ITCS 5356: Intro to Machine Learning

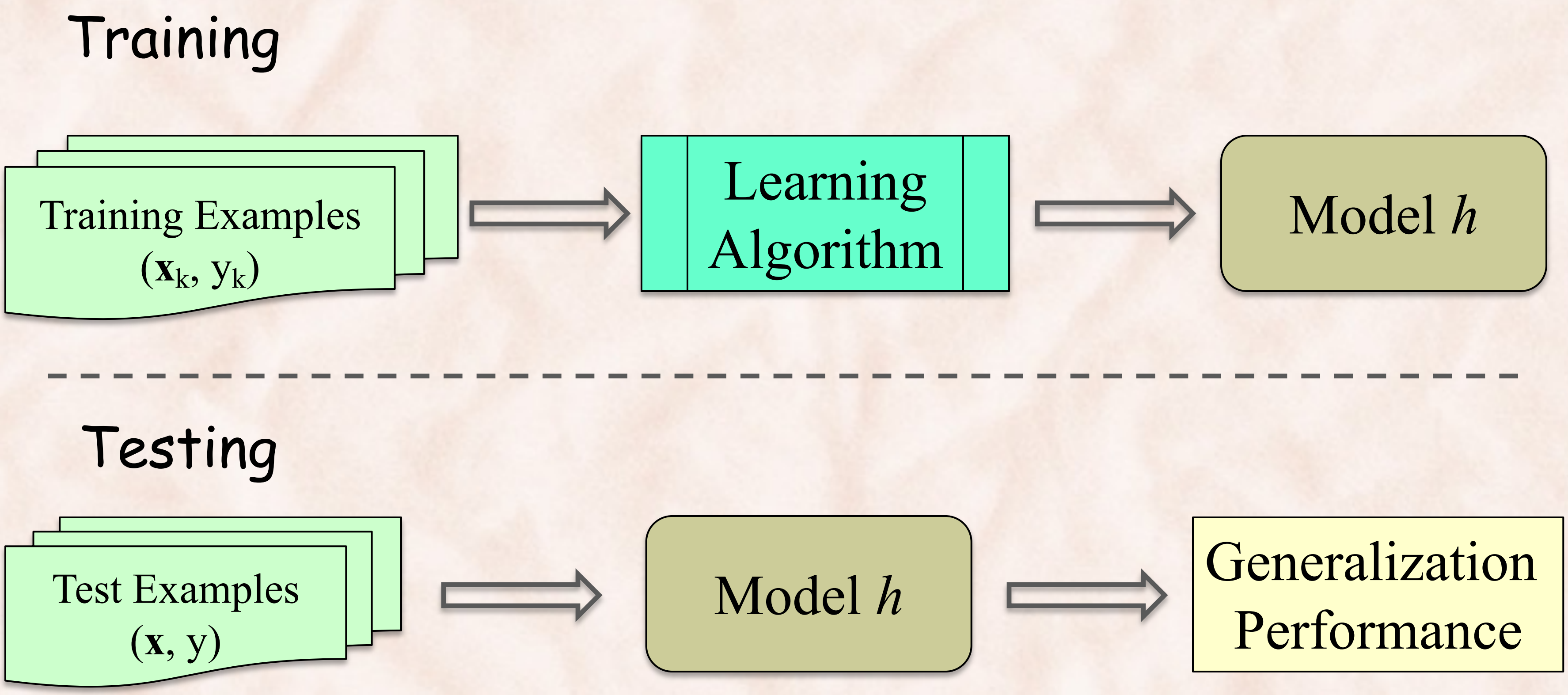
Logistic Regression

Razvan C. Bunescu

Department of Computer Science @ CCI

razvan.bunescu@charlotte.edu

Supervised Learning



Supervised Learning

- **Task** = learn an (unknown) function $f : X \rightarrow Y$ that maps input instances $\mathbf{x} \in X$ to output targets $y = f(\mathbf{x}) \in Y$:
  - **Classification**:
    - The output $y \in Y$ is one of a finite set of discrete categories.
  - **Regression**:
    - The output $y \in Y$ is continuous, or has a continuous component.

- Target function $f(\mathbf{x})$ is known (only) through (noisy) set of training examples:

$$(\mathbf{x}_1,y_1), (\mathbf{x}_2,y_2), \ldots (\mathbf{x}_n,y_n)$$

Parametric Approaches to Supervised Learning

- **Task** = build a function $h(\mathbf{x})$ such that:
  - h matches f well on the training data:
    => h is able to fit data that it has seen.
  - h also matches f well on test data:
    => h is able to generalize to unseen data.

- **Task** = choose h from a "nice" *class of functions* that depend on a vector of parameters $\mathbf{w}$:
  - $h(\mathbf{x}) \equiv h_{\mathbf{w}}(\mathbf{x}) \equiv h(\mathbf{w},\mathbf{x})$
  - **what classes of functions are "nice"?**
    - *linear* $\subset$ *convex* $\subset$ *continuous* $\subset$ *differentiable* $\subset$ ...

Three Parametric Approaches to Classification

1) Discriminant Functions: scoring function $f : X \rightarrow T$ that directly assigns a vector $\mathbf{x}$ to a specific class C_k.
   - Inference and decision combined into a single learning problem.
   - *Linear Discriminant*: the decision surface is a hyperplane in X:
     - Perceptron
     - Support Vector Machines
     - Fisher 's Linear Discriminant

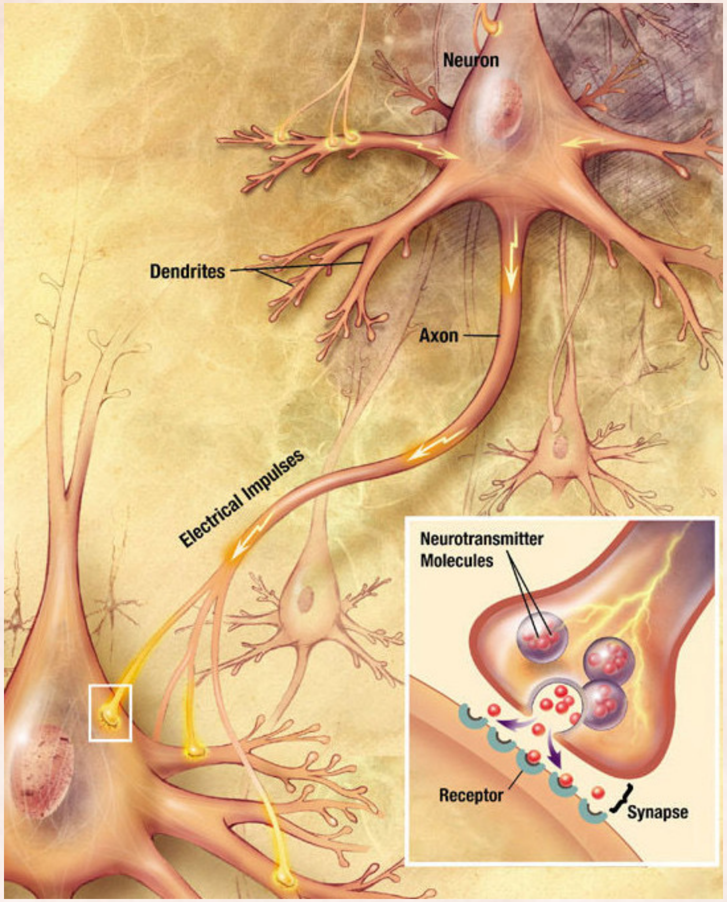
Three Parametric Approaches to Classification

2) Probabilistic Discriminative Models: directly model the posterior class probabilities $p(C_k \mid \mathbf{x})$.
   - Inference and decision are separate.
   - Less data needed to estimate $p(C_k \mid \mathbf{x})$ than $p(\mathbf{x} \mid C_k)$.
   - Can accommodate many overlapping features.
     - Logistic Regression
     - Conditional Random Fields

Three Parametric Approaches to Classification

3) Probabilistic Generative Models:
   - Model class-conditional $p(\mathbf{x} \mid C_k)$ as well as the priors $p(C_k)$, then use Bayes's theorem to find $p(C_k \mid \mathbf{x})$.
     - or model $p(\mathbf{x}, C_k)$ directly, then marginalize to obtain the posterior probabilities $p(C_k \mid \mathbf{x})$.
   - Inference and decision are separate.
   - Can use $p(\mathbf{x})$ for *outlier* or *novelty detection*.
   - Need to model dependencies between features.
     - Naïve Bayes.
     - Hidden Markov Models.

Neurons



Soma is the central part of the neuron:
- *where the input signals are combined.*

Dendrites are cellular extensions:
- *where majority of the input occurs.*

Axon is a fine, long projection:
- *carries nerve signals to other neurons.*

Synapses are molecular structures between axon terminals and other neurons:
- *where the communication takes place.*

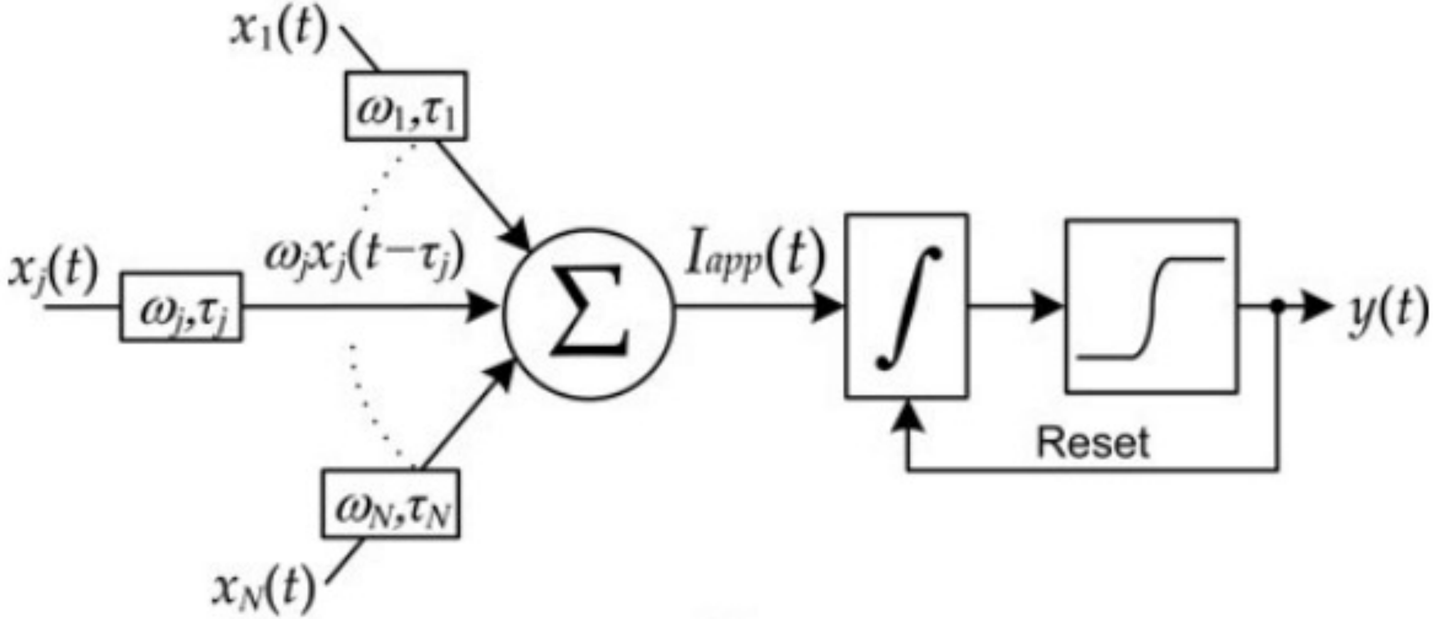
Neuron Models

https://www.research.ibm.com/software/IBMResearch/multimedia/IJCNN2013.neuron-model.pdf

