CS 232: Artificial Intelligence

Spring 2024

Prof. Carolyn Anderson
Wellesley College

Reminders

- HW 5 due Monday, 3/11
- My help hours today: 3:30-4:30
- Lyra's help hours: Sunday 4-6
- Read YLLATAILY Chapter 9-10 for Tuesday

Recap

The two phases of logistic regression

Training: we learn weights w and b using **stochastic gradient descent** and **cross-entropy loss**.

Test: Given a test example x we compute $p(y|x)$ using learned weights w and b, and return whichever label ($y = 1$ or $y = 0$) is higher probability

Classification in (binary) logistic regression: summary

Given:

- a set of classes: (+ sentiment,- sentiment)
- a vector **x** of features `[x1, x2, …, xn]`
  - x1= count("awesome")
  - x2 = log(number of words in review)
- A vector **w** of weights `[w1, w2, …, wn]`
  - w_i for each feature f_i

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

Multi-class Regression

Multinomial Logistic Regression

Often we need more than 2 classes

- Positive/negative/neutral
- Parts of speech (noun, verb, adjective, adverb, preposition, etc.)
- Classify emergency SMSs into different actionable classes

If >2 classes we use **multinomial logistic regression**

= Softmax regression

= Multinomial logit

= Maximum entropy modeling or MaxEnt

So "logistic regression" means binary (2 classes)

Multinomial Logistic Regression

The probability of everything must still sum to 1

```
P(positive|doc) + P(negative|doc) +
P(neutral|doc) = 1
```

Need a generalization of the sigmoid called **softmax**

- Takes a vector $z = [z1, z2, ..., zk]$ of k arbitrary values
- Outputs a probability distribution

The softmax function

Turns a vector $z = [z_1, z_2, \ldots, z_k]$ of k arbitrary values into probabilities

$$p(y=c \mid x) = \frac{\exp(w_c \cdot x + b_c)}{\sum_{j=1}^{k} \exp(w_j \cdot x + b)}$$

$$z_c = w_c \cdot x + b_c$$

Class-specific weights

The softmax function

Turns a vector $z = [z_1, z_2, \ldots, z_k]$ of k arbitrary values into probabilities :

$$\begin{array}{cccccc} z_{app} & z_{main} & z_{dessert} & z_{soup} & z_{salad} & z_{bev} \end{array}$$

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

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

$$[0.055, 0.090, 0.006, 0.099, 0.74, 0.010]$$

$$p(app \mid x, w, b) = \frac{\exp(z_{app})}{\sum_c z_c}$$

Softmax in multinomial logistic regression

$$p(y=c|x) = \frac{\exp(w_c \cdot x + b_c)}{\sum_{j=1}^{K} \exp(w_j \cdot x + b_j)}$$

Input is still the dot product between weight vector *w* and input vector *x*, but now we need separate weight vectors for each of the *K* classes.

Features in binary versus multinomial logistic regression

Binary: positive weight

$$x_5 = \begin{cases} 1 & \text{if “!”} \in \text{doc} \\ 0 & \text{otherwise} \end{cases} \qquad w_5 = 3.0$$

Multinominal: separate weights for each class:

Feature	Definition	$w_{5,+}$	$w_{5,-}$	$w_{5,0}$
$f_5(x)$	$\begin{cases} 1 & \text{if “!”} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$	3.5	3.1	-5.3

How Does Learning Work?

Slides borrowed from Jurafsky & Martin Edition 3

Learning components

A loss function: Last Class

- **cross-entropy loss**

Today

An optimization algorithm:

- **stochastic gradient descent**

Learning in Supervised Classification

Supervised classification:

- We know the correct label y (either 0 or 1) for each x.
- But what the system produces is an estimate, $\hat{y}$

We want to set w and b to minimize the **distance** between our estimate $\hat{y}^{(i)}$ and the true $y^{(i)}$.

- We need a distance estimator: a **loss function** or a **cost function**
- We need an optimization algorithm to update w and b to minimize the loss.

