University of Colorado **Boulder**

Department of Computer Science

CSCI 5622: Machine Learning

Chris Ketelsen

Lecture 4: Stochastic Gradient Descent

Logistic Regression

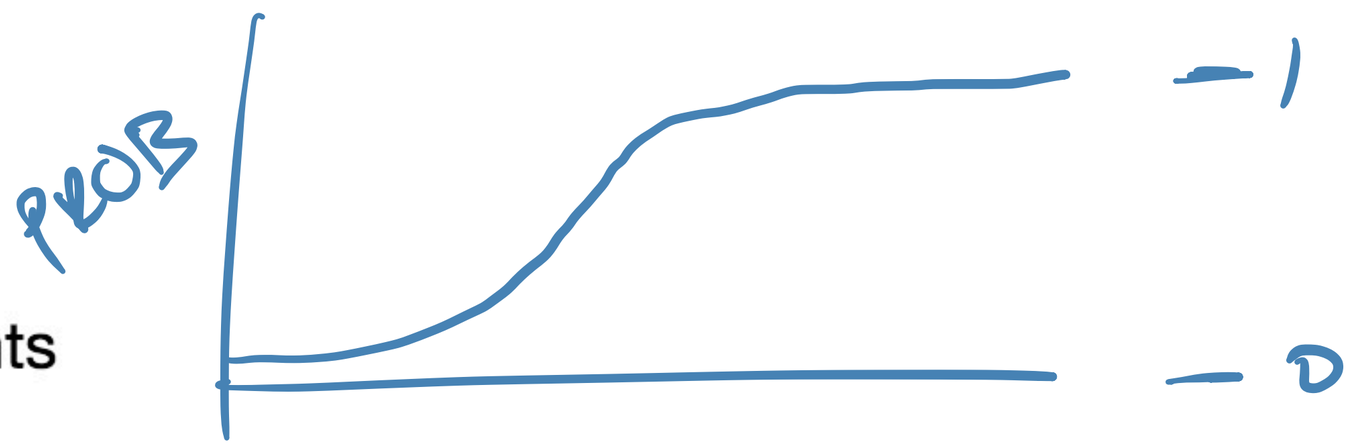
Last time we learned about the Logistic Regression classifier

$$p(y = 1 \mid \mathbf{x}; \mathbf{w}) = \text{sigm}(\mathbf{w}^T \mathbf{x}) = \frac{1}{1 + \exp(-\mathbf{w}^T \mathbf{x})}$$

If we've learned the correct weights $\hat{\mathbf{w}}$ then we can classify new $\mathbf{x}$

Classify $\mathbf{x}$ as $y = 1$ if
$\text{sigm}(\hat{\mathbf{w}}^T \mathbf{x}) > 0.5$
else classify $\mathbf{x}$ as $y = 0$

Today we learn to find the weights



Maximum Likelihood Estimation

For binary classification, we have

$$\begin{aligned} p(y = 1 \mid \mathbf{x}; \mathbf{w}) &= \text{sigm}(\mathbf{w}^T \mathbf{x}) \\ p(y = 0 \mid \mathbf{x}; \mathbf{w}) &= 1 - \text{sigm}(\mathbf{w}^T \mathbf{x}) \end{aligned}$$

which can be written more compactly as

$$p(y \mid \mathbf{x}; \mathbf{w}) = \text{sigm}(\mathbf{w}^T \mathbf{x})^{\,y} (1 - \text{sigm}(\mathbf{w}^T \mathbf{x}))^{\,1-y}$$

Notice that this looks like a Bernoulli random variable (i.e. a coin flip) with mean $\text{sigm}(\mathbf{w}^T \mathbf{x})$

Suppose now that we have training data $\{\mathbf{x}_i, y_i\}_{i=1}^m$

Maximum Likelihood Estimation

ASSUME INDEPENDENCE

For a fixed set of training data, we can write down the likelihood of the parameters $\mathbf{w}$

$$
\begin{aligned}
L(\mathbf{w}) &= p(y_1, y_2, \ldots, y_m \mid \mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_m; \mathbf{w}) \\
&= \prod_{i=1}^{m} p(y_i \mid \mathbf{x}_i; \mathbf{w}) \\
&= \prod_{i=1}^{m} \text{sigm}(\mathbf{w}^T \mathbf{x}_i)^{\,y_i} (1 - \text{sigm}(\mathbf{w}^T \mathbf{x}_i))^{\,1-y_i}
\end{aligned}
$$

