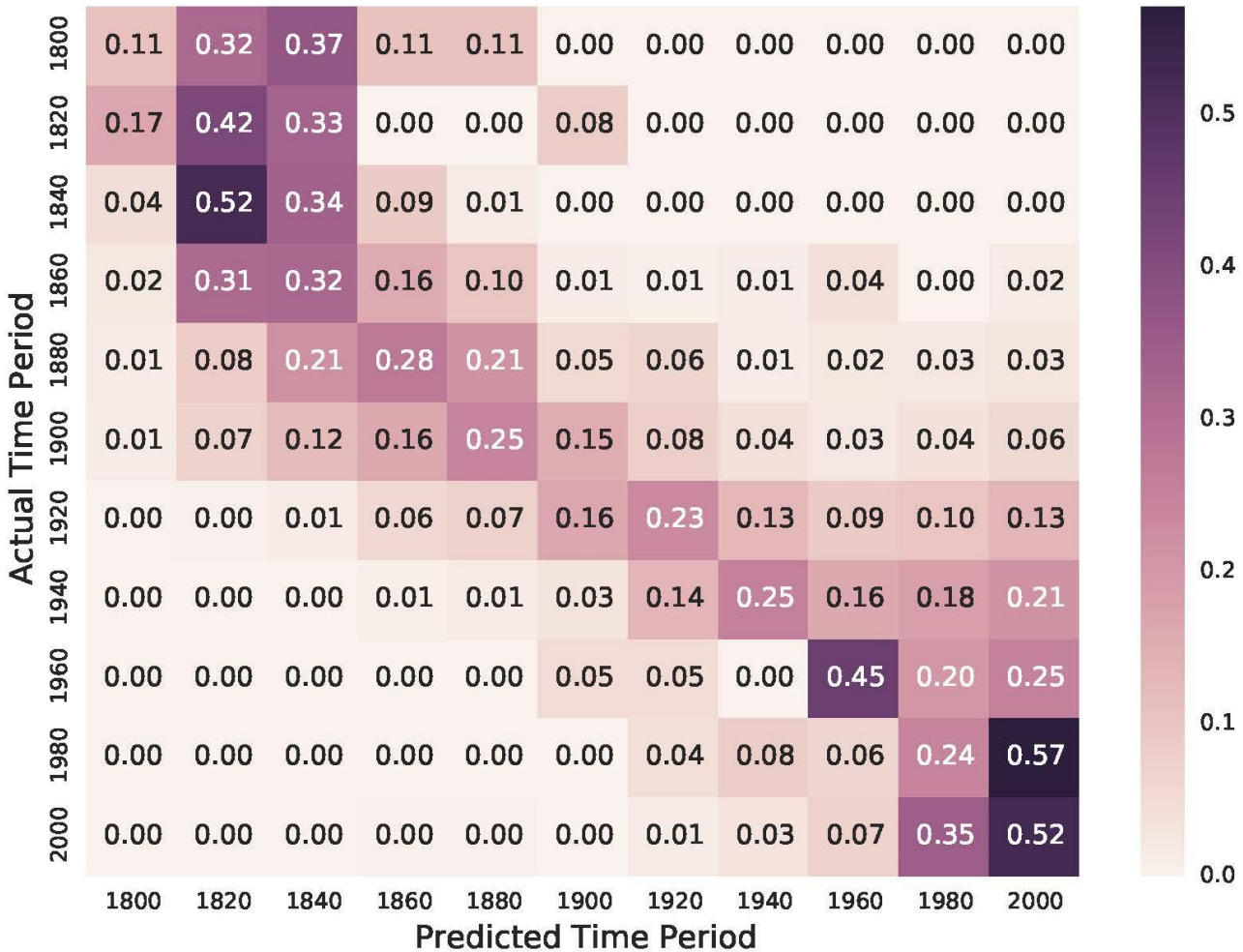
Confusion matrix for binary classification:

		Truth (y)	
		1	0
Prediction $(\hat{y})$	1	**TP**	**FP**
	0	**FN**	**TN**

Confusion matrix for multi-class classification:

		Truth (y)				
		1	2	3	$\ldots$	K
Prediction $(\hat{y})$	1	✓	X	X	$\ldots$	X
	2	X	✓	X	$\ldots$	X
	3	X	X	✓	$\ldots$	X
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
	K	X	X	X	$\ldots$	✓

Example: Classifying Documents by Time Period



- More classes leads to a harder problem!
- Scoring with hit/miss only is excessively harsh
- A top-k success rate gives credit if the correct label would have been one of the model's first k guesses