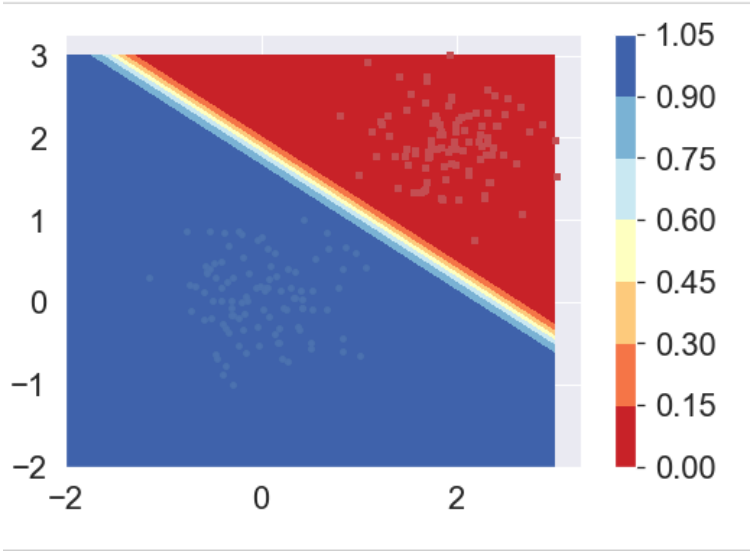
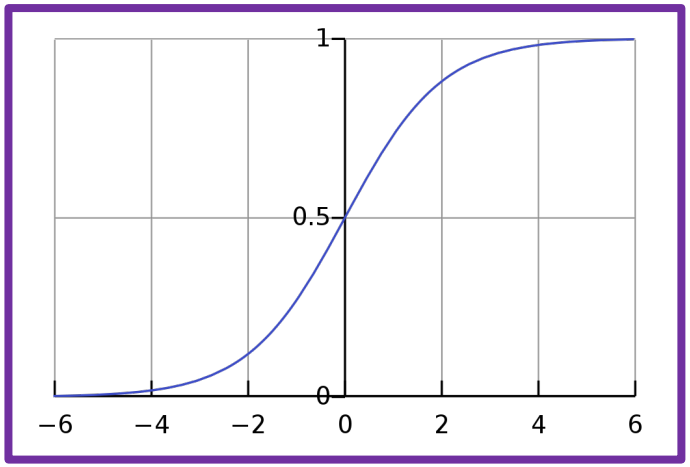
Tufts COMP 135: Introduction to Machine Learning

https://www.cs.tufts.edu/comp/135/2020f/

Logistic Regression





Many slides attributable to:
Erik Sudderth (UCI)
Finale Doshi-Velez (Harvard)
James, Witten, Hastie, Tibshirani (ISL/ESL books)

Prof. Mike Hughes

Objectives Today: Logistic Regression

Logistic Regression

- View as a probabilistic classifier
- Justification: minimize "log loss" is equivalent to maximizing the likelihood of training set
- Computing log loss in numerically stable way
- Computing the gradient wrt parameters
- Training via gradient descent

Check-in Q1:

When training Logistic Regression, we minimize the log loss on the training set.

$$\text{log\_loss}(y, \hat{p}) = -y \log \hat{p} - (1 - y) \log(1 - \hat{p})$$

$$\min_{w,b} \sum_{n=1}^{N} \text{log\_loss}(y_n, \hat{p}(x_n, w, b))$$

Can you provide 2 justifications for why this log loss objective is sensible?

Check-in Q1:

When training Logistic Regression, we minimize the log loss on the training set.

$$\text{log\_loss}(y, \hat{p}) = -y \log \hat{p} - (1 - y) \log(1 - \hat{p})$$

$$\min_{w,b} \sum_{n=1}^{N} \text{log\_loss}(y_n, \hat{p}(x_n, w, b))$$

Can you provide 2 justifications for why this log loss objective is sensible?

1) Log loss is an upper bound of error rate. Minimizing log loss must reduce our (worst case) error rate.

