Natural Language Processing

Logistic Regression

Yulia Tsvetkov

yuliats@cs.washington.edu

PAUL G. ALLEN SCHOOL OF COMPUTER SCIENCE & ENGINEERING

Announcements

- HW1 deadline on Friday
- Extra OHs by TAs
- Yulis’a OHs are cancelled this week
- FAQ on HW1 on Ed
- Quiz next Wed - Zipfs law, LR

Components of a probabilistic machine learning classifier

Given m input/output pairs $(x^{(i)}, y^{(i)})$:

1. A **feature representation** for the input. For each input observation $x^{(i)}$, a vector of features $[x_1, x_2, \ldots, x_n]$. Feature j for input $x^{(i)}$ is x_j, more completely $x_1^{(i)}$, or sometimes $f_j(x)$.

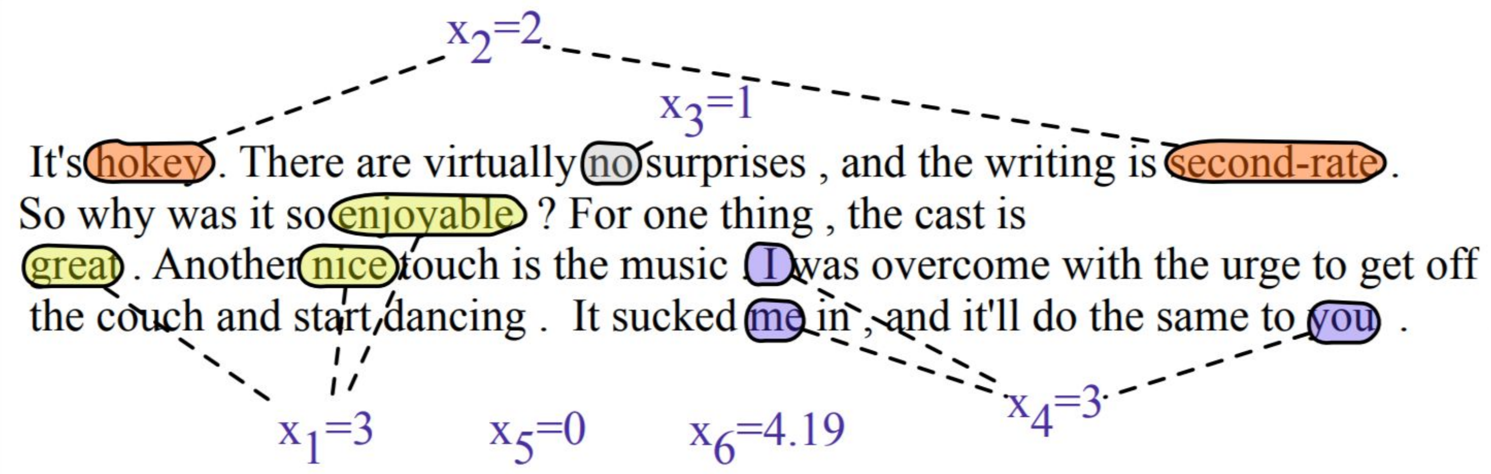
2. A **classification function** that computes $\hat{y}$ the estimated class, via $p(y|x)$, like the **sigmoid** functions

3. An **objective function** for learning, like **cross-entropy loss**

4. An algorithm for **optimizing** the objective function: **stochastic gradient descent**

Sentiment example: 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
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	$\log(\text{word count of doc})$	$\ln(66) = 4.19$

Classifying sentiment for input x

Var	Definition	Value
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

Cross-entropy loss for a single observation x

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

Maximize:
$$\log p(y|x) = \log\left[\hat{y}^{y}(1-\hat{y})^{1-y}\right]$$
$$= y\log\hat{y} + (1-y)\log(1-\hat{y})$$

Now flip sign to turn this into a **cross-entropy loss**: something to minimize

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

Or, plug in definition of $\hat{y} = \sigma(w \cdot x + b)$

$$\boxed{L_{CE}(\hat{y},y) = -[y\log\sigma(\mathbf{w}\cdot\mathbf{x}+b) + (1-y)\log(1-\sigma(\mathbf{w}\cdot\mathbf{x}+b))]}$$

Let's see if this works for our sentiment example

We want loss to be:

- smaller if the model estimate $\hat{y}$ is close to correct
- bigger if model is confused

Let's first suppose the true label of this is $y=1$ (positive)

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 .