Year	Model Name	Reference
1907	Integrate and fire	[13]
1943	McCulloch and Pitts	[11]
1952	Hodgkin-Huxley	[12]
1958	Perceptron	[14]
1961	Fitzhugh-Nagumo	[15]
1965	Leaky integrate-and-fire	[16]
1981	Morris-Lecar	[17]
1986	Quadratic integrate-and-fire	[18]
1989	Hindmarsh-Rose	[19]
1998	Time-varying integrate-and-fire model	[20]
1999	Wilson Polynomial	[21]
2000	Integrate-and-fire or burst	[22]
2001	Resonate-and-fire	[23]
2003	Izhikevich	[24]
2003	Exponential integrate-and-fire	[25]
2004	Generalized integrate-and-fire	[26]
2005	Adaptive exponential integrate-and-fire	[27]
2009	Mihalas-Neibur	[28]

Spiking/LIF Neuron Function

http://ee.princeton.edu/research/prucnal/sites/default/files/06497478.pdf



(b)

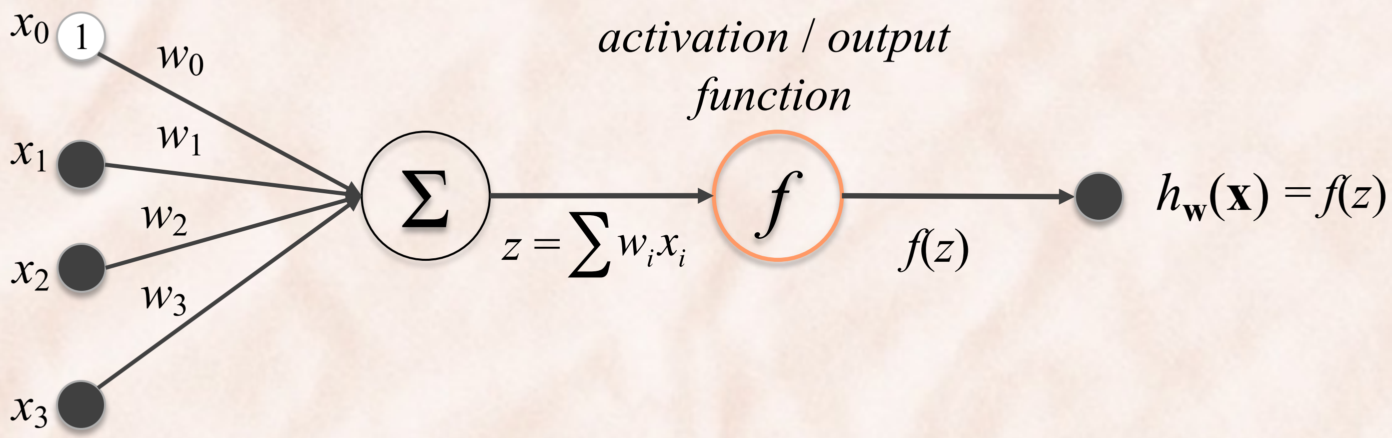
Fig. 2. (a) Illustration and (b) functional description of a leaky integrate-and-fire neuron. Weighted and delayed input signals are summed into the input current $I_{\mathrm{app}}(t)$, which travel to the soma and perturb the internal state variable, the voltage V. Since V is hysteric, the soma performs integration and then applies a threshold to make a spike or no-spike decision. After a spike is released, the voltage V is reset to a value V_{reset}. The resulting spike is sent to other neurons in the network.

Neuron Models

https://www.research.ibm.com/software/IBMResearch/multimedia/IJCNN2013.neuron-model.pdf

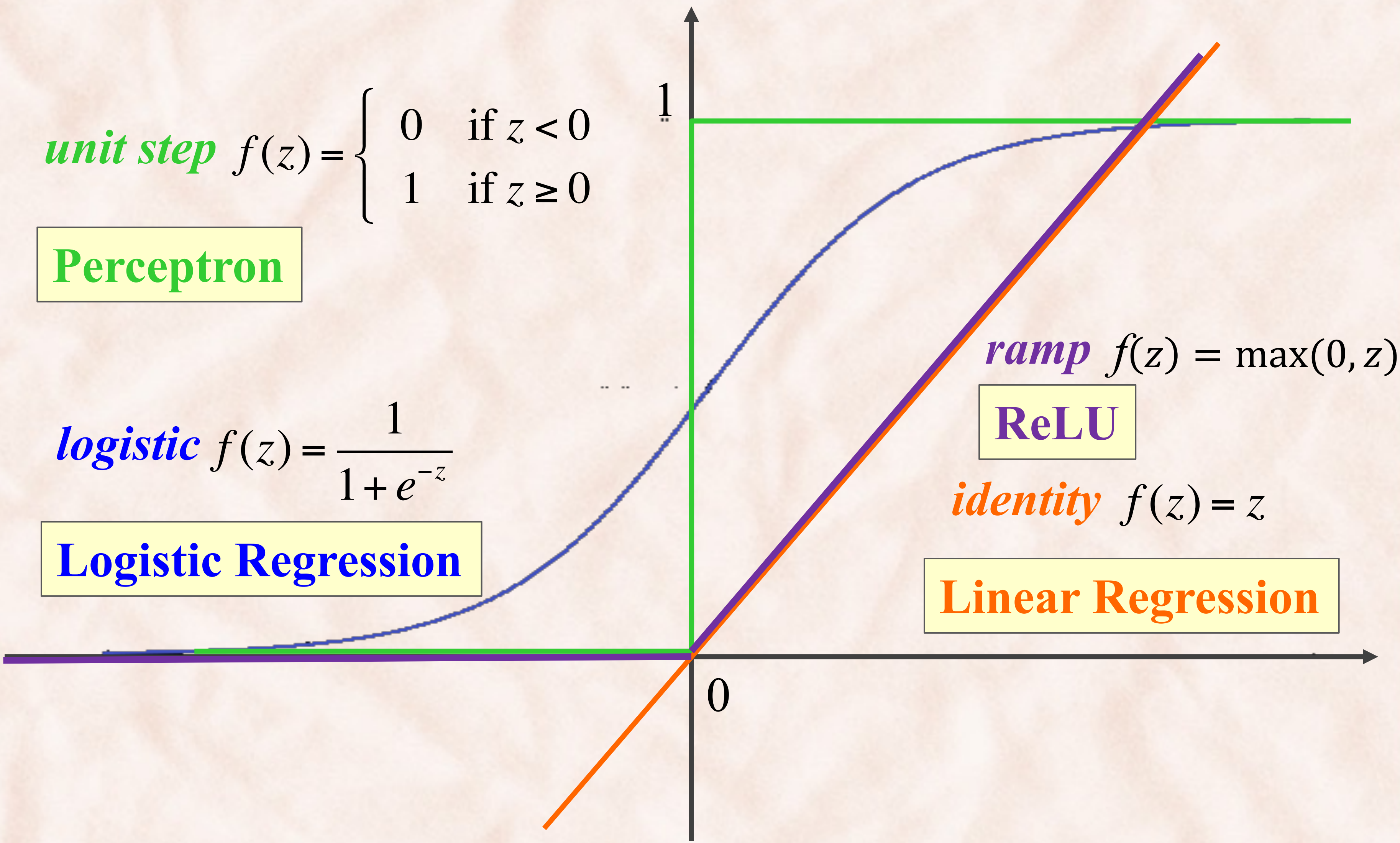
Year	Model Name	Reference
1907	Integrate and fire	[13]
1943	McCulloch and Pitts	[11]
1952	Hodgkin-Huxley	[12]
1958	Perceptron	[14]
1961	Fitzhugh-Nagumo	[15]
1965	Leaky integrate-and-fire	[16]
1981	Morris-Lecar	[17]
1986	Quadratic integrate-and-fire	[18]
1989	Hindmarsh-Rose	[19]
1998	Time-varying integrate-and-fire model	[20]
1999	Wilson Polynomial	[21]
2000	Integrate-and-fire or burst	[22]
2001	Resonate-and-fire	[23]
2003	Izhikevich	[24]
2003	Exponential integrate-and-fire	[25]
2004	Generalized integrate-and-fire	[26]
2005	Adaptive exponential integrate-and-fire	[27]
2009	Mihalas-Neibur	[28]

McCulloch-Pitts Neuron Function

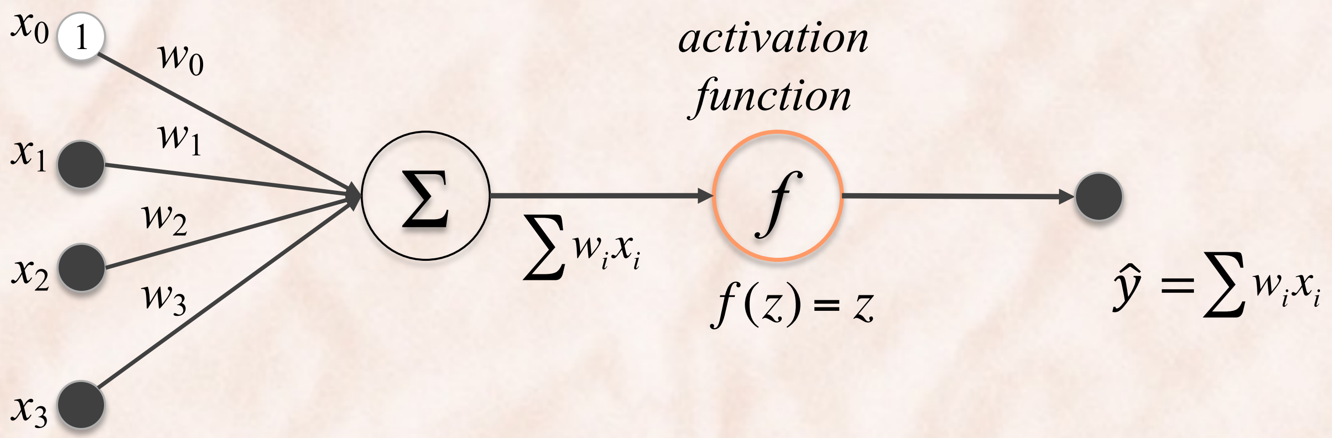


- Algebraic interpretation:
  - The output of the neuron is a **linear combination** of inputs from other neurons, **rescaled by** the synaptic **weights**.
    - weights w_i correspond to the synaptic weights (activating or inhibiting).
    - summation corresponds to combination of signals in the soma.
  - It is often transformed through an **activation / output function**.