TRAINING

TRAINING

← PRODUCT

For the given training data, $\{\mathbf{x}_i, y_i\}_{i=1}^{m}$, find parameters $\mathbf{w}$ that are most likely, given the data, by maximizing $L(\mathbf{w})$

Maximum Likelihood Estimation

$x_i \in \mathbb{R}^D$

To avoid the impending product rules, we'll take logs

Also, because optimization is usually cast as minimization, we'll make things negative to get the **negative log-likelihood**

$$NLL(\mathbf{w}) = -\log L(\mathbf{w})$$

$$NLL(\mathbf{w}) = -\sum_{i=1}^{m} \left[y_i \log \text{sigm}(\mathbf{w}^T \mathbf{x}_i) + (1 - y_i) \log(1 - \text{sigm}(\mathbf{w}^T \mathbf{x}_i)) \right]$$

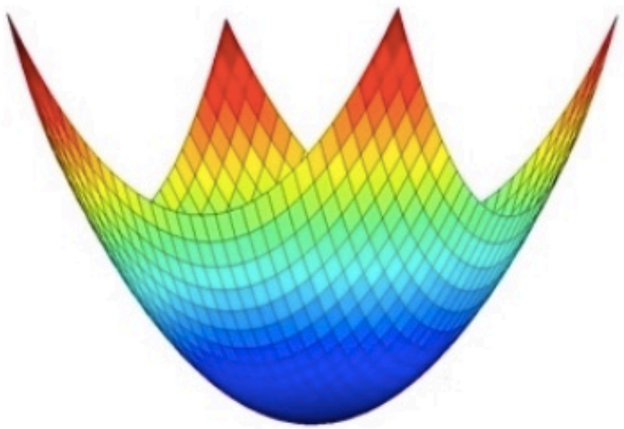
Remember that $\log$ is monotonically increasing, so a function is maximized and its negative log are minimized at the same point

That function looks nasty! This looks hard!

Maximum Likelihood Estimation

But it's really not that bad

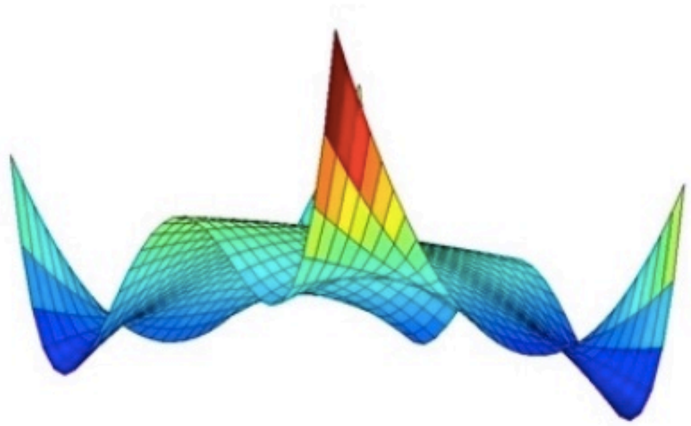
Both $L(\mathbf{w})$ and $NLL(\mathbf{w})$ are **convex** functions



Maximum Likelihood Estimation

But it's really not that bad

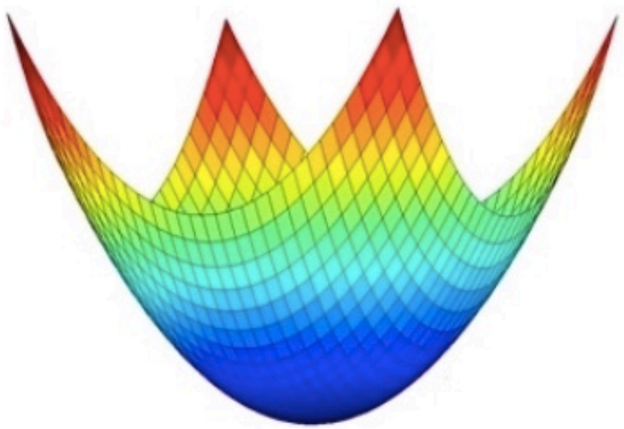
Which are much better than **non-convex** functions



Maximum Likelihood Estimation

But it's really not that bad

Both $L(\mathbf{w})$ and $NLL(\mathbf{w})$ are **convex** functions



We'll use an iterative method called **Gradient Descent**

Basic Idea: Pick a starting point, and gradually walk downhill until we get to the bottom

Gradient Descent

But how do we know which way is "down"?