Let's see if this works for our sentiment example

True value is y=1 (positive). How well is our model doing?

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

Pretty well! What's the loss?

$$\begin{aligned} L_{\text{CE}}(\hat{y}, y) &= -[y \log \sigma(\mathbf{w} \cdot \mathbf{x} + b) + (1-y)\log(1 - \sigma(\mathbf{w} \cdot \mathbf{x} + b))] \\ &= -[\log \sigma(\mathbf{w} \cdot \mathbf{x} + b)] \\ &= -\log(.70) \\ &= .36 \end{aligned}$$

Let's see if this works for our sentiment example

Suppose the true value instead was y=0 (negative).

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

What's the loss?

$$
\begin{aligned}
L_{\mathrm{CE}}(\hat{y}, y) &= -[y \log \sigma(\mathbf{w} \cdot \mathbf{x} + b) + (1-y) \log (1 - \sigma(\mathbf{w} \cdot \mathbf{x} + b))] \\
&= -[\log (1 - \sigma(\mathbf{w} \cdot \mathbf{x} + b))] \\
&= -\log (.30) \\
&= 1.2
\end{aligned}
$$

Let's see if this works for our sentiment example

The loss when the model was right (if true y=1)

$$
\begin{aligned}
L_{CE}(\hat{y},y) &= -[y\log\sigma(\mathbf{w}\cdot\mathbf{x}+b)+(1-y)\log(1-\sigma(\mathbf{w}\cdot\mathbf{x}+b))] \\
&= -[\log\sigma(\mathbf{w}\cdot\mathbf{x}+b)] \\
&= -\log(.70) \\
&= .36
\end{aligned}
$$

The loss when the model was wrong (if true y=0)

$$
\begin{aligned}
L_{CE}(\hat{y},y) &= -[y\log\sigma(\mathbf{w}\cdot\mathbf{x}+b)+(1-y)\log(1-\sigma(\mathbf{w}\cdot\mathbf{x}+b))] \\
&= -[\log(1-\sigma(\mathbf{w}\cdot\mathbf{x}+b))] \\
&= -\log(.30) \\
&= 1.2
\end{aligned}
$$

Sure enough, loss was bigger when model was wrong!

Learning components

A loss function:

- **cross-entropy loss**

An optimization algorithm:

- **stochastic gradient descent**

Stochastic Gradient Descent

- Stochastic Gradient Descent algorithm
  - is used to optimize the weights
  - for logistic regression
  - also for neural networks

Our goal: minimize the loss

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

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

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

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} \frac{1}{m} \sum_{i=1}^{m} L_{\mathrm{CE}}(f(x^{(i)};\theta), y^{(i)})$$