Activation Functions



Linear Regression



- Polynomial curve fitting is Linear Regression:

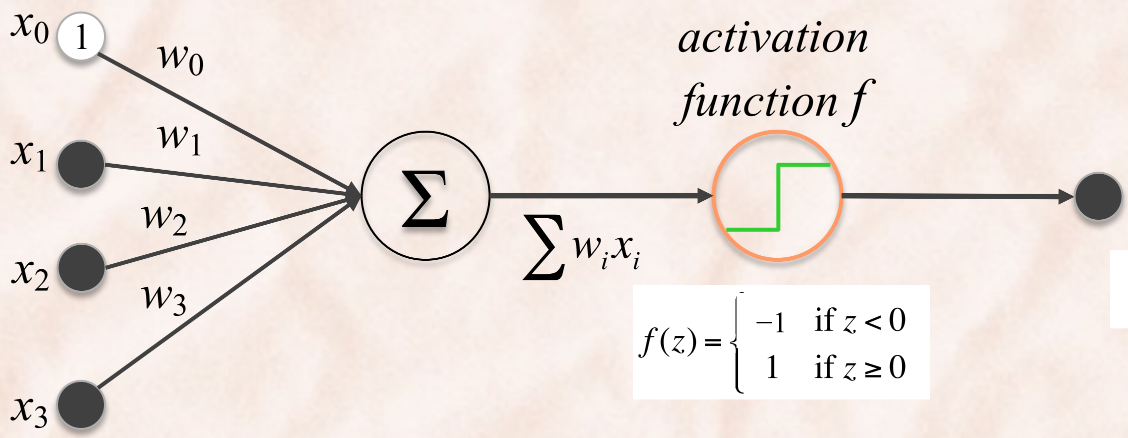
  $\mathbf{x} = \varphi(x) = [1, x, x^2, \ldots, x^M]^T$ $\qquad \hat{y} = \mathbf{w}^T\mathbf{x}$

- What **error/cost function** to minimize?

$$J(\mathbf{w}) = \frac{1}{2N}\sum_{n=1}^{N}(\hat{y}_n - y_n)^2$$

Use *normal equations* or ***gradient descent***

Perceptron



$$f(z) = \begin{cases} -1 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$

$$\hat{y} = \begin{cases} 1 & \text{if } \mathbf{w}^T\mathbf{x} > 0 \\ -1 & \text{otherwise} \end{cases}$$

- Assume classes C = $\{c_1, c_2\} = \{+1, -1\}$.
- Training set is $(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots (\mathbf{x}_n, y_n)$.

  $\mathbf{x} = [1, x_1, x_2, ..., x_k]^T$

  $\hat{y} = sgn(\mathbf{w}^T\mathbf{x}) = sgn(w_0 + w_1 x_1 + \ldots + w_k x_k)$

a linear discriminant function

Linear Discriminant Functions

- Use a linear function of the input vector:

$$h(\mathbf{x}) = \underbrace{\mathbf{w}^T}_{\text{weight vector}} \varphi(\mathbf{x}) + \underbrace{w_0}_{bias\,=\,-\,threshold}$$

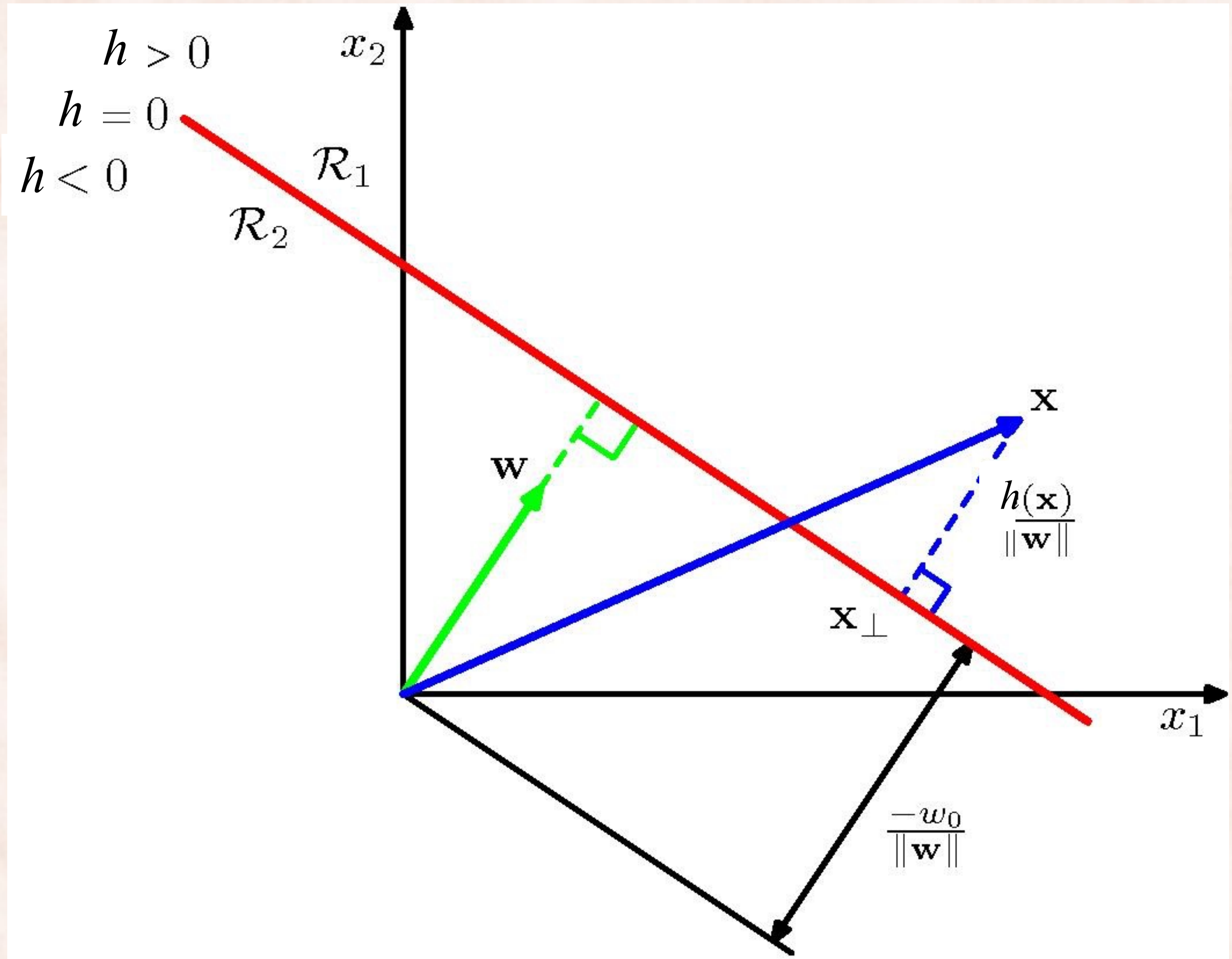
- Decision:

  $\mathbf{x} \in C_1$ if $h(\mathbf{x}) \geq 0$, otherwise $\mathbf{x} \in C_2$.

  $\Rightarrow$ decision boundary is hyperplane $h(\mathbf{x}) = 0$.

- Properties:
  - $\mathbf{w}$ is orthogonal to vectors lying within the decision surface.
  - w_0 controls the location of the decision hyperplane.

Geometric Interpretation



From Perceptron to Logistic Regression

- Features $\mathbf{x} = [1, x_1, x_2, x_3, \ldots, x_K]$
- Weights $\mathbf{w} = [w_0, w_1, w_2, w_3, \ldots, w_K]$

Discriminant function model

Perceptron

Training: Find $\mathbf{w}$ to fit training data.
Inference: Compute $h(\mathbf{x}) = \mathbf{w}^T\mathbf{x}$
Decision:
- if $h(\mathbf{x}) \geq 0$ output label +1
- else output label -1

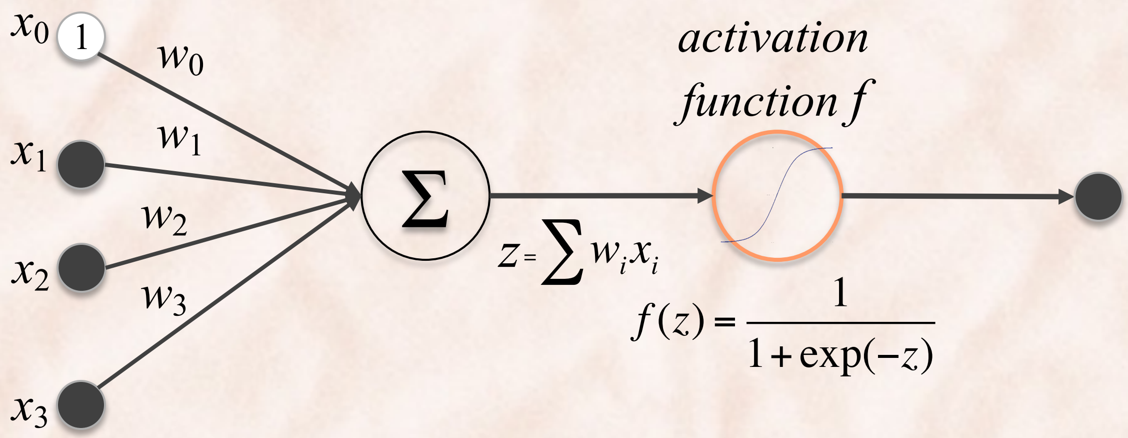
Probabilistic discriminative model

Logistic Regression

Training: Find $\mathbf{w}$ to fit training data.
Inference: Compute $z = \mathbf{w}^T\mathbf{x}$
Decision:
- if $z \geq 0$ output label 1
- else output label 0

Take logit z, compute probabilistic output $p(y = 1|\mathbf{x}) = \sigma(z) = \frac{1}{1+\exp(-z)}$

Logistic Regression for Binary Classification

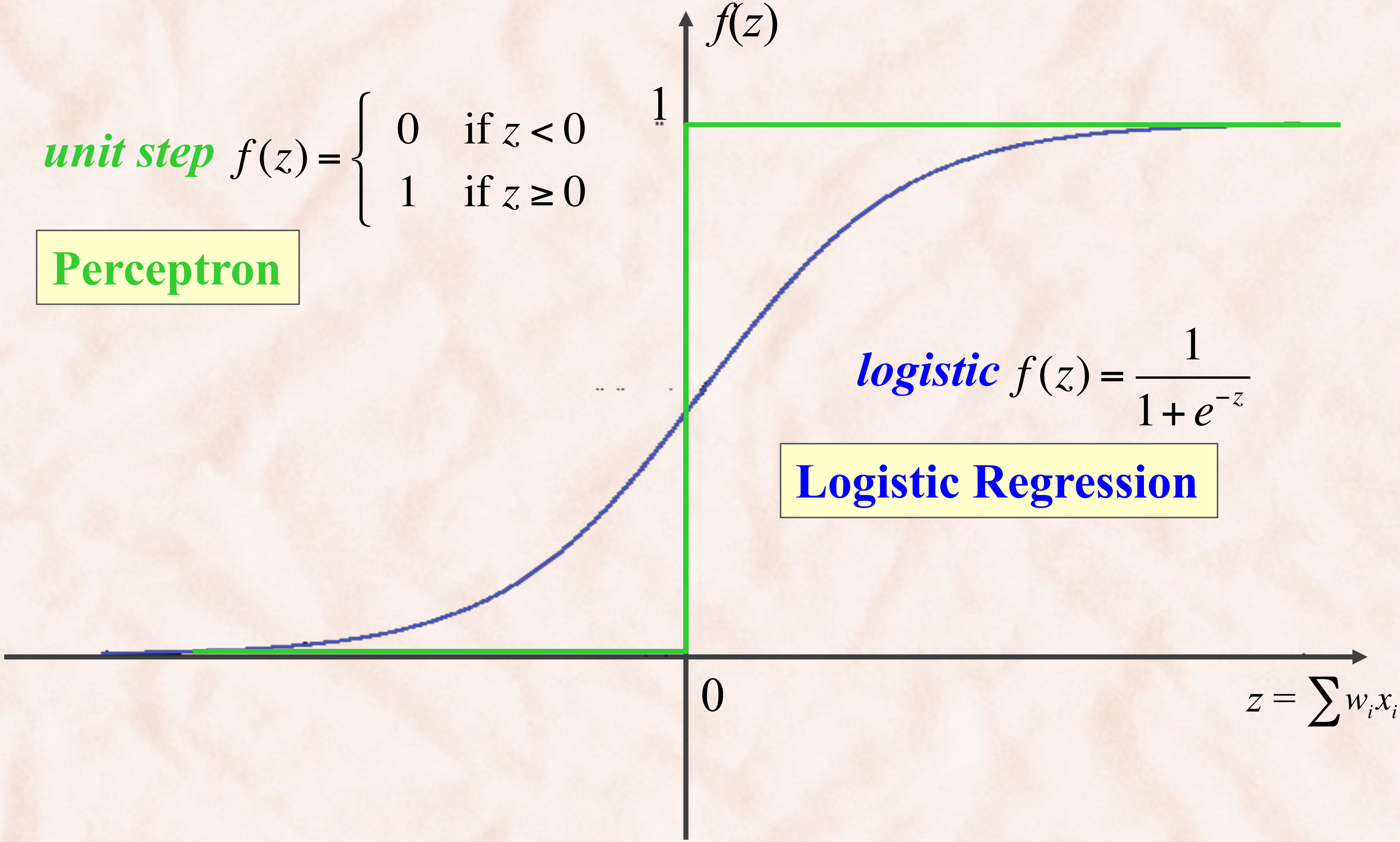


$$\hat{y} = \frac{1}{1+\exp(-\mathbf{w}^T\mathbf{x})}$$

- Used for binary **classification**:
  - Labels C = {C_1, C_2} = {1, 0}
  - Output C_1 *if and only if* $\hat{y} = \sigma(\mathbf{w}^T\mathbf{x}) > 0.5$
- Training set is $(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots (\mathbf{x}_N, y_N)$.