Follow the **(Negative) GRADIENT**!

The gradient of a function f is a **vector** that points in the direction that f **increases** the fastest (remember, we want to **decrease** fast)

The general form we'll use to update our guess is

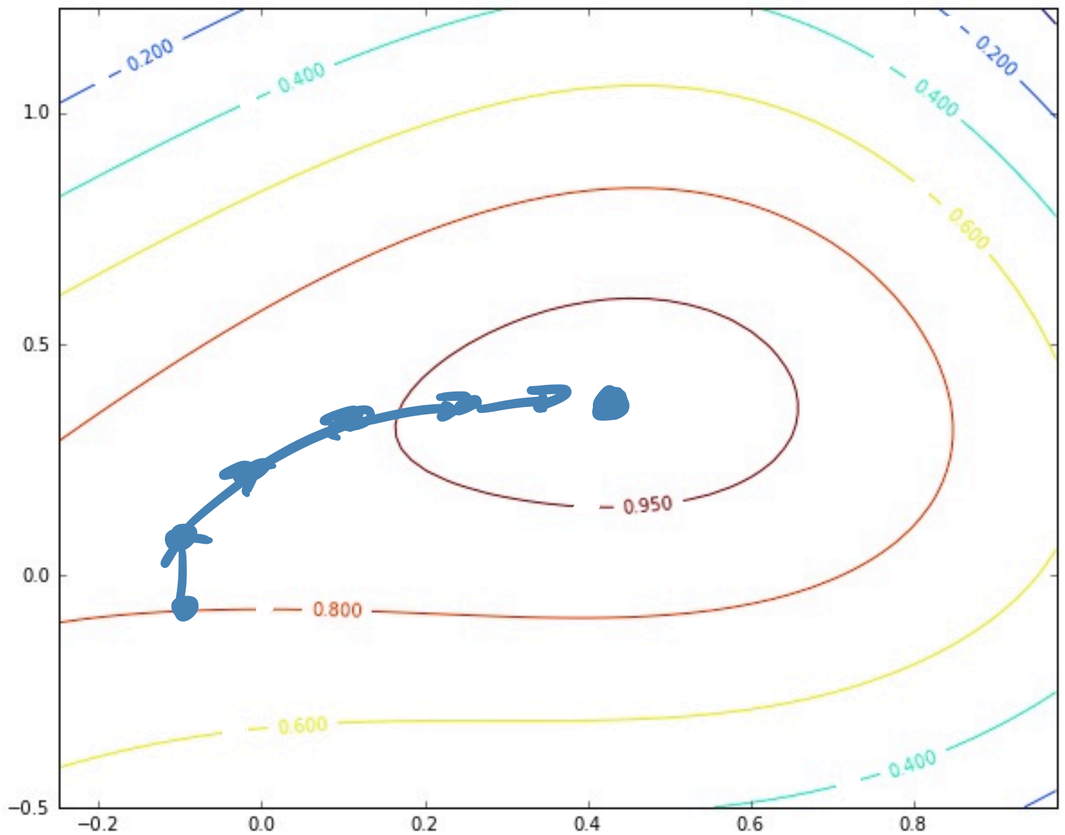
$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} NLL(\mathbf{w})$$

$$\nabla_{\mathbf{w}} NLL(\mathbf{w}) = \left[\frac{\partial NLL(\mathbf{w})}{\partial w_0}, \frac{\partial NLL(\mathbf{w})}{\partial w_1}, \cdots, \frac{\partial NLL(\mathbf{w})}{\partial w_D} \right]^T$$

η is called the **learning rate**. It determines how big of a step we take in the downhill direction

Gradient Descent

A cartoonish example



Gradient Descent

Let's write the vector update in terms of individual components

$$w_k \leftarrow w_k - \eta \frac{\partial NLL(\mathbf{w})}{\partial w_k}, \quad \text{for } k = 0, \ldots, D$$