$$L_{\mathrm{CE}}(\hat{y}, y)$$

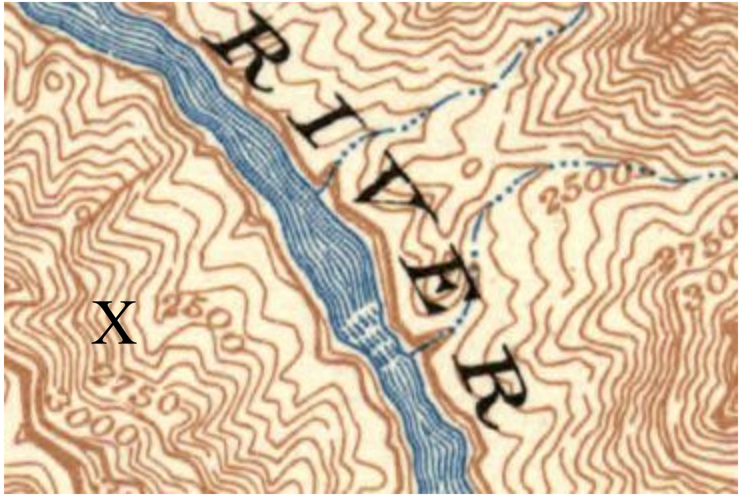
Intuition of gradient descent

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

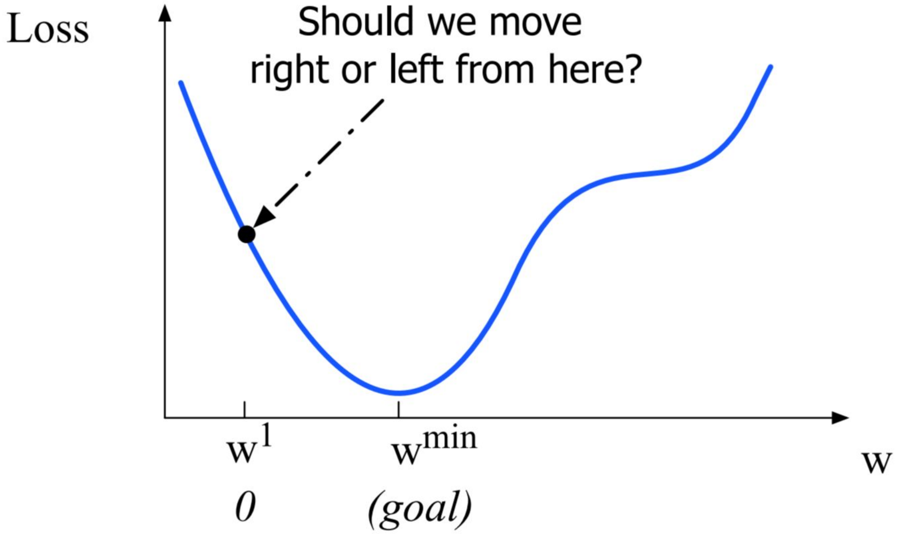
For logistic regression, loss function is **convex**

- 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?

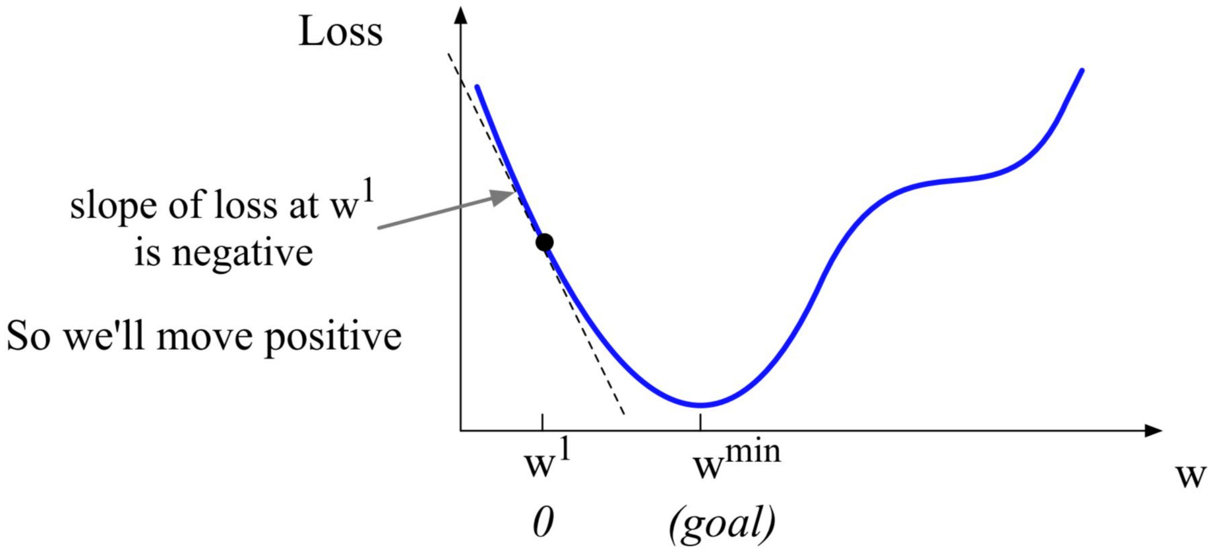
A: Move w in the reverse direction from the slope of the function