  $\mathbf{x} = [1, x_1, x_2, \ldots, x_K]^T$

Activation / Output Functions f



Logistic Regression for Binary Classification

- Model output can be interpreted as **posterior class probabilities**:

Prob. of +*ve* clasas: $\hat{y} = p(y = 1|\mathbf{x}) = \sigma(\mathbf{w}^T\mathbf{x}_n) = \dfrac{1}{1 + \exp(-\mathbf{w}^T\mathbf{x}_n)}$

Prob. of −*ve* class: $1 - \hat{y} = p(y = 0|\mathbf{x}) = 1 - \sigma(\mathbf{w}^T\mathbf{x}_n) = \sigma(-\mathbf{w}^T\mathbf{x}_n)$

Linear *decision boundary*

- Inference:
  - Output +*ve* class if $\hat{y} \geq 0.5$, else output −*ve* class.
    - *assuming uniform misclassification costs …*

Example: Text Classification

- **Input**:
  - a document $\mathbf{x}$, represented as a **feature vector** $\mathbf{x} = [x_1, x_2, \ldots, x_n]$
  - a fixed set of classes $C = \{c_1, c_2, \ldots, c_K\}$

- **Output**:
  - a predicted class $\hat{y} \in C$
    - binary classification: prediction $\hat{y} \in \{c_1, c_2\}$

Example: Sentiment Analysis

The film is absolutely gorgeous. It's one that you really must see on the biggest, best screen you can find, preferably in a theater with really great sound. The seats were shaking at some points. There is so much spectacle here, it's a little overwhelming at times. And it's all so well-crafted. Other than the lack of sweat — still odd for such a hot planet — Arrakis feels real and we see much more of it this time around.

- For feature x_i, weight w_i tells how important x_i is for the positive label:
  - x_i ="review contains '`gorgeous`'": $w_i =$ `+10`
  - x_j ="review contains '`abysmal`'": $w_j =$ `-10`
  - x_k ="review contains '`mediocre`'": $w_k =$ `-2`

Logistic Regression for Text Classification

- Input observation:
  - Document vector $\mathbf{x} = [x_1, x_2, \ldots, x_n]$
- Weights:
  - One per feature: $\mathbf{w} = [w_1, w_2, \ldots, w_n]$
- Output:
  - **Binary** logistic regression:
    - predicted class $\hat{y} \in \{0,1\}$
  - **Multinomial** logistic regression:
    - predicted class $\hat{y} \in \{0, 1, 2, \ldots\}$

Classification with Logistic Regression

- For each feature x_i, weight w_i tells us importance of x_i
  - Plus we'll have a bias b (we called it w_0 earlier …)

- We'll sum up all the weighted features and the bias:

$$z = \left(\sum_{i=1}^{n} w_i x_i\right) + b$$

- If this sum is high, we say $y = 1$; if low, then $y = 0$

$$z = w \cdot x + b$$

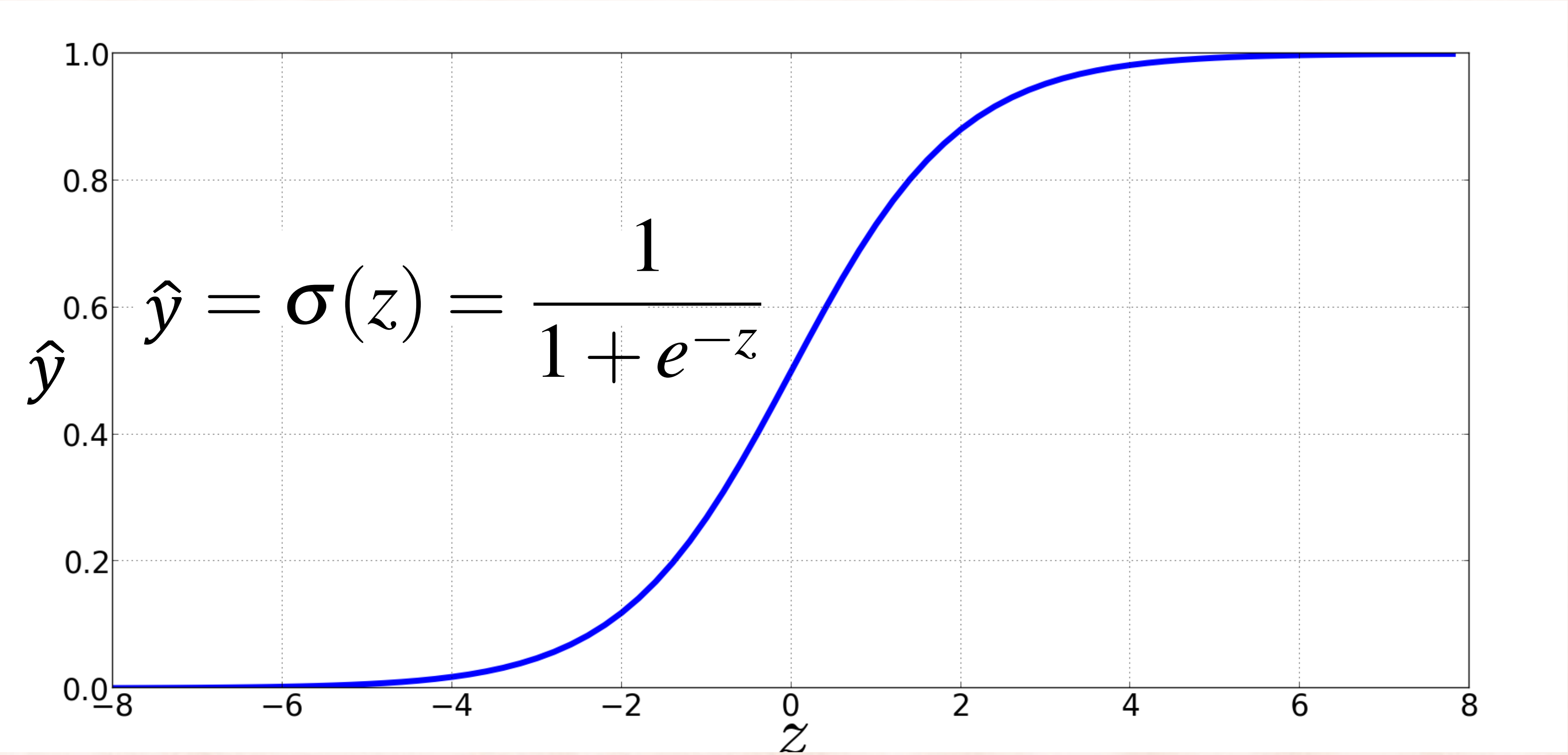
From logit z to probability p

- **Problem**: z isn't a probability, it's just a number!

- **Solution**: use a function of z that goes from 0 to 1.
  - the *logistic sigmoid*.

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

The very useful logistic sigmoid



Making probabilities with sigmoids

$$\begin{aligned}
P(y=1) &= \sigma(w \cdot x + b) \\
&= \frac{1}{1+\exp(-(w \cdot x + b))}
\end{aligned}$$

$$\begin{aligned}
P(y=0) &= 1 - \sigma(w \cdot x + b) \\
&= 1 - \frac{1}{1+\exp(-(w \cdot x + b))} \\
&= \frac{\exp(-(w \cdot x + b))}{1+\exp(-(w \cdot x + b))} = \sigma(-(w \cdot x + b))
\end{aligned}$$

Turning a probability into a classifier

- We'll compute $\mathbf{w}^T\mathbf{x}+b$
- And then we'll pass it through the sigmoid function:

$$\sigma(\mathbf{w}^T\mathbf{x}+b)$$

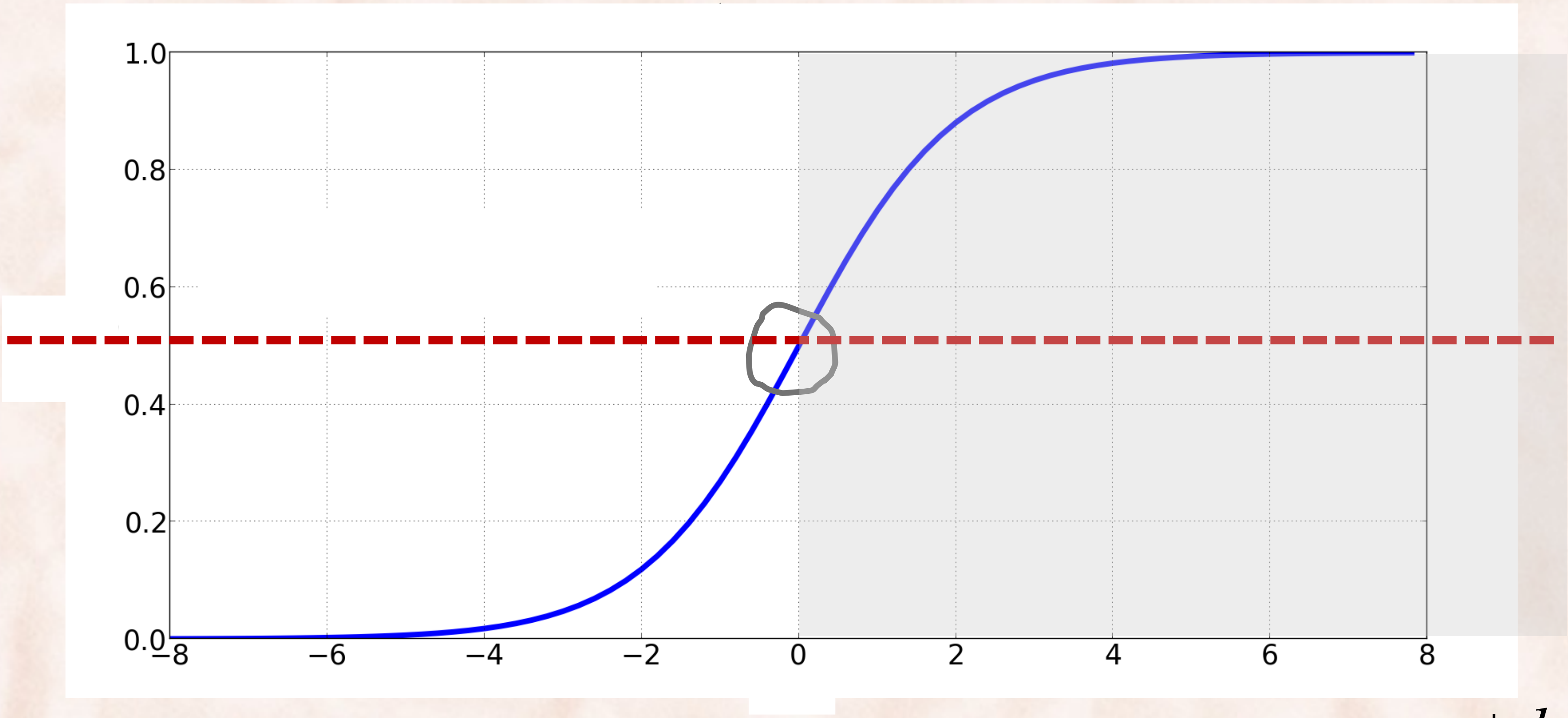
- And we'll just treat it as a probability.