Loss Function

Intuition of negative log likelihood loss (cross-entropy loss):

We choose the parameters *w*,*b* that maximize

- the log probability
- of the true *y* labels in the training data
- given the observations *x*

=

Minimize the negative log probability

Loss Function

Goal: maximize probability of the correct label $p(y|x)$

Since there are only 2 discrete outcomes (0 or 1) we can express the probability $p(y|x)$ from our classifier as:

$$p(y|x) = \hat{y}^y (1-\hat{y})^{1-y}$$

noting:

if y=1, this simplifies to $\hat{y}$

if y=0, this simplifies to 1– $\hat{y}$

Now take the log of both sides (mathematically handy)

Maximize:

$$\begin{aligned}\log p(y|x) &= \log\left[\hat{y}^y (1-\hat{y})^{1-y}\right] \\ &= \underline{y \log \hat{y}} + \underline{(1-y)\log(1-\hat{y})}\end{aligned}$$

Loss Function

Goal: maximize probability of the correct label $p(y|x)$

Minimize the cross-entropy loss

Minimize: $L_{\mathrm{CE}}(\hat{y}, y) = -\log p(y|x) \;=\; -[y \log \hat{y} + (1-y)\log(1-\hat{y})]$

$$L_{\mathrm{CE}}(\hat{y}, y) \;=\; -\left[y \log \sigma(w \cdot x + b) + (1-y) \log\left(1 - \sigma(w \cdot x + b)\right)\right]$$

For multi-class regression: $L_{\mathrm{CE}}(\hat{\mathbf{y}}, \mathbf{y}) \;=\; -\sum_{k=1}^{K} y_k \log \hat{y}_k$

Learning components

A loss function:

- **cross-entropy loss**

An optimization algorithm:

- **stochastic gradient descent**

Stochastic Gradient Descent

Game

Optimization

diff. between $\hat{y}$ & y as small as possible

Goal: find weights + bias (W,b) that minimize loss.

Let's make explicit that the loss function is parameterized by weights θ=(w,b)

$\theta = (W, b)$ $W = \{w_f$ for every feature

- And we'll represent $\hat{y}$ as $f(x; \theta)$ to make the dependence on θ more obvious

$f(x;\theta)$

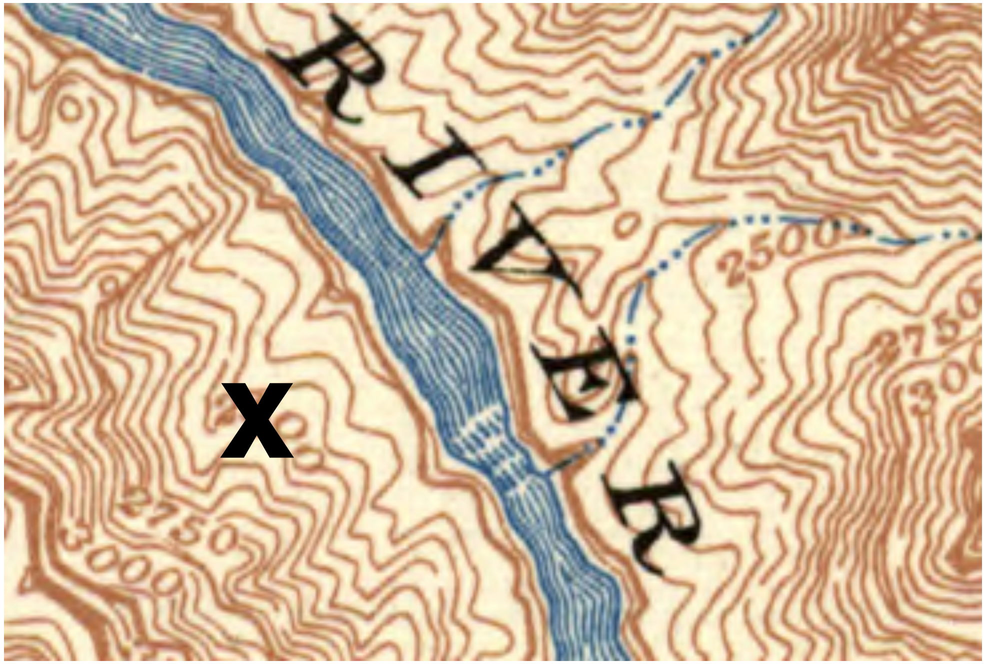
We want the weights that minimize the loss, averaged over all examples:

$$\hat{\theta} = \text{argmin} \frac{1}{m} \sum_{i=1}^{m} L_{CE}(\underbrace{f(x^i; \theta)}_{\hat{y}}, y^i)$$

For m recipes, x^i is the ith recipe and y^i is its label.