It turns out, can show that

$$\frac{\partial NLL(\mathbf{w})}{\partial w_k} = \sum_{i=1}^{m} [\ \text{sigm}(\mathbf{w}^T \mathbf{x}_i) \ - y_i \ ] x_{ik}$$

Here x_{ik} is the k^{th} entry in the i^{th} training example

$$X = \begin{bmatrix} x_1^T \\ x_2^T \\ \vdots \\ x_m^T \end{bmatrix} \Rightarrow \quad x_i^T = (1, x_{i1}, x_{i2} \ldots x_{ik})$$

Gradient Descent

Assuming we've stored the $\mathbf{x}_i$'s in a matrix, psuedocode for Gradient Descent might look like

In [ ]:
```
while not converged:
    for kk in [0,...,D]:
        g[kk] = 0
        for ii in [1,...,m]:
            g[kk] = g[kk] + (sigm(dot(w, x[ii,:]) - y[ii]) * x[ii,kk]
    w = w - eta * g
```

Many questions:

- Where did that formula for the gradient come from?
- When's this going to become **STOCHASTIC**?
- How do we choose the learning rate?
- Where do I start and when do I stop?

Math Details

Recall that the negative log-likelihood function that we're trying to minimize is given by

$$NLL(\mathbf{w}) = -\sum_{i=1}^{m} \left[y_i \log \text{sigm}(\mathbf{w}^T \mathbf{x}_i) + (1 - y_i) \log(1 - \text{sigm}(\mathbf{w}^T \mathbf{x}_i)) \right]$$

A few slides back I claimed that we could write the k^{th} entry in $\nabla_{\mathbf{w}} NLL(\mathbf{w})$ in the very simple form

$$\frac{\partial NLL(\mathbf{w})}{\partial w_k} = \sum_{i=1}^{m} [\ \text{sigm}(\mathbf{w}^T \mathbf{x}_i) - y_i\ ] x_{ik}$$

Math Details

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

Let's use the simpler notation $s(z) = \text{sigm}(z)$

The sigmoid function has a nice derivative property

Fact: $\frac{d}{dz} s(z) = s(z)[1 - s(z)]$

Proof:

$$\frac{d}{dz}\left(\frac{1}{1+e^{-z}}\right) = \frac{d}{dz}(1+e^{-z})^{-1}$$

$$= -1(1+e^{-z})^{-2}(-e^{-z}) = \frac{e^{-z}}{(1+e^{-z})^2}$$

$$\frac{1}{1+e^{-z}}\left(\frac{e^{-z}}{1+e^{-z}}\right) = \frac{1}{1+e^{-z}}\left(\frac{(1+e^{-z})-1}{1+e^{-z}}\right) = \frac{1}{1+e^{-z}}\left(1 - \frac{1}{1+e^{-z}}\right)$$

$s(z)$

Math Details

Taking the derivative of $NLL(\mathbf{w})$ wrt to w_k gives

$$\frac{\partial NLL(\mathbf{w})}{dw_k} = -\sum_{i=1}^{m}\left[y_i \frac{\partial}{\partial w_k}\log s(\mathbf{w}^T\mathbf{x}_i) + (1-y_i)\frac{\partial}{\partial w_k}\log(1-s(\mathbf{w}^T\mathbf{x}_i))\right]$$

Look at the two derivative terms separately

$$\frac{\partial}{\partial w_k}\log s(\mathbf{w}^T\mathbf{x}_i) = \frac{1}{s(w^Tx_i)}\frac{\partial}{\partial z}s(w^Tx_i)\, x_{ik}$$

$$= \frac{1}{s(w^Tx_i)}\, s(w^Tx_i)(1-s(w^Tx_i))\frac{\partial}{\partial w_k}(w^Tx_i)$$

$$= (1-s(w^Tx_i))\,x_{ik}$$

Math Details

Taking the derivative of $NLL(\mathbf{w})$ wrt to w_k gives

$$\frac{\partial NLL(\mathbf{w})}{dw_k} = -\sum_{i=1}^{m}\left[y_i[1 - s(\mathbf{w}^T\mathbf{x}_i)]x_{ik} + (1 - y_i)\frac{\partial}{\partial w_k}\log(1 - s(\mathbf{w}^T\mathbf{x}_i))\right]$$