$$\text{Prediction } \hat{y} = \begin{cases} 1 & \text{if } P(y=1|x) > 0.5 \quad \textit{iff } w^T x + b > 0 \\ 0 & \text{otherwise} \quad \textit{iff } w^T x + b \leq 0 \end{cases}$$

0.5 here is called the **decision threshold**

The LR Classifier

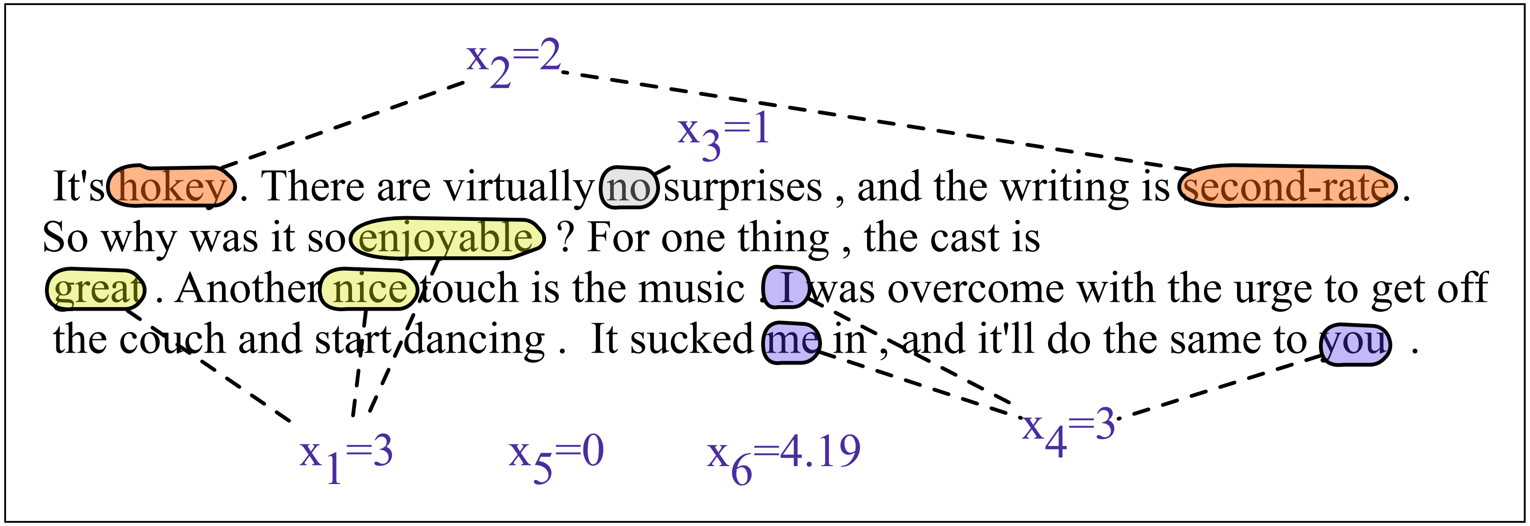
$$\hat{y} = P(y=1) \;=\; \sigma(w \cdot x + b)$$



$$z = w \cdot x + b$$

Sentiment Analysis: Does $y = 1$ or $y = 0$?

It's hokey . There are virtually no surprises , and the writing is second-rate . So why was it so enjoyable ? For one thing , the cast is great . Another nice touch is the music . I was overcome with the urge to get off the couch and start dancing . It sucked me in , and it 'll do the same to you .



Var	Definition	Value in Fig. 5.2
x_1	count(positive lexicon) $\in$ doc)	3
x_2	count(negative lexicon) $\in$ doc)	2
x_3	$\begin{cases} 1 & \text{if “no”} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$	1
x_4	count(1st and 2nd pronouns $\in$ doc)	3
x_5	$\begin{cases} 1 & \text{if “!”} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$	0
x_6	log(word count of doc)	$\ln(66) = 4.19$

Classifying Sentiment for Input $\mathbf{x}$

Var	Definition	Value in Fig. 5.2
x_1	count(positive lexicon) $\in$ doc)	3
x_2	count(negative lexicon) $\in$ doc)	2
x_3	$\begin{cases} 1 & \text{if “no”} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$	1
x_4	count(1st and 2nd pronouns $\in$ doc)	3
x_5	$\begin{cases} 1 & \text{if “!”} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$	0
x_6	log(word count of doc)	$\ln(66) = 4.19$

Suppose $\mathbf{w} = [2.5, -5.0, -1.2, 0.5, 2.0, 0.7]$

$b = 0.1$

Classifying Sentiment for Input **x**

$$\begin{aligned} p(+|x) = P(Y=1|x) &= \sigma(w \cdot x + b) \\ &= \sigma([2.5, -5.0, -1.2, 0.5, 2.0, 0.7] \cdot [3, 2, 1, 3, 0, 4.19] + 0.1) \\ &= \sigma(.833) \\ &= 0.70 \end{aligned} \tag{5.6}$$

$$\begin{aligned} p(-|x) = P(Y=0|x) &= 1 - \sigma(w \cdot x + b) \\ &= 0.30 \end{aligned}$$

(Binary) Logistic Regression: Summary

- Given as **input**:
  - a set of classes: (+*ve* sentiment, −*ve* sentiment)
  - a vector **x** of features $[x_1, x_2, \ldots, x_n]$
    - $x_1 = count(\text{"awesome"})$.
    - $x_2 = log(\text{number of words in review})$.
  - a vector **w** of weights $[w_1, w_2, \ldots, w_n]$
- Logistic Regression computes as **output**:

$$\boxed{\begin{aligned} P(y=1) &= \sigma(w \cdot x + b) \\ &= \frac{1}{1+e^{-(w \cdot x + b)}} \end{aligned}}$$

Wait, where did the **w** come from?

- **Supervised learning** for classification:
  - We know the correct label y (either 0 or 1) for each training $\mathbf{x}$.
  - What the system produces is an estimate $\hat{y} = p(y = 1|\mathbf{x})$

- **Training**: we want to set $\mathbf{w}$ and b to minimize the **distance** between our estimate $\hat{y}$ and the true y.
  - We need a distance estimator: a **loss function** or a **cost function**
    - **Cross-Entropy** loss = **Negative Log-Likelihood (NLL)**
  - We need an **optimization algorithm** to update w and b to minimize the loss.
    - **Stochastic Gradient Descent (SGD)**

Logistic Regression for Binary Classification

- Model output can be interpreted as **class probabilities**:

Prob. of $+ve$ class: $\hat{y} = p(y = 1|\mathbf{x}) = \sigma(\mathbf{w}^T\mathbf{x}_n) = \dfrac{1}{1 + \exp(-\mathbf{w}^T\mathbf{x}_n)}$

Prob. of $-ve$ class: $1 - \hat{y} = p(y = 0|\mathbf{x}) = 1 - \sigma(\mathbf{w}^T\mathbf{x}_n) = \sigma(-\mathbf{w}^T\mathbf{x}_n)$

- How do we train a logistic regression model, i.e. how do we find parameters **w** and b?
  - What **cost function** to minimize?

Logistic Regression Learning

- Learning = finding the "right" parameters $\mathbf{w}^T = [w_0, w_1, \dots, w_K]$
  - Find $\mathbf{w}$ that minimizes a ***cost function*** $J(\mathbf{w})$ which measures the misfit between $\hat{y}_n$ and y_n.
  - Expect that if model performing well on training examples $\mathbf{x}_n$
    $\Rightarrow$ same model will perform well on arbitrary test examples $\mathbf{x} \in X$.

- **Least Squares** cost function?

$$J(\mathbf{w}) = \frac{1}{2N}\sum_{n=1}^{N}(\hat{y}_n - y_n)^2$$

  - Differentiable => can use gradient descent ✓
  - Non-convex => not guaranteed to find the global optimum ✗

Maximum Likelihood Estimation

Maximum Likelihood Estimation (MLE): find parameters that maximize the likelihood of the labels $\mathbf{y} = [y_1, y_2, \ldots, y_N]$

- The likelihood function is: $p(\mathbf{y}|\mathbf{w}, \mathrm{X}) = \prod_{n=1}^{N} p(y_n|\mathbf{w}, \mathbf{x}_n)$

- The negative log-likelihood (cross entropy) loss:

$$L(\mathbf{w}) = -\ln p(\mathbf{y}|\mathbf{w}) = -\sum_{n=1}^{N} \ln p(y_n|\mathbf{x}_n)$$

$$p(y_n = 1|\mathbf{x}) = \sigma(\mathbf{w}^T\mathbf{x}_n) = \frac{1}{1 + \exp(-\mathbf{w}^T\mathbf{x}_n)}$$

Maximum Likelihood Estimation

Training set is D = $\{\langle \mathbf{x}_n, y_n \rangle \mid y_n \in \{0,1\}, n \in 1\ldots\mathrm{N}\}$

We have defined $\hat{y}_n = p(y_n = 1|\mathbf{x}_n) = \sigma(\mathbf{w}^T\mathbf{x}_n)$

Maximum Likelihood Estimation (MLE) principle: find parameters that maximize the likelihood of the labels.

- The **likelihood** is $p(\mathbf{y}|\mathbf{w}) = \prod_{n=1}^{N} \hat{y}_n{}^{y_n}(1-\hat{y}_n)^{(1-y_n)}$

- The negative log-likelihood (cross entropy) **cost function**:

$$L(\mathbf{w}) = -\ln p(\mathbf{y}|\mathbf{w}) = -\sum_{n=1}^{N} y_n \ln \hat{y}_n + (1-y_n)\ln(1-\hat{y}_n)$$

MLE for Logistic Regression

- The MLE optimization problem is:

  convex in **w**

  $$\hat{\mathbf{w}} = \underset{\mathbf{w}}{\operatorname{argmin}}\, J(\mathbf{w}) = \underset{\mathbf{w}}{\operatorname{argmin}} \boxed{-\ln p(\mathbf{y}|\mathbf{w})}$$

- MLE solution is given by $\nabla L(\mathbf{w}) = 0$
  - Solve numerically with gradient based methods:
    - Stochastic gradient descent, conjugate gradient, L-BFGS, ...
  - Gradient is $\nabla L(\mathbf{w}) = \sum_{n=1}^{N} (\hat{y}_n - y_n)\mathbf{x}_n$
    - If we separate bias $b{=}w_0$ from **w**, what is $\nabla L(b)$?