Intuition of gradient descent

How do I get to the bottom of this river canyon?



Look around me 360°

Find the direction of steepest slope down

Go that way

Our goal: minimize the loss

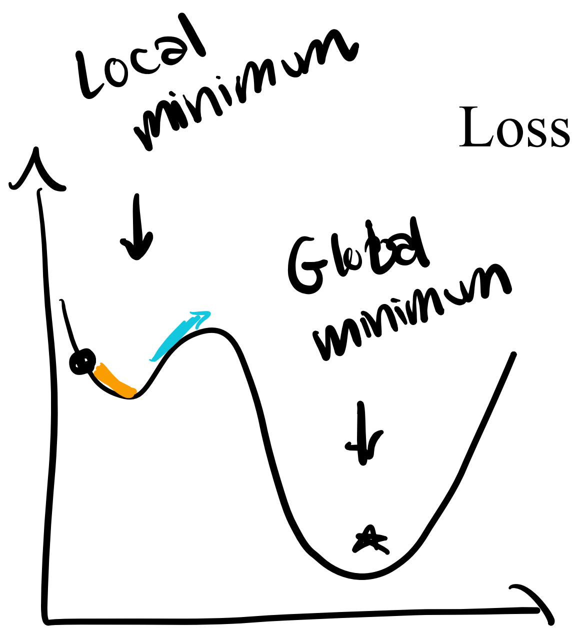
For logistic regression, loss function is **convex**

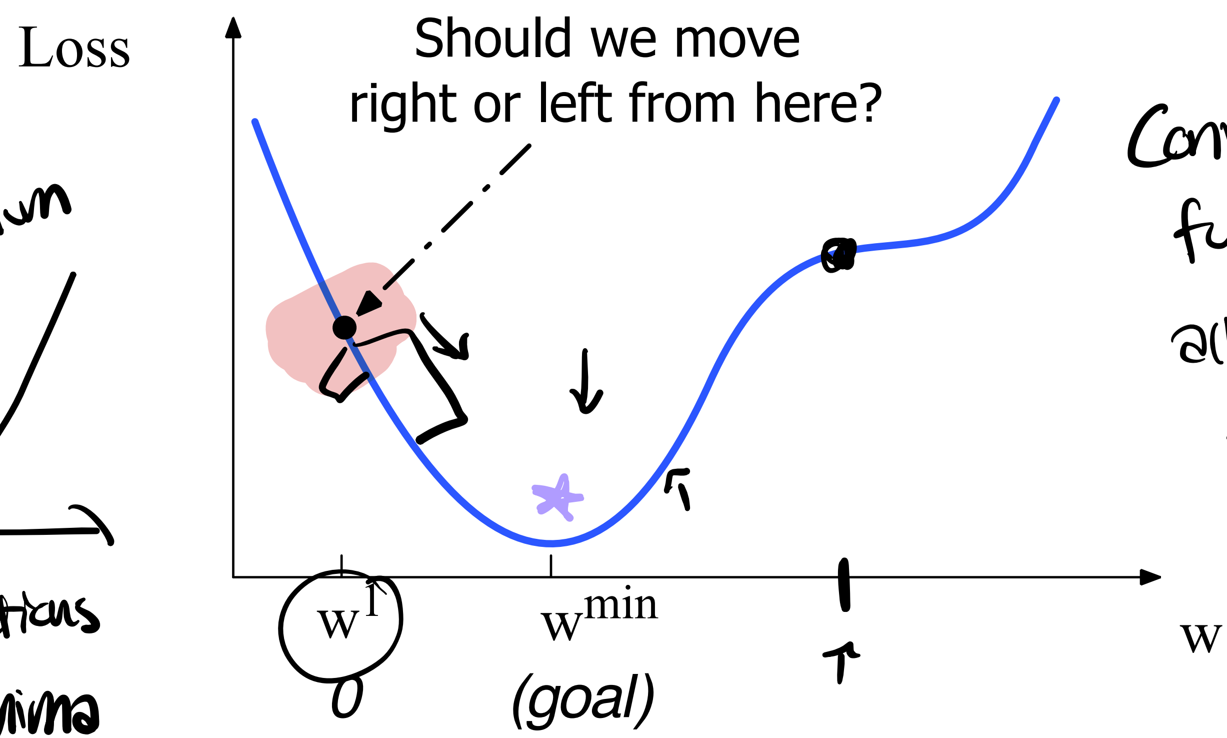
- A convex function has just one minimum
- Gradient descent starting from any point is guaranteed to find the minimum
    - (Loss for neural networks is non-convex)