Look at the two derivative terms separately

$$\frac{\partial}{\partial w_k}\log\left(1 - s(\mathbf{w}^T\mathbf{x}_i)\right) = \frac{1}{1 - S(w^T x_i)}\left(-S(w^T x_i)(1 - S(w^T x_i))\right) x_{ik}$$

$$= -S(w^T x_i)\, x_{ik}$$

Math Details

We then have

$$\frac{\partial NLL(\mathbf{w})}{dw_k} = -\sum_{i=1}^{m} \left[y_i [1 - s(\mathbf{w}^T \mathbf{x}_i)] x_{ik} - (1 - y_i) s(\mathbf{w}^T \mathbf{x}_i) x_{ik} \right]$$

$$= -\sum_{i=1}^{m} \{ y_i - s(w^T x_i) \} x_{ik}$$

$$= \sum_{i=1}^{m} \{ s(w^T x_i) - y_i \} x_{ik}$$

Math Details

We then have

$$\begin{aligned}
\frac{\partial NLL(\mathbf{w})}{dw_k} &= -\sum_{i=1}^{m} \left[y_i [1 - s(\mathbf{w}^T \mathbf{x}_i)] x_{ik} - (1 - y_i) s(\mathbf{w}^T \mathbf{x}_i) x_{ik} \right] \\
&= \sum_{i=1}^{m} [\text{sigm}(\mathbf{w}^T \mathbf{x}_i) - y_i] x_{ik}
\end{aligned}$$

Gradient Descent

Let's write the vector update in terms of individual components

$$w_k \leftarrow w_k - \eta \frac{\partial NLL(\mathbf{w})}{\partial w_k}, \quad \text{for } k = 0, \ldots, D$$

And now we know that

$$\frac{\partial NLL(\mathbf{w})}{\partial w_k} = \sum_{i=1}^{m} [\ \text{sigm}(\mathbf{w}^T \mathbf{x}_i) - y_i\ ] x_{ik}$$

So in component form, we have for each $k = 0, \ldots, D$:

$$w_k \leftarrow w_k - \eta \sum_{i=1}^{m} [\ \text{sigm}(\mathbf{w}^T \mathbf{x}_i) - y_i\ ] x_{ik}$$

Stochastic Gradient Descent

SGD is motivated by the fact that our data is **BIG**

Almost certainly too big to fit into memory

Looping over each training example everytime we update an entry in $\mathbf{w}$ would be **SLOW**

What if we updated weights one training example at a time?

Stochastic Motivation: Updating with the **true** gradient

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \, \mathbb{E}[\nabla_{\mathbf{w}} NLL(\mathbf{w})]$$

Don't have access to the expectation, so instead we approximate it with a single training example

Stochastic Gradient Descent

Stochastic Motivation: Updating with the **true** gradient

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \, \mathbb{E}[\nabla_{\mathbf{w}} NLL(\mathbf{w})]$$

Don't have access, so instead we approximate it with a single training example

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \, \nabla_{\mathbf{w}} NLL(\mathbf{w}; \mathbf{x}_i, y_i)$$

In component form

$$w_k \leftarrow w_k - \eta[\ \mathrm{sigm}(\mathbf{w}^T \mathbf{x}_i) - y_i\ ]x_{ik}, \quad \text{for } k = 0, \ldots, D$$

After updating all weights based on one training example, select a not-yet-used example at random and repeat until all examples used

Stochastic Gradient Descent

Psuedocode for the process might look like this

In [ ]:
```
while not converged:
    shuffle(x, y)
    for ii in [1,...,m]:
        muii = sigm(dot(w, x[ii,:]))-y[ii]
        for kk in [0,...,D]:
            w[kk] = w[kk] - eta * muii * x[ii,kk]
```

Machine Learning Vernacular: One complete trip through the training data is called an **Epoch**

OK, so this doesn't seem so bad

But we can do better...