Interlude on Gradient Descent

- Need to find parameters **w** that minimize the negative log-likelihood loss:

$$\hat{\mathbf{w}} = \arg\min_{\mathbf{w}} \boxed{-\sum_{n=1}^{N} \ln p(y_n|\mathbf{x}_n)}$$

the loss $L(\mathbf{w})$

method to compute loss $L(\mathbf{w})$

method to compute gradient $\nabla L(\mathbf{w})$

GD-based optimizers (SGD, ADAM, …)

params $\hat{\mathbf{w}}$

Overfitting

- A model that perfectly matches the training data may have a problem.
- It may also **overfit** to the data, modeling noise:
  - A random word that perfectly predicts y (it happens to only occur in one class) will get a very high weight.
  - Failing to generalize to a test set without this word.
- A good model should be able to **generalize**.

Overfitting

+

This movie drew me in, and it'll do the same to you.

–

I can't tell you how much I hated this movie. It sucked.

Useful or harmless features:

X1 = "this"
X2 = "movie
X3 = "hated"

X4 = "drew me in"

4gram features that just "memorize" training set and might cause problems:

X5 = "the same to you"
X7 = "tell you how much"

Overfitting

- 4-gram model on **tiny data** will just memorize the data:
  - 100% accuracy on the training set.
- But it will be surprised by novel 4-grams in the test data.
  - Low accuracy on test set.

capacity = how many params in 4-gram model?

- Models that are too powerful can **overfit** the data:
  - Fitting the details of the training data so exactly that the model doesn't generalize well to the test set.
    - How to avoid overfitting?
      - L2 and L1 Regularization in logistic regression.
      - SGD and Dropout in neural networks.

Regularized Logistic Regression

- Use a Gaussian prior over the parameters:

$$\mathbf{w} = [w_1, \ldots , w_M]^{\mathrm{T}}$$

$$p(\mathbf{w}) = N(\mathbf{0}, \alpha^{-1}\mathbf{I}) = \left(\frac{\alpha}{2\pi}\right)^{(M+1)/2} \exp\left\{-\frac{\alpha}{2}\mathbf{w}^T\mathbf{w}\right\}$$

- Bayes' Theorem:

$$p(\mathbf{w}|\mathbf{y}) = \frac{p(\mathbf{y}|\mathbf{w})p(\mathbf{w})}{p(\mathbf{y})} \propto p(\mathbf{y}|\mathbf{w})p(\mathbf{w})$$

- MAP solution:

$$\begin{aligned}\hat{\mathbf{w}} &= \arg\max_{\mathbf{w}} p(\mathbf{w}|\mathbf{y}) \\ &= \arg\max_{\mathbf{w}} p(\mathbf{y}|\mathbf{w})p(\mathbf{w})\end{aligned}$$

Regularized Logistic Regression

- MAP solution:

$$\hat{\mathbf{w}} = \arg\max_{\mathbf{w}} p(\mathbf{y}|\mathbf{w})p(\mathbf{w})$$

$$= \arg\min_{\mathbf{w}} - \ln p(\mathbf{y}|\mathbf{w}) - \ln p(\mathbf{w})$$

$$= \arg\min_{\mathbf{w}} - \ln p(\mathbf{y}|\mathbf{w}) - \frac{\alpha}{2}\mathbf{w}^T\mathbf{w}$$

$$= \boxed{\arg\min_{\mathbf{w}} \ L_D(\mathbf{w}) + L_w(\mathbf{w})}$$

still convex in **w**

$$L_D(\mathbf{w}) = -\frac{1}{N}\sum_{n=1}^{N} y_n \ln \hat{y}_n + (1 - y_n)\ln(1 - \hat{y}_n)$$

data term
(we also average)

$$L_w(\mathbf{w}) = \frac{\lambda}{2}\mathbf{w}^T\mathbf{w}$$

regularization term

Regularized Logistic Regression

- MAP (maximum likelihood + L_2 regularization) solution:

$$\mathbf{w} = \arg\min_{\mathbf{w}} L_D(\mathbf{w}) + L_C(\mathbf{w})$$

λ is also called ***decay***

$$= \arg\min_{\mathbf{w}} -\frac{1}{N}\sum_{n=1}^{N} \ln p(y_n|\mathbf{x}_n) + \frac{\lambda}{2}\|\mathbf{w}\|^2$$

- MAP solution is given by $\nabla L(\mathbf{w}) = 0$

$$\hat{y}_n = \sigma(\mathbf{w}^T\mathbf{x}_n + b)$$

$$\nabla L(\mathbf{w}) = \nabla L_D(\mathbf{w}) + \nabla L_C(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N}(\hat{y}_n - y_n)\,\mathbf{x}_n + \lambda\mathbf{w}$$

- Cannot solve analytically => solve numerically using (**stochastic**) **gradient descent** [PRML 3.1.3], conjugate gradient, L-BFGS, …

Wait, where does λ come from?

$$\mathbf{w} = \arg\min_{\mathbf{w}} -\frac{1}{N}\sum_{n=1}^{N} \ln p(y_n|\mathbf{x}_n) + \frac{\lambda}{2}\|\mathbf{w}\|^2$$

solved using e.g. SGD

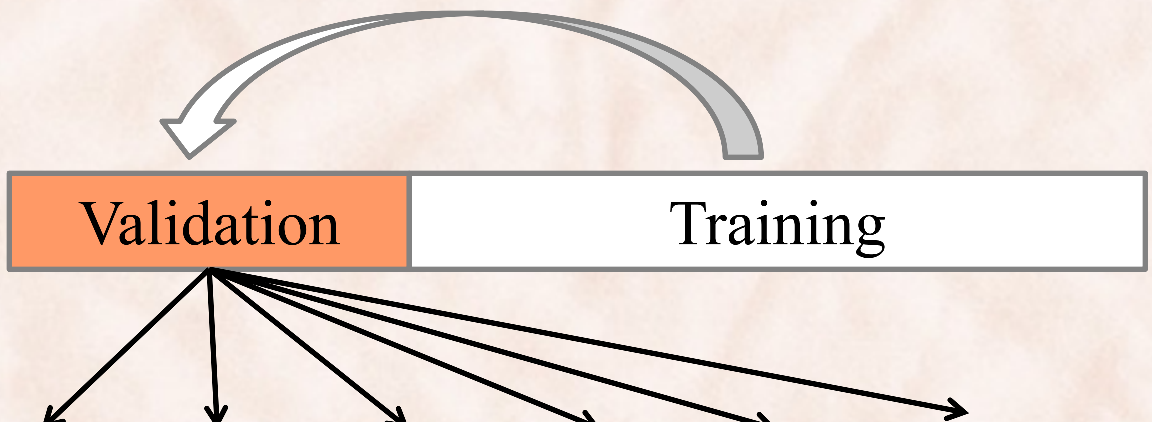
need to set λ before training

- Cannot train λ together with parameters **w**, why?
- We call λ a **hyper-parameter**.
  - We **tune** λ before **training** **w**.

Hyperparameter Tuning:
how to select a good value for hyperparam λ?

- Put aside an independent *validation set*.
- Select parameters giving best performance on validation set.

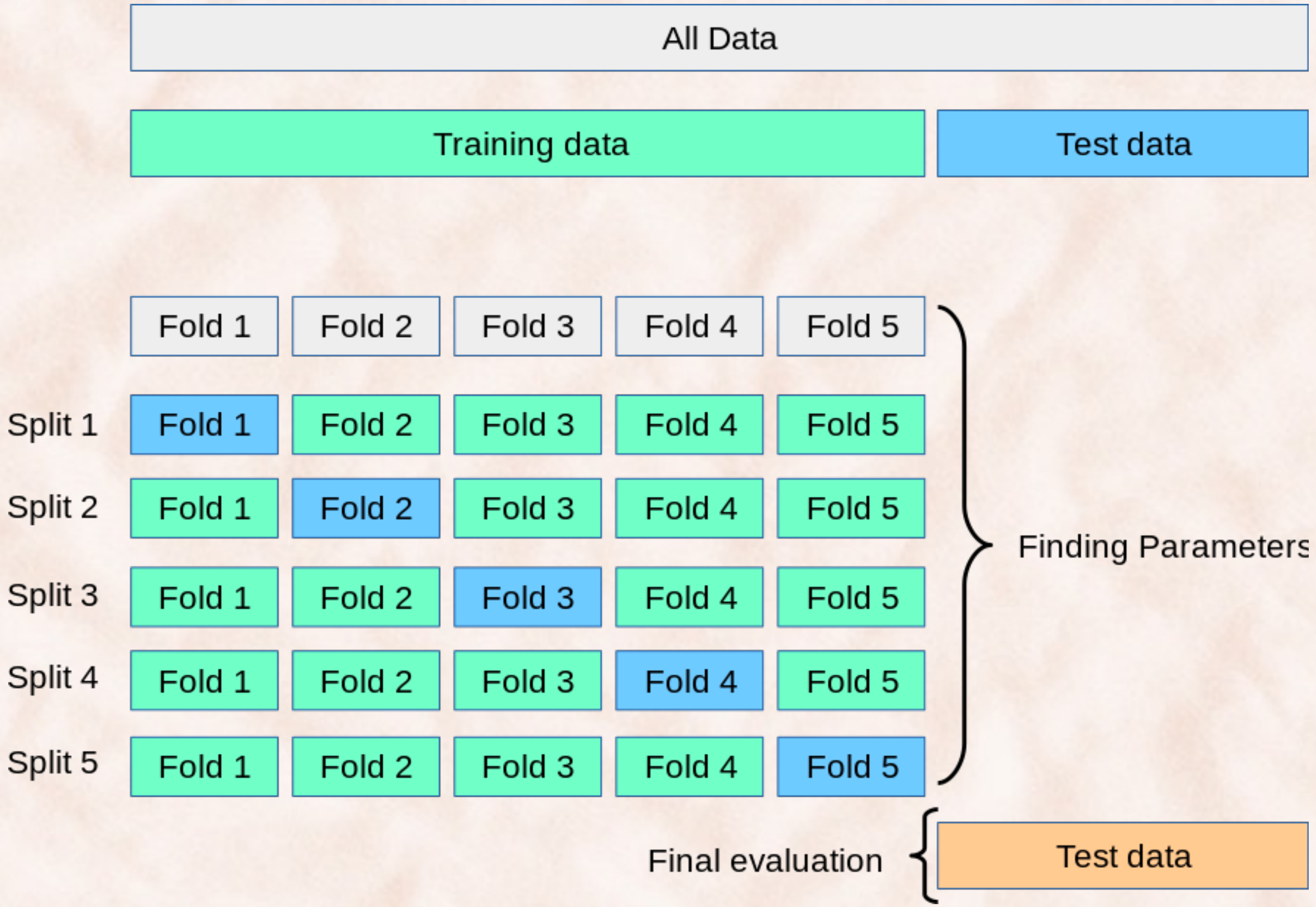
$$\ln \lambda \in \{-40, -35, -30, -25, -20, -15\}$$



ln λ	-40	-35	-30	-25	-20	-15
Loss	1.05	0.30	0.25	0.27	0.52	0.55

K-fold Cross-Validation

https://scikit-learn.org/stable/modules/cross_validation.html



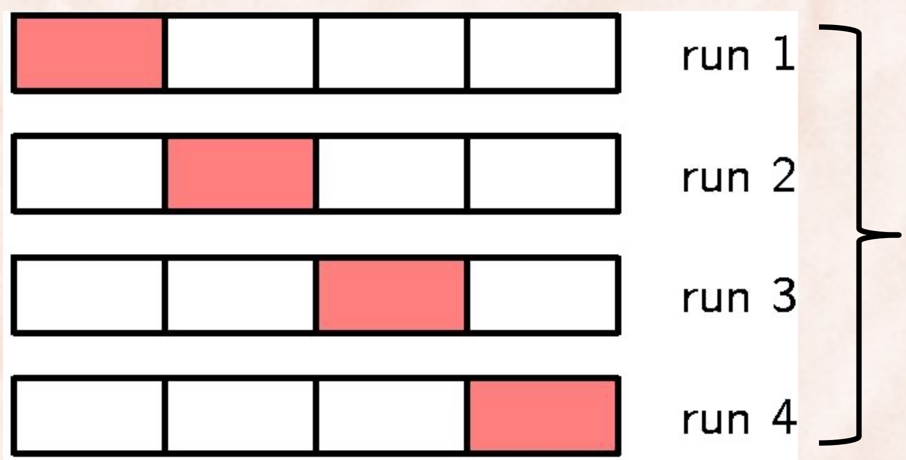
K-fold Cross-Validation

- Split the training data into K folds and try a wide range of tunning parameter values:
  - split the data into K folds of roughly equal size
  - iterate over a set of values for λ
    - iterate over k = 1, 2, ..., K
      - use all folds except k for training
      - validate (calculate test error) in the k-th fold
    - loss[λ] = average loss over the K folds
  - choose the value of λ that gives the smallest loss.