Let's first visualize for a single scalar w

Q: Given current w, should we make it bigger or smaller?

A: Move *w* in the reverse direction from the slope of the function





Convex functions allow us to find global minima

Concave functions hide global minima

Gradients

The **gradient** of a function of many variables is a vector pointing in the direction of the greatest increase in a function.

Gradient Descent: Find the gradient of the loss function at the current point and move in the **opposite** direction.

How much do we move in that direction ?

- The value of the gradient (slope) weighted by a **learning rate** η
- Higher learning rate means we make a bigger change to *w* at each step

Gradient: $\frac{\partial L(f(x;w),y)}{\partial w}$

Weight update: $w^{t+1} = w^{t} - \eta \frac{\partial L(f(x;w),y)}{\partial w}$

learning rate (eta)

Now let's consider N dimensions

We want to know where in the N-dimensional space (for *N* parameters in θ) we should go.

The **gradient** is a **vector** that expresses the directional components of the sharpest slope along each of the *N* dimensions.

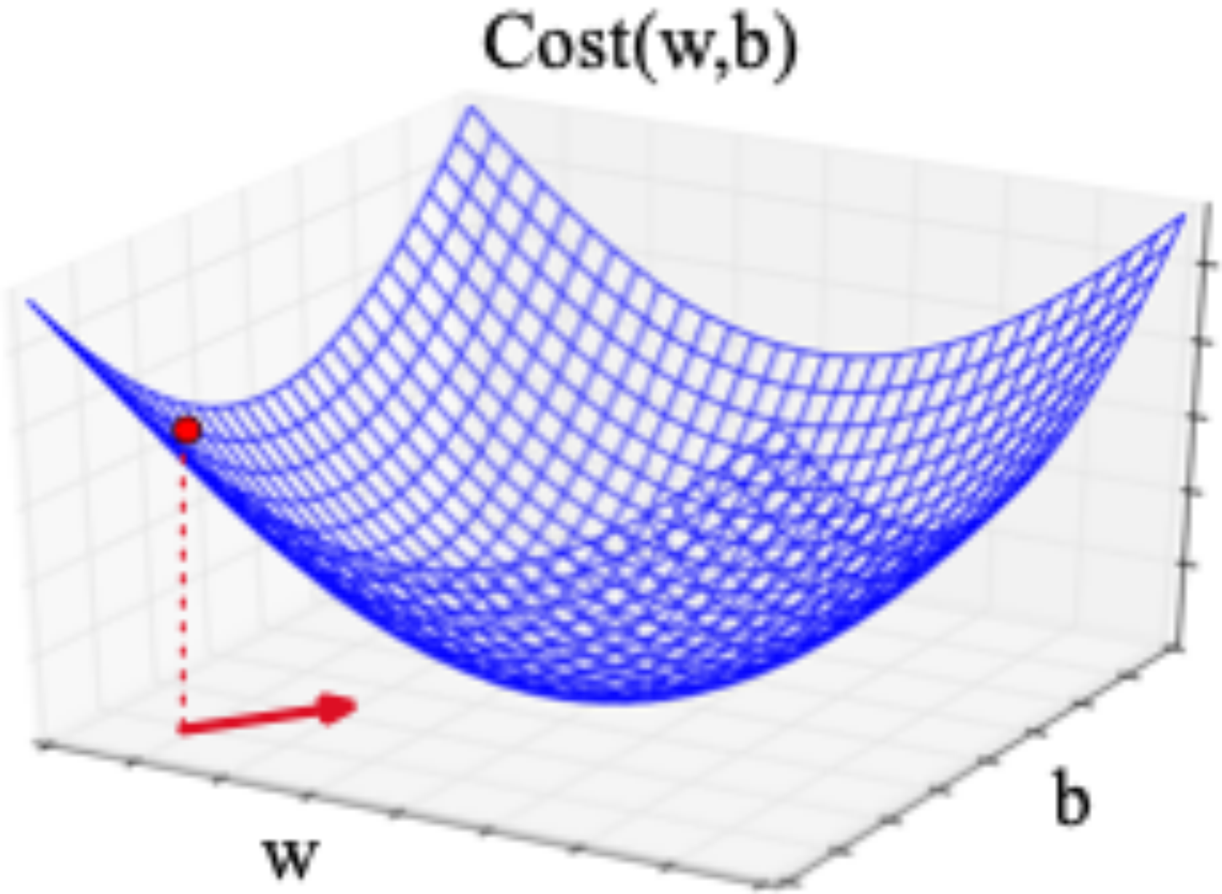
$$\frac{\partial L(f(x;\theta),y)}{\partial \theta} = \left[\frac{\partial L(f(x;w_i),y)}{\partial w_i}, \; \ldots, \frac{\partial L(f(x;w),y)}{\partial w_x} \right]$$