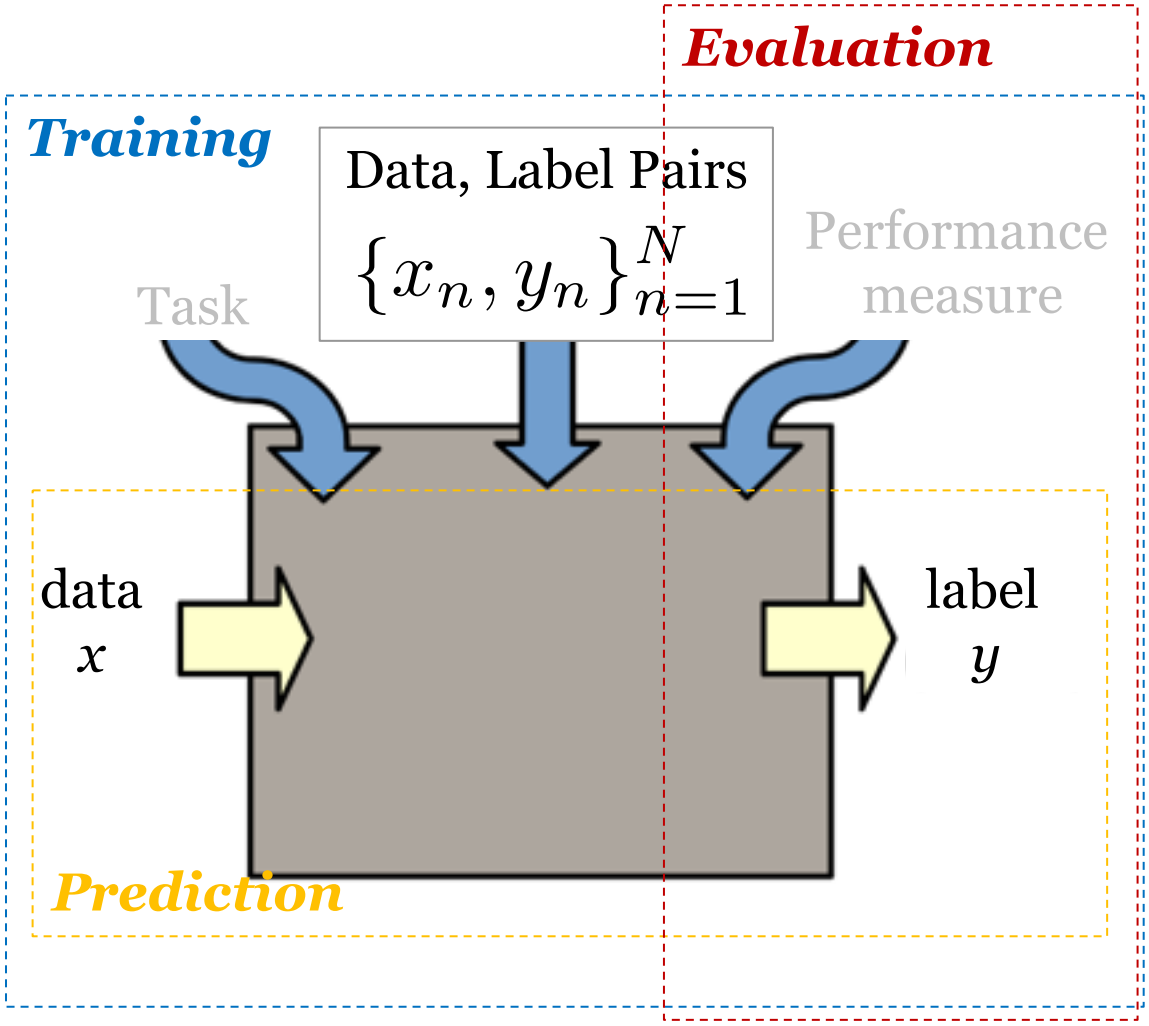
2) Log loss = Binary Cross Entropy. Interpret as learning best probabilistic "encoding' of the training data

What will we learn?