Let's first visualize for a single scalar w

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

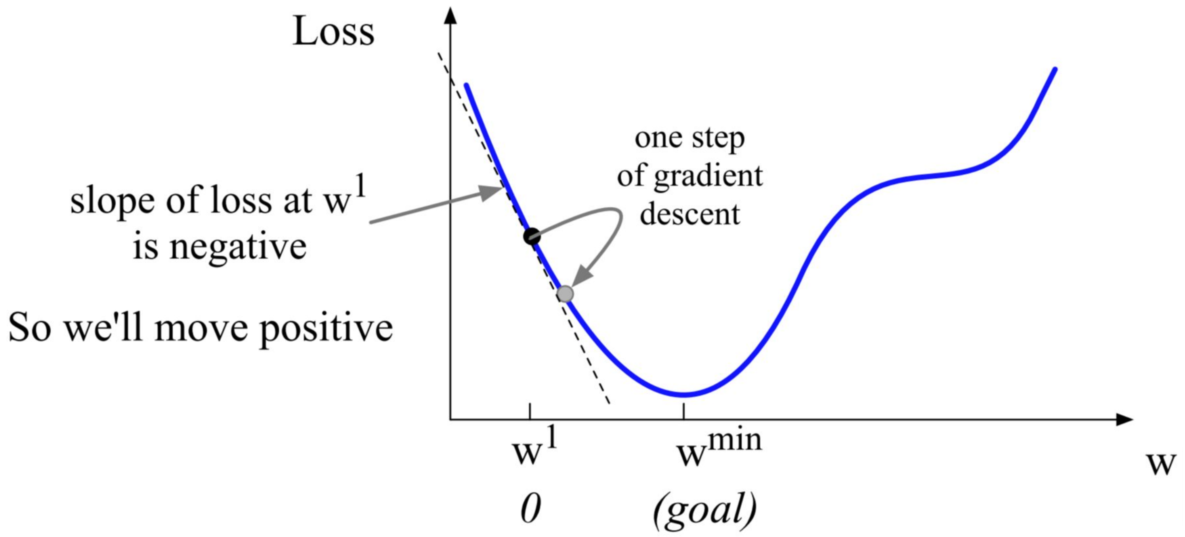
A: Move w in the reverse direction from the slope of the function



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



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 in our example) $\frac{d}{dw}L(f(x;w),y)$
  - weighted by a learning rate η

- Higher learning rate means move w faster

$$w^{t+1} = w^t - \eta \frac{d}{dw}L(f(x;w),y)$$

Now let's consider N dimensions

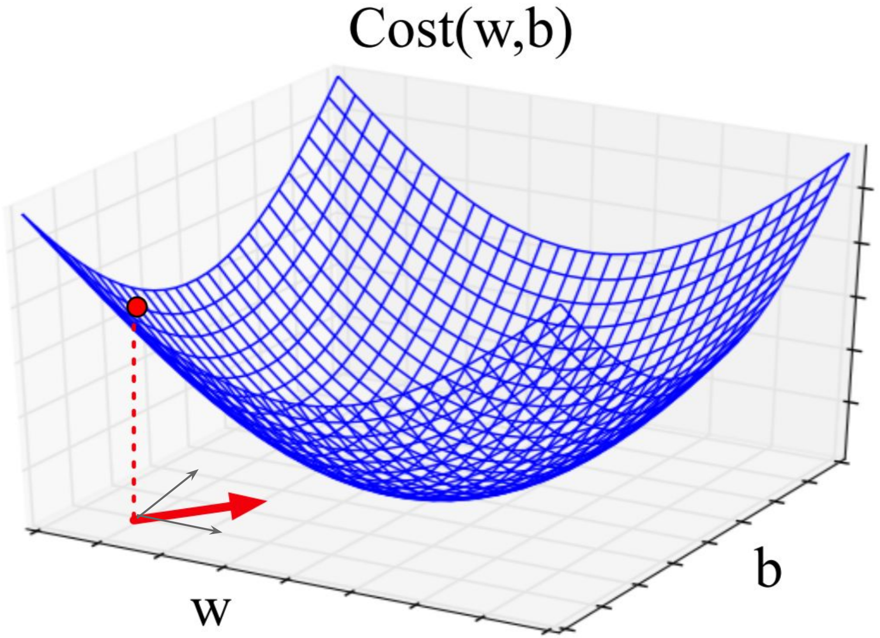
We want to know where in the N-dimensional space (of the N parameters that make up θ) we should move.

The gradient is just such a vector; it expresses the directional components of the sharpest slope along each of the N dimensions.