for x weights in W

Two dimensions: w and b

Visualizing the gradient vector at the red point

It has two dimensions shown in the x-y plane



function STOCHASTIC GRADIENT DESCENT($L()$, $f()$, x, y) **returns** θ

  # where: L is the loss function

  #   f is a function parameterized by θ

  #   x is the set of training inputs $x^{(1)}, x^{(2)}, \ldots, x^{(m)}$

  #   y is the set of training outputs (labels) $y^{(1)}, y^{(2)}, \ldots, y^{(m)}$

$\theta \leftarrow 0$

repeat til done   # see caption

for each recipe

  For each training tuple $(x^{(i)}, y^{(i)})$ (in random order)

    1. Optional (for reporting):   # How are we doing on this tuple?

make a guess

      Compute $\hat{y}^{(i)} = f(x^{(i)}; \theta)$ # What is our estimated output $\hat{y}$?

check guess

      Compute the loss $L(\hat{y}^{(i)}, y^{(i)})$ # How far off is $\hat{y}^{(i)}$ from the true output $y^{(i)}$?

compute gradient g

    2. $g \leftarrow \nabla_{\theta} L(f(x^{(i)}; \theta), y^{(i)})$ # How should we move θ to maximize loss?

    3. $\theta \leftarrow \theta - \eta\, g$ # Go the other way instead

return θ

update weight by discounting gradient & subtracting from current weight setting

Once per every item in dataset = epoch

Hyperparameters

We set this ourselves

The learning rate η is a **hyperparameter**

- too high: the learner will take big steps and overshoot
- too low: the learner will take too long

Hyperparameters:

- Briefly, a special kind of parameter for an ML model
- Instead of being learned by algorithm from supervision (like regular parameters), they are chosen by algorithm designer.

Overfitting

A model that perfectly match the training data has a problem.

It will 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

Useful or harmless features

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

+

X1 = "this"
X2 = "movie
X3 = "hated"
X4 = "drew me in"

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

-

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

"Memorizing" the training data can cause problems

Overfitting

4-gram model on tiny data will just memorize the data

- 100% accuracy on the training set

But it will be surprised by the novel 4-grams in the test data

- Low accuracy on test set

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?
  - Regularization in logistic regression
  - Dropout in neural networks

Logistic Regression Example: Pet Picture Classification

Building a Model

Tensorflow is a Python library, but most functions are implemented in C (so they are fast!).

Tensorflow provides useful abstractions for models:

- **tensor**: n-dimensional container for data
- **layer**: apply functions to an input tensor of n dimensions to produce an output tensor of m dimensions.
- **model**: consist of layers connected together
- **optimizer**: an optimization function used to computes weight updates

Training

Train the model

```
epochs = 50

callbacks = [
    keras.callbacks.ModelCheckpoint("save_at_{epoch}.h5"),
]
model.compile(
    optimizer=keras.optimizers.Adam(1e-3),  ↙ learning rate
    loss="binary_crossentropy",
    metrics=["accuracy"],
)
model.fit(
    train_ds, epochs=epochs, callbacks=callbacks, validation_data=val_ds,
)
```