Supervised Learning

Unsupervised Learning

Reinforcement Learning



Task: Binary Classification

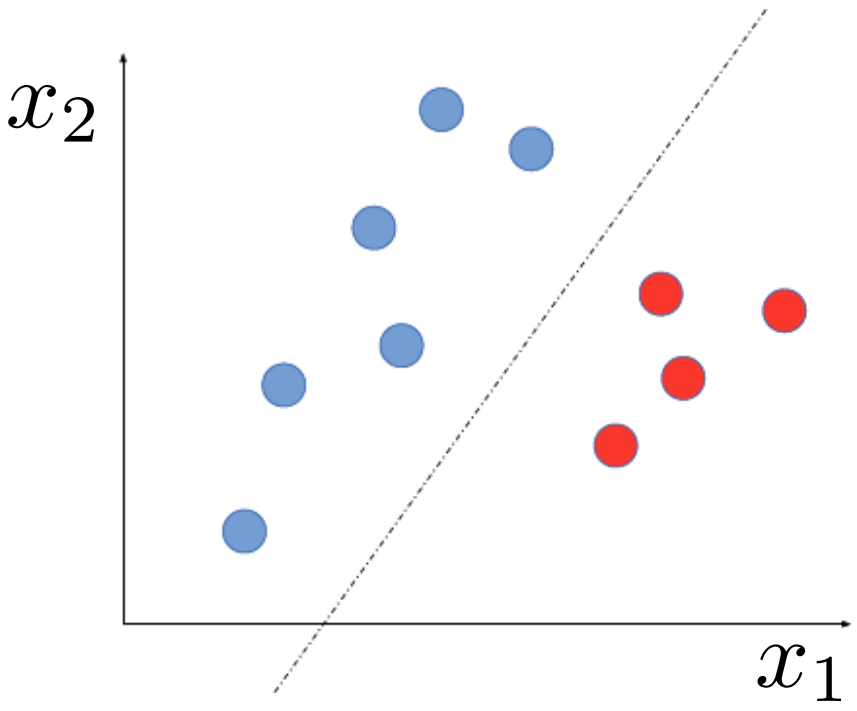
Supervised Learning

binary classification

Unsupervised Learning

Reinforcement Learning

y is a binary variable (red or blue)



Probability Prediction

Goal: Predict probability of label given features

- Input: $x_i \triangleq [x_{i1}, x_{i2}, \ldots x_{if} \ldots x_{iF}]$

  "features"
  "covariates"
  "attributes"

  Entries can be real-valued, or other numeric types (e.g. integer, binary)

- Output: $\hat{p}_i \triangleq p(Y_i = 1 | x_i)$

  "probability"

  Value between 0 and 1
  e.g. 0.001, 0.513, 0.987

```
>>> yproba_N2 = model.predict_proba(x_NF)
>>> yproba1_N = model.predict_proba(x_NF)[:,1]
>>> yproba1_N[:5]
[0.143, 0.432, 0.523, 0.003, 0.994]
```

Review: Logistic Regression

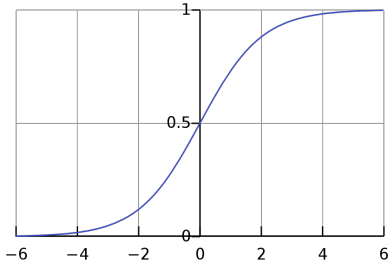
Parameters:

weight vector $w = [w_1, w_2, \ldots w_f \ldots w_F]$

bias scalar b

Prediction:

$$\hat{p}(x_i, w, b) = p(y_i = 1 | x_i) \triangleq \text{sigmoid}\left(\sum_{f=1}^{F} w_f x_{if} + b\right)$$



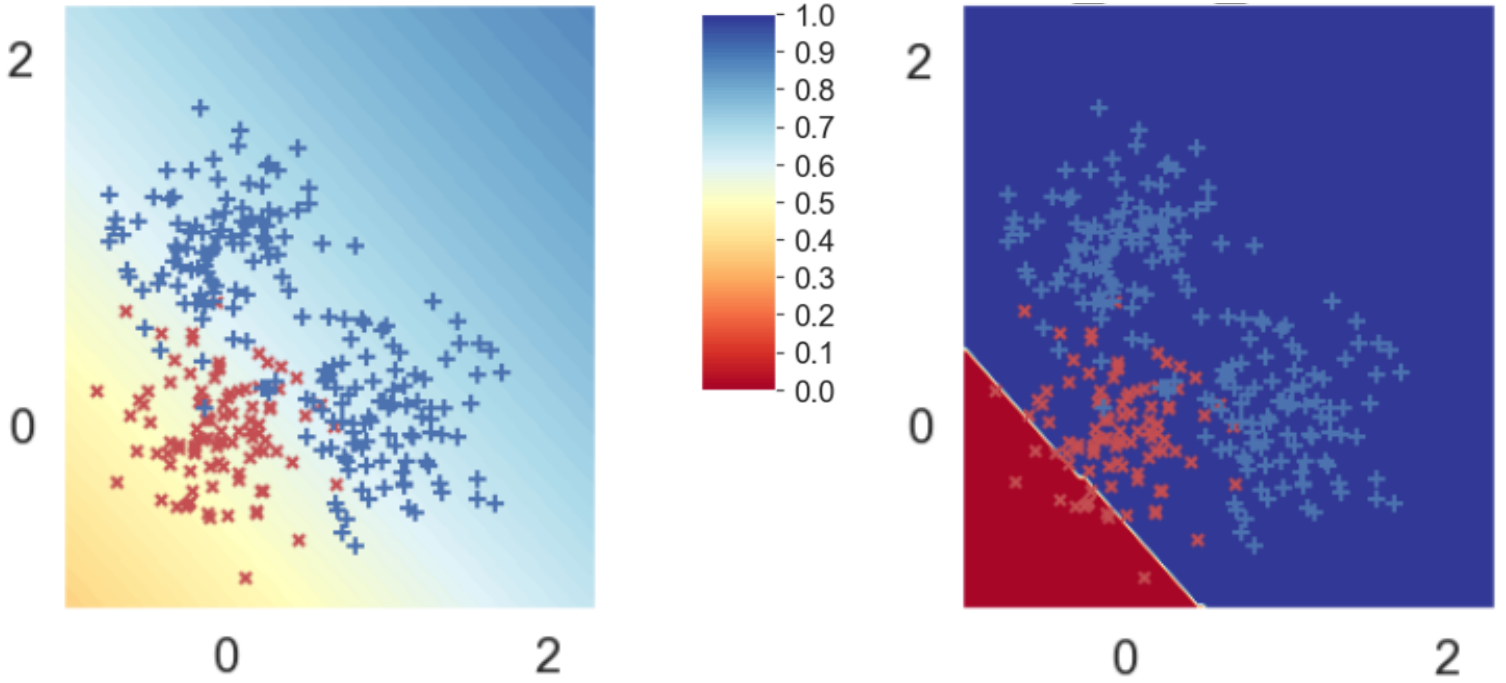
$$\text{sigmoid}(z) = \frac{1}{1 + e^{-z}}$$