https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LassoCV.html

Model Evaluation

- K-fold evaluation:
  - randomly partition dataset in K equally sized subsets P_1, P_2, … P_k
  - for each fold *i* in {1, 2, …, k}:
    - test on P_i, train on $P_1 \cup \ldots \cup P_{i-1} \cup P_{i+1} \cup \ldots \cup P_k$
  - compute average error/accuracy across K folds.



4-fold evaluation

Implementation: Vectorization of LR

- **Version 1**: Compute gradient component-wise.

$$\nabla L(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N}(\hat{y}_n - y_n)\mathbf{x}_n$$

```
grad = np.zeros(K)
for n in range(N):
    h = sigmoid(w.dot(X[n]))   // This NumPy code assumes examples stored in rows of X.
    temp = h − y[n]
    for k in range(K):
      grad[k] = grad[k] + temp * X[k,n] / N
```

```
def sigmoid(x):
    return 1 / (1 + np.exp(−x))
```

Implementation: Vectorization of LR

- **Version 2**: Compute gradient, partially vectorized.

$$\nabla L(\mathbf{w}) = \frac{1}{N} \sum_{n=1}^{N} (\hat{y}_n - y_n)\mathbf{x}_n$$

```
grad = np.zeros(K)
for n in range(N):      // This NumPy code assumes examples stored in rows of X.
    grad = grad + (sigmoid(w.dot(X[n])) − y[n]) * X[n] / N
```

```
def sigmoid(x):
    return 1 / (1 + np.exp(−x))
```

Implementation: Vectorization of LR

- **Version 3**: Compute gradient, vectorized.

$$\nabla L(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N}(\hat{y}_n - y_n)\mathbf{x}_n$$

grad = X.T.dot(sigmoid(X.dot(**w**) − **y**)) / N

```
def sigmoid(x):
    return 1 / (1 + np.exp(−x))
```

Vectorization of LR with Separate Bias

- Separate the bias b from the weight vector $\mathbf{w}$.
- Compute gradient separately with respect to $\mathbf{w}$ and b:

$$\hat{y}_n = \sigma(\mathbf{w}^T \mathbf{x}_n + b)$$

  – Gradient with respect to $\mathbf{w}$ is: $\nabla L(\mathbf{w}) = \frac{1}{N} \sum_{n=1}^{N} (y_n - \hat{y}_n) \mathbf{x}_n$

grad_w = X.T.dot(sigmoid(X.dot(**w**) + *b*) − **y**) / N

  – Gradient with respect to bias b is: $\Delta L(b) = -\frac{1}{N} \sum_{n=1}^{N} (\hat{y}_n - y_n)$

grad_b = *# YOUR CODE HERE* ☺

Vectorization of LR with Regularization

- Only the gradient with respect to **w** changes:
  - never use L_2 regularization on bias.

$$\nabla L(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N}(\hat{y}_n - y_n)\,\mathbf{x}_n + \lambda\mathbf{w}$$

---

grad = X.T.dot(sigmoid(X.dot(**w**) + b) − **y**) / N + α**w**

Binary Logistic Regression in *sklearn*

scikit-learn.org/stable/modules/linear_model.html#logistic-regression

scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

1.1.11.1. Binary Case

For notational ease, we assume that the target y_i takes values in the set $\{0, 1\}$ for data point i. Once fitted, the `predict_proba` method of `LogisticRegression` predicts the probability of the positive class $P(y_i = 1|X_i)$ as

$$\hat{p}(X_i) = \operatorname{expit}(X_i w + w_0) = \frac{1}{1 + \exp(-X_i w - w_0)}.$$

As an optimization problem, binary class logistic regression with regularization term $r(w)$ minimizes the following cost function:

$$\min_w \frac{1}{S} \sum_{i=1}^{n} s_i \left(-y_i \log(\hat{p}(X_i)) - (1 - y_i) \log(1 - \hat{p}(X_i))\right) + \frac{r(w)}{SC}, \quad (1)$$

where s_i corresponds to the weights assigned by the user to a specific training sample (the vector s is formed by element-wise multiplication of the class weights and sample weights), and the sum $S = \sum_{i=1}^{n} s_i$.

We currently provide four choices for the regularization term $r(w)$ via the `penalty` argument:

penalty	$r(w)$
`None`	0
ℓ_1	$\|w\|_1$
ℓ_2	$\frac{1}{2}\|w\|_2^2 = \frac{1}{2}w^T w$
`ElasticNet`	$\frac{1-\rho}{2} w^T w + \rho \|w\|_1$

1.1.11.3. Solvers

The solvers implemented in the class `LogisticRegression` are “lbfgs”, “liblinear”, “newton-cg”, “newton-cholesky”, “sag” and “saga”:

The following table summarizes the penalties and multinomial multiclass supported by each solver:

	Solvers					
Penalties	**‘lbfgs’**	**‘liblinear’**	**‘newton-cg’**	**‘newton-cholesky’**	**‘sag’**	**‘saga’**
L2 penalty	yes	no	yes	no	yes	yes
L1 penalty	no	yes	no	no	no	yes
Elastic-Net (L1 + L2)	no	no	no	no	no	yes
No penalty (‘none’)	yes	no	yes	yes	yes	yes
Multiclass support						
multinomial multiclass	yes	no	yes	no	yes	yes
Behaviors						
Penalize the intercept (bad)	no	yes	no	no	no	no
Faster for large datasets	no	no	no	no	yes	yes
Robust to unscaled datasets	yes	yes	yes	yes	no	no

What if we have K > 2 classes?

- Logit score z is still the dot product between a weight vector and the input vector.

- But now we have a separate weight vector $\mathbf{w}_c$ for each class $c = 1, 2, \ldots, k$

$$z_c = \mathbf{w}_c^T \mathbf{x}$$

- How do we transform z_c into a probability p_c?

What if we have K > 2 classes?

- Need a generalization of the sigmoid σ called the **softmax**:
  - Softmax takes as input a vector $\mathbf{z} = [z_1, z_2, ..., z_K]$ of K values.
  - It outputs a probability distribution softmax($\mathbf{z}$) = $\mathbf{p} = [p_1, p_2, ..., p_K]$
    - Need each value in the range [0,1].
    - Need all the values summing to 1.

$$\text{softmax}([z_1, z_2, ..., z_K]) = [p_1, p_2, ..., p_K]$$

$$\text{softmax}(z) = \left[\frac{\exp(z_1)}{\sum_{i=1}^{k}\exp(z_i)}, \frac{\exp(z_2)}{\sum_{i=1}^{k}\exp(z_i)}, \ldots, \frac{\exp(z_k)}{\sum_{i=1}^{k}\exp(z_i)}\right]$$

The **softmax** function

- Turns a vector $\mathbf{z} = [z_1, z_2, ..., z_k]$ of k values into probabilities:

$$z = [0.6, 1.1, -1.5, 1.2, 3.2, -1.1]$$

$$\text{softmax}(z) = \left[\frac{\exp(z_1)}{\sum_{i=1}^{k} \exp(z_i)}, \frac{\exp(z_2)}{\sum_{i=1}^{k} \exp(z_i)}, ..., \frac{\exp(z_k)}{\sum_{i=1}^{k} \exp(z_i)} \right]$$

$$\text{softmax}(z) = [0.055, 0.090, 0.0067, 0.10, 0.74, 0.010]$$

What if we have K > 2 classes?

- Logit score z_c is still the dot product between a weight vector and the input vector.
- But now we have a separate weight vector $\mathbf{w}_c$ for each class $c = 1, 2, \ldots, k$

$$p(y = c|\mathbf{x}) = \frac{\exp(z_c)}{\sum_{j=1}^{k} \exp(z_j)}$$

$$= \frac{\exp(\mathbf{w}_c^T \mathbf{x})}{\sum_{j=1}^{k} \exp(\mathbf{w}_j^T \mathbf{x})}$$

Multinomial Logistic Regression in *sklearn*

scikit-learn.org/stable/modules/linear_model.html#logistic-regression

scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

For multiclass problems, if you want multinomial, choose ‘newton-cg’, ‘sag’, ‘saga’ or ‘lbfgs’ for training.

The choice of the algorithm depends on the penalty chosen and on (multinomial) multiclass support:

solver	penalty	multinomial multiclass
‘lbfgs’	‘l2’, None	yes
‘liblinear’	‘l1’, ‘l2’	no
‘newton-cg’	‘l2’, None	yes
‘newton-cholesky’	‘l2’, None	no
‘sag’	‘l2’, None	yes
‘saga’	‘elasticnet’, ‘l1’, ‘l2’, None	yes

Temperature

- Softmax with a *temperature* parameter $T \geq 0$:

$$p(y = c|\mathbf{x}) = \frac{\exp(z_c/T)}{\sum_{j=1}^{k} \exp(z_j/T)}$$

- When $T = 1$, we get the original softmax distribution.
  - What happens when T = 0?
  - What happens when T > 1?
  - What happens when T < 1?

- $T = 0$ and $T > 1$ widely used for generation with LLMs!

Softmax Regression = Logistic Regression for Multiclass Classification

- Multiclass classification:

  $T = \{C_1, C_2, ..., C_K\} = \{1, 2, ..., K\}$.

- Training set is $(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots (\mathbf{x}_N, y_N)$.

  $\mathbf{x} = [1, x_1, x_2, ..., x_M]$

  $y_1, y_2, \ldots y_N \in \{1, 2, ..., K\}$

- One weight vector per class [PRML 4.3.4]:

$$p(C_k \mid \mathbf{x}) = \frac{\exp(\mathbf{w}_k^T \mathbf{x}))}{\sum_j \exp(\mathbf{w}_j^T \mathbf{x})}$$

bias parameter inside each $\mathbf{w}_j$

$$p(C_k \mid \mathbf{x}_n) = \frac{\exp(\mathbf{w}_k^T \mathbf{x}_n + b_k)}{\sum_{j=1..K} \exp(\mathbf{w}_j^T \mathbf{x}_n + b_j)}$$

separate bias parameter b_j

Softmax Regression (K ≥ 2)

- Inference:

$$C_* = \arg\max_{C_k} p(C_k \mid \mathbf{x})$$

$$= \arg\max_{C_k} \frac{\exp(\mathbf{w}_k^T \mathbf{x})}{\boxed{\sum_j \exp(\mathbf{w}_j^T \mathbf{x})}}$$

Z(**x**) *a normalization constant*

$$= \arg\max_{C_k} \exp(\mathbf{w}_k^T \mathbf{x})$$

$$= \arg\max_{C_k} \mathbf{w}_k^T \mathbf{x}$$

- Training using:
  - Maximum Likelihood (ML)
  - Maximum A Posteriori (MAP) with a Gaussian prior on **w**.