Imagine 2 dimensions, w and b

Visualizing the gradient vector at the red point

It has two dimensions shown in the x-y plane



Real gradients

Are much longer; lots and lots of weights

For each dimension w_i the gradient component i tells us the slope with respect to that variable.

- "How much would a small change in w_i influence the total loss function L?"
- We express the slope as a partial derivative ∂ of the loss ∂w_i $\frac{\partial}{\partial w_i}$

The gradient is then defined as a vector of these partials.

The gradient

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

$$\nabla_\theta L(f(x;\theta),y)) = \begin{bmatrix} \frac{\partial}{\partial w_1} L(f(x;\theta),y) \\ \frac{\partial}{\partial w_2} L(f(x;\theta),y) \\ \vdots \\ \frac{\partial}{\partial w_n} L(f(x;\theta),y) \end{bmatrix}$$

The final equation for updating θ based on the gradient is thus:

$$\theta_{t+1} = \theta_t - \eta \nabla L(f(x;\theta),y)$$

What are these partial derivatives for logistic regression?

The loss function

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

The elegant derivative of this function (see Section 5.10 for the derivation)

$$\frac{\partial L_{\mathrm{CE}}(\hat{y},y)}{\partial w_j} = [\sigma(w\cdot x+b)-y]x_j$$

$$= (\hat{y}-y)\mathbf{x}_j$$

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

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

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

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

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

  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

Hyperparameters

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.

Mini-batch training

Stochastic gradient descent chooses a single random example at a time.

That can result in choppy movements

More common to compute gradient over batches of training instances.

Batch training: entire dataset

Mini-batch training: m examples (512, or 1024)

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**

Regularization

A solution for overfitting

Add a **regularization** term $R(\theta)$ to the loss function (for now written as maximizing logprob rather than minimizing loss)

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}} \sum_{i=1}^{m} \log P(y^{(i)} | x^{(i)}) - \alpha R(\theta)$$

Idea: choose an $R(\theta)$ that penalizes large weights

- fitting the data well with lots of big weights not as good as fitting the data a little less well, with small weights

L2 regularization (ridge regression)

The sum of the squares of the weights

$$R(\theta) = ||\theta||_2^2 = \sum_{j=1}^{n} \theta_j^2$$

L2 regularized objective function:

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}} \left[\sum_{i=1}^{m} \log P(y^{(i)} | x^{(i)}) \right] - \alpha \sum_{j=1}^{n} \theta_j^2$$

L1 regularization (=lasso regression)

The sum of the (absolute value of the) weights

$$R(\theta) \;=\; ||\theta||_1 = \sum_{i=1}^{n} |\theta_i|$$

L1 regularized objective function:

$$\hat{\theta} \;=\; \underset{\theta}{\operatorname{argmax}} \left[\sum_{1=i}^{m} \log P(y^{(i)}|x^{(i)}) \right] - \alpha \sum_{j=1}^{n} |\theta_j|$$

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

= (defunct names : Maximum entropy modeling or MaxEnt

So "logistic regression" will just mean binary (2 output 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 the **softmax**

- Takes a vector $z = [z_1, z_2, ..., z_k]$ of k arbitrary values
- Outputs a probability distribution
- each value in the range $[0,1]$
- all the values summing to 1

We'll discuss it more when we talk about neural networks

softmax: a generalization of sigmoid

- For a vector z of dimensionality k, the softmax is:

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

$$\text{softmax}(z_i) = \frac{\exp(z_i)}{\sum_{j=1}^{k}\exp(z_j)} \quad 1 \le i \le k$$

Example:

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

$$\text{softmax}(z) = [0.055, 0.090, 0.006, 0.099, 0.74, 0.010]$$

Components of a probabilistic machine learning classifier

Given m input/output pairs $(x^{(i)}, y^{(i)})$:

1. A **feature representation** for the input. For each input observation $x^{(i)}$, a vector of features $[x_1, x_2, \ldots, x_n]$. Feature j for input $x^{(i)}$ is x_j, more completely $x_1^{(i)}$, or sometimes $f_j(x)$.

2. A **classification function** that computes $\hat{y}$ the estimated class, via $p(y|x)$, like the **sigmoid** or **softmax** functions

3. An **objective function** for learning, like **cross-entropy loss**

4. An algorithm for **optimizing** the objective function: **stochastic gradient descent**

Next class:

- Language models