Training:

Find weights and bias that minimize (penalized) **log loss**

$$\min_{w,b} \sum_{n=1}^{N} \text{log\_loss}(y_n, \hat{p}(x_n, w, b))$$

Logistic Regression: Predicted Probas vs Binary Decisions



Predicted probability function is monotonically increasing in one direction
*That direction is **perpendicular** to the decision boundary*

Decision boundary is the set of x values where Pr(y=1|x) = 0.5

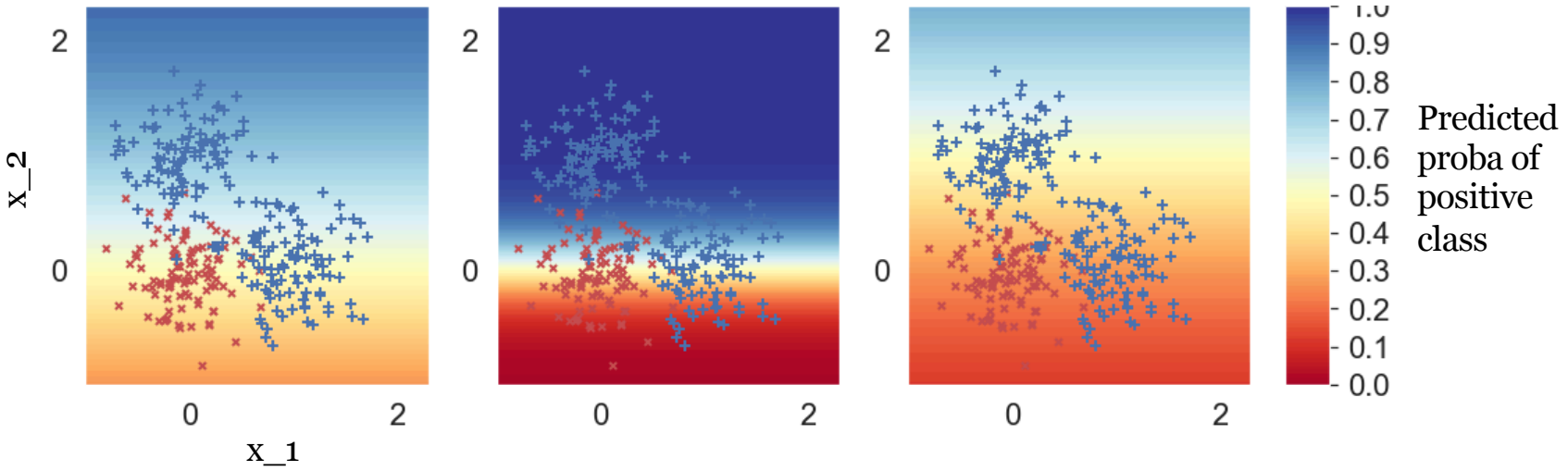
Decision boundary is a linear function of x

Check-in (warm up for lab)

Consider logistic regression classifier for 2D features

What is the value (approximately)

of w_1, w_2, and bias for each plot below?