Softmax Regression

- The **negative log-likelihood** error function is:

$$E_D(\mathbf{w}) = -\frac{1}{N}\ln\prod_{n=1}^{N} p(t_n \mid \mathbf{x}_n) = \boxed{-\frac{1}{N}\sum_{n=1}^{N}\ln\frac{\exp(\mathbf{w}_{t_n}^T\mathbf{x}_n)}{Z(\mathbf{x}_n)}}$$

convex in **w**

- The Maximum Likelihood solution is:

$$\mathbf{w}_{ML} = \arg\min_{\mathbf{w}} E_D(\mathbf{w})$$

- The **gradient** is (prove it):

$$\nabla_{\mathbf{w}_k} E_D(\mathbf{w}) = -\frac{1}{N}\sum_{n=1}^{N}\left(\delta_k(t_n) - p(C_k \mid \mathbf{x}_n)\right)\mathbf{x}_n$$

where $\delta_t(x) = \begin{cases} 1 & x = t \\ 0 & x \neq t \end{cases}$ is the *Kronecker delta* function.

Regularized Softmax Regression

- The new **cost** function is:

$$E(\mathbf{w}) = E_D(\mathbf{w}) + E_{\mathbf{w}}(\mathbf{w})$$

$$= -\frac{1}{N}\sum_{n=1}^{N} \ln \frac{\exp\left(\mathbf{w}_{t_n}^T \mathbf{x}_n\right)}{Z(\mathbf{x}_n)} + \frac{\alpha}{2}\|\mathbf{W}\|^2$$

- The new **gradient** is (prove it):

$$\boldsymbol{grad}_k = \nabla_{\mathbf{w}_k} E(\mathbf{w}) = -\frac{1}{N}\sum_{n=1}^{N}\left(\delta_k(t_n) - p(C_k \mid \mathbf{x}_n)\right)\mathbf{x}_n + \alpha \mathbf{w}_k$$

Softmax Regression

- ML solution is given by $\nabla E_D(\mathbf{w}) = 0$.
  - Cannot solve analytically.
  - Solve numerically, by pluging [*cost*, *gradient*] = $[E(\mathbf{w}), \nabla E(\mathbf{w})]$ values into general convex solvers:
    - L-BFGS
    - Newton methods
    - conjugate gradient
    - (stochastic / minibatch) gradient-based methods.
      - gradient descent (with / without momentum).
      - AdaGrad, AdaDelta
      - RMSProp
      - ADAM, ...

Implementation

- Need to compute [*cost*, ***grad***]:
  - *cost* $= -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K}\delta_k(t_n)\ln p(C_k \mid \mathbf{x}_n) + \frac{\alpha}{2}\sum_{k=1}^{K}\mathbf{w}_k^T\mathbf{w}_k$
  - ***grad***$_k$ $= -\frac{1}{N}\sum_{n=1}^{N}\left(\delta_k(t_n) - p(C_k \mid \mathbf{x}_n)\right)\mathbf{x}_n + \alpha\mathbf{w}_k$

=> need to compute, for k = 1, ..., K:

- *output* $p(C_k \mid \mathbf{x}_n) = \frac{\exp(\mathbf{w}_k^T\mathbf{x}_n))}{\sum_j \exp(\mathbf{w}_j^T\mathbf{x}_n)}$

Overflow when $\mathbf{w}_k{}^T\mathbf{x}_n$ are too large.

Implementation: Preventing Overflows

- Subtract from each product $\mathbf{w}_k{}^{\mathrm{T}}\mathbf{x}_n$ the maximum product:

$$c_n = \max_{1 \le k \le K} \mathbf{w}_k^T \mathbf{x}_n$$

$$p(C_k \mid \mathbf{x}_n) = \frac{\exp(\mathbf{w}_k^T \mathbf{x}_n - c_n))}{\sum_j \exp(\mathbf{w}_j^T \mathbf{x}_n - c_n)}$$

- When using separate bias b_k, replace $\mathbf{w}_k^T \mathbf{x}_n$ everywhere with $\mathbf{w}_k^T \mathbf{x}_n + b_k$.

Vectorization of Softmax with Separate Bias

- Separate the bias b_k from the weight vector $\mathbf{w}_k$.
- Compute gradient separately with respect to $\mathbf{w}_k$ and b_k :
  - Gradient with respect to $\mathbf{w}_k$ is:

$$\mathbf{grad}_k = -\frac{1}{N}\sum_{n=1}^{N}\left(\delta_k(t_n) - p(C_k \mid \mathbf{x}_n)\right)\mathbf{x}_n + \alpha \mathbf{w}_k$$

Gradient matrix is $[\mathbf{grad}_1 \mid \mathbf{grad}_2 \mid \ldots \mid \mathbf{grad}_K]$

$$p(C_k|\mathbf{x}_n) = \frac{\exp(\mathbf{w}_k^T\mathbf{x}_n + b_k)}{\sum_{j=1..K}\exp(\mathbf{w}_j^T\mathbf{x}_n + b_j)}$$

  - Gradient with respect to b_k is:

$$\Delta b_k = -\frac{1}{N}\sum_{n=1}^{N}(\delta_k(t_n) - p(C_k|\mathbf{x}_n))$$

$$\delta_k(t_n) = \begin{cases} 1\,, if\ t_n = k \\ 0\,, if\ t_n \neq k \end{cases}$$

Gradient vector is $\Delta \mathrm{b} = [\Delta b_1 \mid \Delta b_2 \mid \ldots \mid \Delta b_K]$

Vectorization of Softmax

- Need to compute [*cost*, ***grad***, Δb]:

$$p(C_k|\mathbf{x}_n) = \frac{\exp(\mathbf{w}_k^T\mathbf{x}_n + b_k)}{\sum_{j=1..K}\exp(\mathbf{w}_j^T\mathbf{x}_n + b_j)}$$

  - *cost* $= -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K}\delta_k(t_n)\ln p(C_k \mid \mathbf{x}_n) + \frac{\alpha}{2}\sum_{k=1}^{K}\mathbf{w}_k^T\mathbf{w}_k$
  - ***grad***$_k$ $= -\frac{1}{N}\sum_{n=1}^{N}\left(\delta_k(t_n) - p(C_k \mid \mathbf{x}_n)\right)\mathbf{x}_n + \alpha\mathbf{w}_k$

=> compute ground truth matrix G such that G[k,n] = $\delta_k(t_n)$

$$\delta_k(t_n) = \begin{cases} 1, if\ t_n = k \\ 0, if\ t_n \neq k \end{cases}$$

```
from scipy.sparse import coo_matrix
groundTruth = coo_matrix((np.ones(N, dtype = np.uint8),
                         (labels, np.arange(N)))).toarray()
```

Vectorization of Softmax

- Compute $cost = -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K}\delta_k(t_n)\ln p(C_k \mid \mathbf{x}_n) + \frac{\alpha}{2}\sum_{k=1}^{K}\mathbf{w}_k^T\mathbf{w}_k$
  - Compute matrix of $\mathbf{w}_k^T\mathbf{x}_n + b_k$.
  - Compute matrix of $\mathbf{w}_k^T\mathbf{x}_n + b_k - c_n$.
  - Compute matrix of $\exp(\mathbf{w}_k^T\mathbf{x}_n + b_k - c_n)$.
  - Compute matrix of $\ln p(C_k|\mathbf{x}_n)$.
  - Compute log-likelihood cost using all the above.

$$p(C_k|\mathbf{x}_n) = \frac{\exp(\mathbf{w}_k^T\mathbf{x}_n + b_k)}{\sum_{j=1..K}\exp(\mathbf{w}_j^T\mathbf{x}_n + b_j)}$$

$$\delta_k(t_n) = \begin{cases} 1\,, if\ t_n = k \\ 0\,, if\ t_n \neq k \end{cases}$$

$$c_n = \max_{1\le k\le K} \mathbf{w}_k^T\mathbf{x}_n + b_k$$

$$\ln p(C_k|\mathbf{x}_n) = \mathbf{w}_k^T\mathbf{x}_n + b_k - \ln\Big(\sum_{j=1..K}\exp(\mathbf{w}_j^T\mathbf{x}_n + b_j)\Big)$$

Vectorization of Softmax

- Compute $\mathbf{grad}_k = -\frac{1}{N}\sum_{n=1}^{N}\left(\delta_k(t_n) - p(C_k \mid \mathbf{x}_n)\right)\mathbf{x}_n + \alpha\mathbf{w}_k$
  - **Gradient matrix** = [**grad**$_1$ | **grad**$_2$ | … | **grad**$_K$]
  - Compute matrix of $p(C_k|\mathbf{x}_n)$.
  - Compute matrix of gradient of data term.
  - Compute matrix of gradient of regularization term.
  - Compute ground truth matrix G such that G[k,n] = $\delta_k(t_n)$

$$p(C_k|\mathbf{x}_n) = \frac{\exp(\mathbf{w}_k^T\mathbf{x}_n + b_k)}{\sum_{j=1..K}\exp(\mathbf{w}_j^T\mathbf{x}_n + b_j)}$$

$$\delta_k(t_n) = \begin{cases} 1\,, if\ t_n = k \\ 0\,, if\ t_n \neq k \end{cases}$$

Vectorization of Softmax

- Useful Numpy functions:
  - np.dot()
  - np.amax()
  - np.argmax()
  - np.exp()
  - np.sum()
  - np.log()
  - np.mean()

Implementation: Gradient Checking

- Want to minimize $J(\theta)$, where θ is a scalar.

- Mathematical definition of derivative:

$$\frac{d}{d\theta}J(\theta) = \lim_{\varepsilon \to 0}\frac{J(\theta+\varepsilon)-J(\theta-\varepsilon)}{2\varepsilon}$$

- Numerical approximation of derivative:

$$\frac{d}{d\theta}J(\theta) \approx \frac{J(\theta+\varepsilon)-J(\theta-\varepsilon)}{2\varepsilon} \qquad \text{where } \varepsilon = 0.0001$$

Implementation: Gradient Checking

- If $\boldsymbol{\theta}$ is a vector of parameters θ_i,
  - Compute numerical derivative with respect to each θ_i.
    - Create a vector $\mathbf{v}$ that is ε in position i and 0 everywhere else:
      - *How do you do this without a for loop in NumPy?*
    - Compute $G_{\text{num}}(\theta_i) = (J(\boldsymbol{\theta} + \mathbf{v}) - J(\boldsymbol{\theta} - \mathbf{v})) / 2\varepsilon$
  - Aggregate all derivatives $G_{\text{num}}(\theta_i)$ into numerical gradient $G_{\text{num}}(\boldsymbol{\theta})$.

- Compare numerical gradient $G_{\text{num}}(\boldsymbol{\theta})$ with implementation of gradient $G_{\text{imp}}(\boldsymbol{\theta})$:

$$\frac{\left\| G_{num}(\boldsymbol{\theta}) - G_{imp}(\boldsymbol{\theta}) \right\|}{\left\| G_{num}(\boldsymbol{\theta}) + G_{imp}(\boldsymbol{\theta}) \right\|} \leq 10^{-6}$$