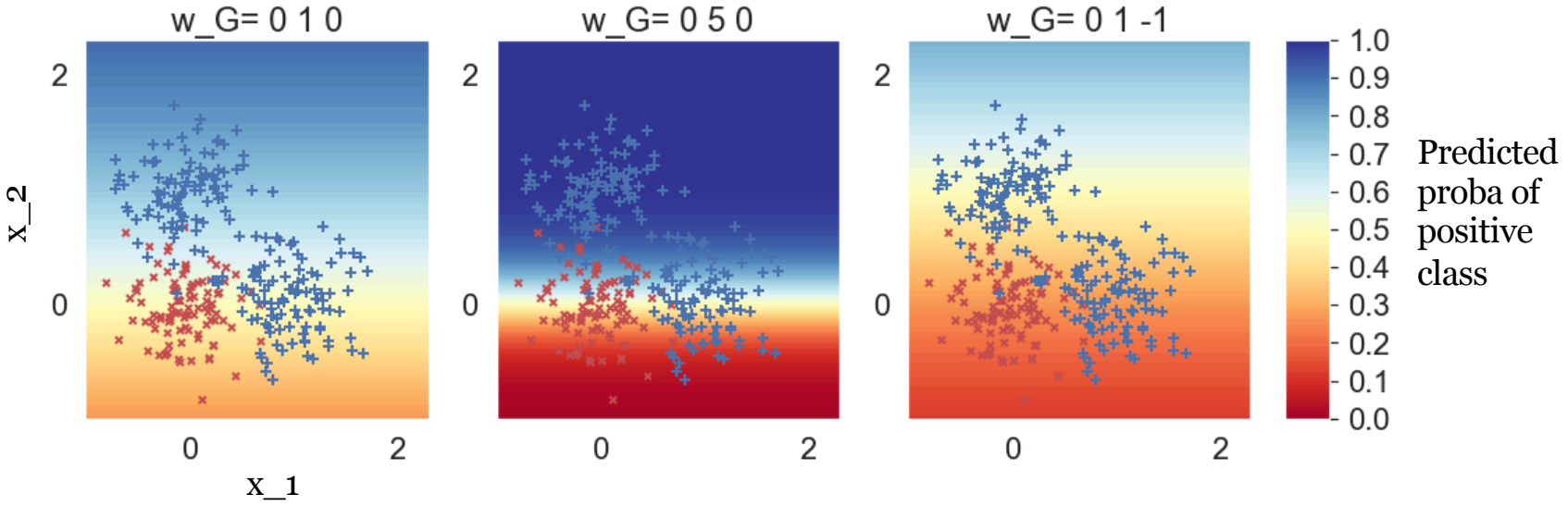


Check-in **Answered**

Consider logistic regression classifier for 2D features

What is the value (approximately)

of w_1, w_2, and bias for each plot below?



Optimization Objective
Why minimize log loss?
A **probabilistic** justification

Likelihood of labels under LR

We can write the probability for each possible outcome as:

$$p(Y_i = 1|x_i) = \sigma(w^T x_i + b)$$
$$p(Y_i = 0|x_i) = 1 - \sigma(w^T x_i + b)$$

We can write the probability mass function of random variable Y as:

$$p(Y_i = y_i|x_i) = \left[\sigma(w^T x_i + b)\right]^{y_i} \left[1 - \sigma(w^T x_i + b)\right]^{1-y_i}$$

Interpret: p(y | x) is the "likelihood" of label y given input features x
Goal: Fit model to make the training data as likely as possible

Maximizing likelihood

$$\max_{w,b} \prod_{n=1}^{N} p(Y_n = y_n | x_n, w, b)$$

Why might this be hard in practice?

Hint: Think about datasets with 1000s of examples N

Maximizing **log** likelihood

The **logarithm** (with any base) is a monotonic transform

a > b implies log (a) > log (b)

Thus, the following are equivalent problems

$$w^*, b^* = \arg\max_{w,b} \prod_{n=1}^{N} p(Y_n = y_n | x_n, w, b)$$

$$w^*, b^* = \arg\max_{w,b} \log \left[\prod_{n=1}^{N} p(Y_n = y_n | x_n, w, b) \right]$$

Log likelihood for LR

We can write the probability mass function of Y as:

$$p(Y_i = y_i | x_i) = \left[\sigma(w^T x_i + b)\right]^{y_i} \left[1 - \sigma(w^T x_i + b)\right]^{1-y_i}$$

Our training objective is to maximize log likelihood

$$w^*, b^* = \arg\max_{w,b} \log \left[\prod_{n=1}^{N} p(Y_n = y_n | x_n, w, b)\right]$$

In order to maximize likelihood, we can minimize negative log likelihood.

Two **equivalent** optimization problems:

$$w^*, b^* = \arg\max_{w,b} \sum_{n=1}^{N} \log p(Y_n = y_n | x_n, w, b)$$

$$w^*, b^* = \arg\min_{w,b} -\sum_{n=1}^{N} \log p(Y_n = y_n | x_n, w, b)$$

Summary of "likelihood" justification for the way we train logistic regression

- We defined a probabilistic model for y given x
- We want to maximize probability of training data under this model ("maximize likelihood")
- We can show that another optimization problem ("maximize log likelihood") is easier numerically but produces the same optimal values for weights and bias
- Equivalent to minimizing -1 * log likelihood
- Turns out, minimizing log loss is precisely the same thing as minimizing negative log likelihood

Computing the loss for Logistic Regression (LR) in a numerically stable way

Simplified notation

- Feature vector with first entry constant

$$\tilde{x}_n = [1 \; x_{n1} \; x_{n2} \; \ldots \; x_{nF}]$$

- Weight vector (first entry is the "bias")

$$w = [w_0 \;\; w_1 \;\; w_2 \ldots w_F]$$

- "Score" value s (real number, -inf to +inf)

$$s_n = w^T \tilde{x}_n$$

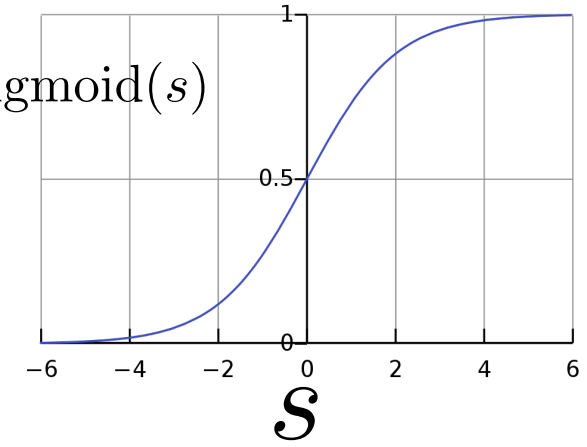
Training Logistic Regression

$$\tilde{x}_n = [1 \; x_{n1} \; x_{n2} \; \ldots \; x_{nF}]$$

$$w = [w_0 \quad w_1 \quad w_2 \ldots w_F]$$

$$s_n = w^T \tilde{x}_n$$

$\sigma(s) = \text{sigmoid}(s)$



$$\min_w \quad \sum_{n=1}^{N} \text{BCE}(y_n, \sigma(s_n(w)))$$

How can we evaluate this loss as a function of w reliably?
How can we evaluate gradients of this loss with respect to w?

Simplifying per-example loss

$$s_n = w^T \tilde{x}_n$$

$$\begin{aligned}
\text{scoreBCE}(y_n, s_n) &= \text{BCE}(y_n, \sigma(s_n)) \\
&= -y_n \log_2 \sigma(s_n) - (1 - y_n) \log_2 (1 - \sigma(s_n)) \\
&= -y_n \log_2 \frac{1}{1 + e^{-s_n}} - (1 - y_n) \log_2 \frac{e^{-s_n}}{1 + e^{-s_n}} \\
&= \begin{cases} \log_2(1 + e^{-s_n}) & \text{if } y_n = 1 \\ \log_2(1 + e^{s_n}) & \text{if } y_n = 0 \end{cases} \\
&= \log_2 \left(1 + e^{\text{flip}(y_n) s_n}\right) \\
&= \log_2 \left(e^0 + e^{\text{flip}(y_n) s_n}\right)
\end{aligned}$$

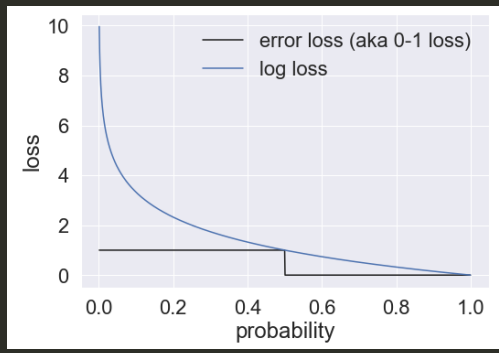
$$\text{flip}(y_n) = \begin{cases} -1 & \text{if } y_n = 1 \\ +1 & \text{if } y_n = 0 \end{cases}$$

Why care about numerical issues?

1) Avoid NaNs and other issues
2) Correctly rank bad solutions (avoid saturation)
3) Non-zero gradients everywhere (no flat regions)

```
# Try some extreme scores (should get worse BCE with more extreme scores)
>>> ans = calc_mean_BCE_from_scores([0], [+99.])
>>> print("%.6f" % (ans))
142.826809
>>> ans = calc_mean_BCE_from_scores([0], [+999.])
>>> print("%.6f" % (ans))
1441.252346
>>> ans = calc_mean_BCE_from_scores([0], [+9999.])
>>> print("%.6f" % (ans))
14425.507714

# Try with "probas": sigmoid saturates! using scores is *better*
>>> a = calc_mean_BCE_from_probas([0], sigmoid([+999.]))
>>> print("%.6f" % (a))
46.508147
>>> a = calc_mean_BCE_from_probas([0], sigmoid([+9999.]))
>>> print("%.6f" % (a))
46.508147
```



logsumexp : the problem

See HW2 on website

In several ML tasks, given a vector $a \in \mathbb{R}$ of L real numbers, we need to frequently evaluate a function like this:

$$\text{logsumexp}([a_1, a_2, \ldots a_L]) = \log\left(e^{a_1} + e^{a_2} + \ldots e^{a_L}\right)$$

We call this the "logsumexp" function.

If we just implement this in NumPy code, we'll quickly notice problems for some inputs that should be "easy".

For example, consider the case where our vector a has two entries: `a = [1000, -999]`.

$$\text{logsumexp}([1000.0, -999]) = \log\left(e^{1000} + e^{-999}\right) = 1000 + \epsilon$$

where we use ϵ to indicate a *very* small positive float value. The true result is not quite 1000, but probably so close that our finite computers won't be able to represent the difference.

But what does a quick implementation in NumPy do?

```
# BAD! Naive computation overflows
>>> import numpy as np
>>> np.log(np.sum(np.exp([1000.0, -999.])))
RuntimeWarning: overflow encountered in exp
inf

# GOOD: a "numerically stable" computation from a good library
>>> from scipy.special import logsumexp
>>> logsumexp([1000., -999.])
1000.0
```

logsumexp : the solution

See HW2 on website

$$\begin{aligned}
\text{logsumexp}([a_1, a_2, \ldots a_L]) &= \log\left(e^{a_1} + e^{a_2} + \ldots e^{a_L}\right) \\
&= \log\left(e^{m}(e^{a_1 - m} + e^{a_2 - m} + \ldots e^{a_L - m})\right) \\
&= \log\left(e^{m}\right) + \log\left(e^{a_1 - m} + e^{a_2 - m} + \ldots e^{a_L - m}\right) \\
&= m + \log\left(e^{a_1 - m} + e^{a_2 - m} + \ldots e^{a_L - m}\right)
\end{aligned}$$

The above is true for *any* scalar real value m. However, if we pick m smartly, this computation becomes more robust to any possible input values in our vector a.

If we select m equal to the maximum entry of the vector a ($m = \max_\ell a_\ell$), we can guarantee that each of these right-hand side terms indexed by ℓ satisfy: $0 \leq e^{a_\ell - m} \leq 1$. Furthermore, at least one of these terms equals 1.0 precisely. Thus, the total sum inside the log must be between 1 and L. We can easily compute `np.log(1.0)` or `np.log(L)` and values in between for any reasonable value of L (even if our vector has a few billion entries).

Gradient of log loss wrt weights

Per-example loss **when y=0**

$$s_n = w^T \tilde{x}_n$$

$$J(s_n(w)) = \log(1 + e^{s_n})$$

Gradient w.r.t. weight on feature f: Use the **chain rule**

$$\frac{\partial}{\partial w_f} J(s_n(w)) = \frac{\partial}{\partial s_n} J(s_n) \cdot \frac{\partial}{\partial w_f} s_n(w)$$

Simplifying yields:

$$\frac{\partial}{\partial w_f} J = (\sigma(s_n(w)) - y_n)\, \tilde{x}_{nf}$$

Gradient of log loss wrt weights

$$s_n = w^T \tilde{x}_n$$

$$\frac{\partial}{\partial w_f} J = \left(\sigma\left(s_n(w)\right) - y_n\right) x_{nf}$$

Nice interpretation: (PREDICTION - TRUE RESPONSE) * FEATURE

If feature is positive:

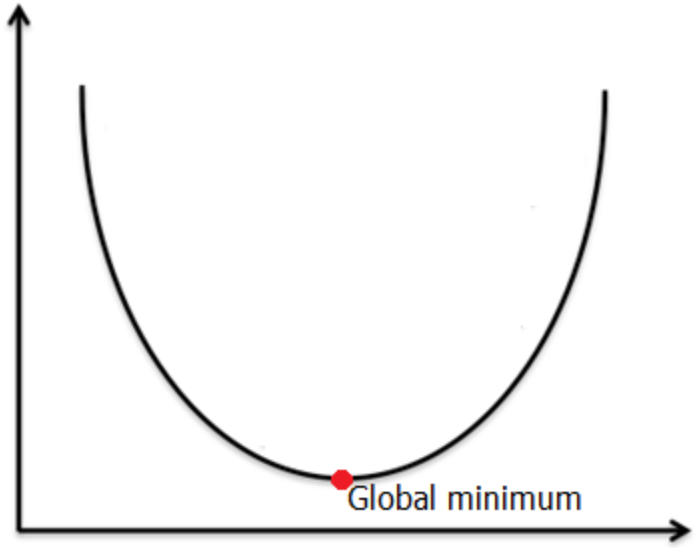
Predicted probability is larger than truth means gradient is positive

To improve our model (go downhill) we should make that weight smaller

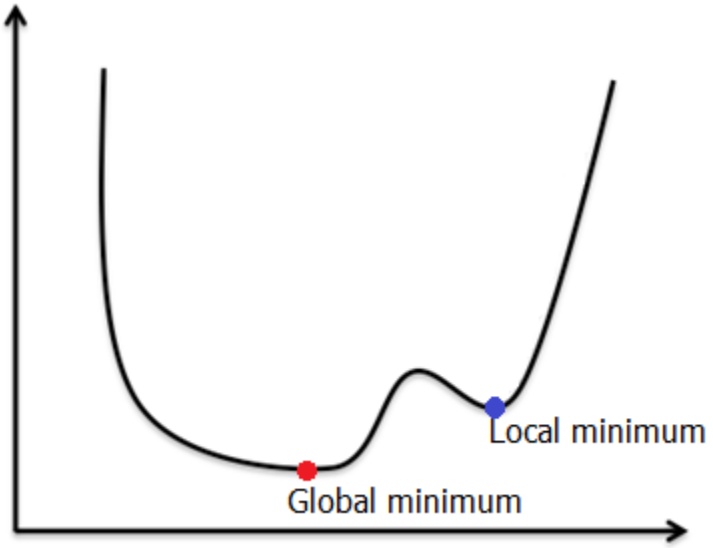
If feature is negative, we move weights in the opposite direction.

Gradient descent for Logistic Regression

Will gradient descent always find same solution?

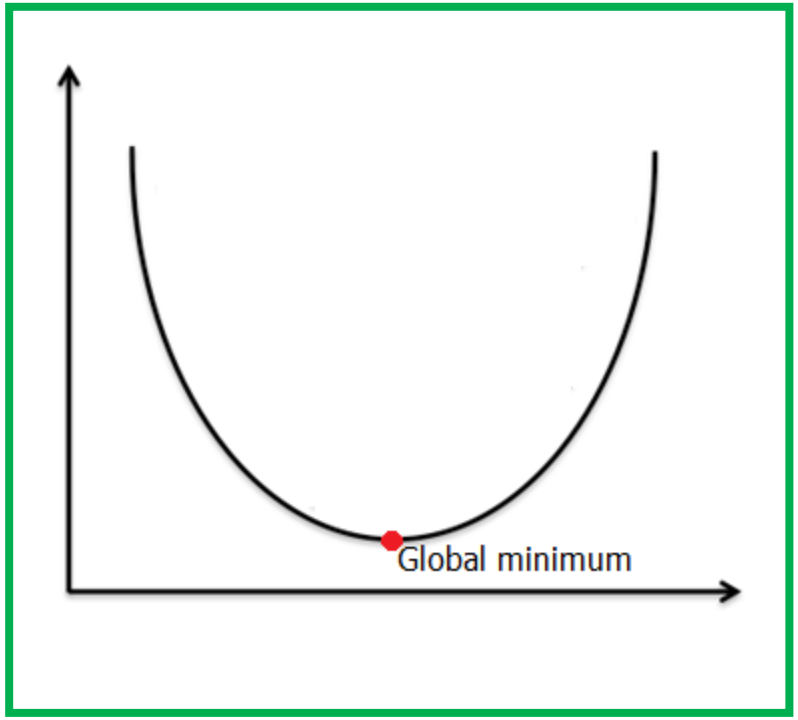


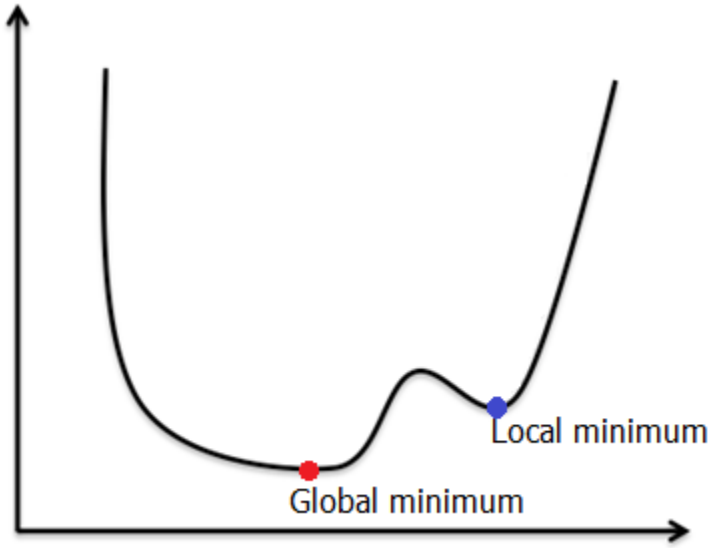
Yes, if loss looks like this



Not if multiple local minima exist

Log loss (aka BCE) is **convex** as a function of weight parameters





Objectives Today: Logistic Regression

Logistic Regression

- View as a probabilistic classifier
- Justification: minimize "log loss" is equivalent to maximizing the likelihood of training set
- Computing log loss in numerically stable way
- Computing the gradient
- Training via gradient descent

Lab: practice visual intuition for how weights and bias affect predicted probabilities