Mini-Batch Stochastic Gradient Descent

Randomly grab a fixed number of training samples and use them to update $\mathbf{w}$

$$w_k \leftarrow w_k - \eta \sum_{i=1}^{m_b} [\ \mathrm{sigm}(\mathbf{w}^T \mathbf{x}_i) - y_i\ ] x_{ik}$$

Typical batch sizes are around 256 training examples

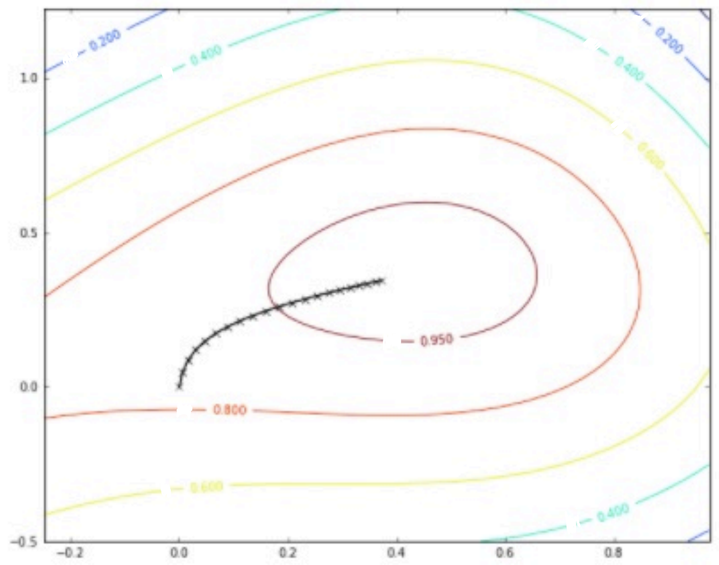
Likely can fit the entire batch into memory

Vectorized linear algebra routines make it fast

Choosing the Learning Rate

This is the hard part

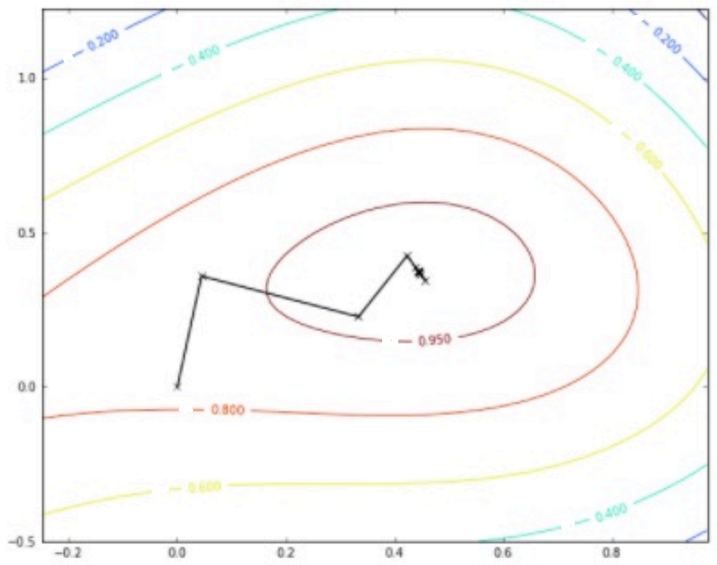
Too small a learning rate and it takes forever to get to maximum



Choosing the Learning Rate

This is the hard part

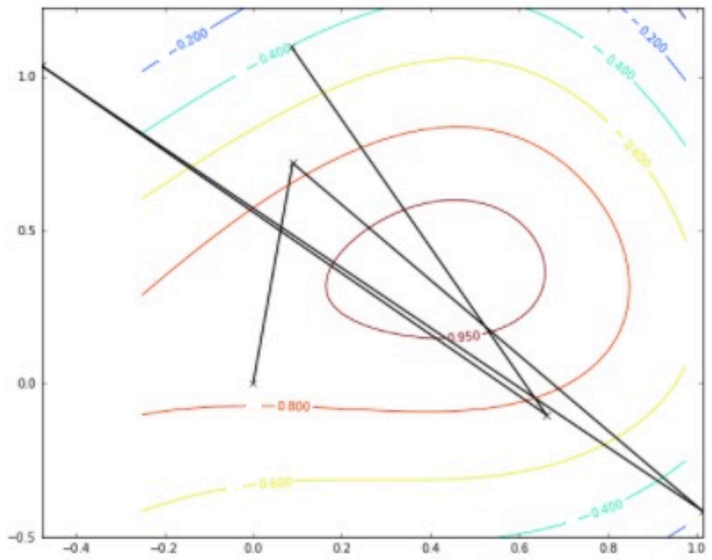
Slightly bigger can give weird oscillations, but maybe still converge



Choosing the Learning Rate

This is the hard part

Too big and you just bounce around (or diverge)

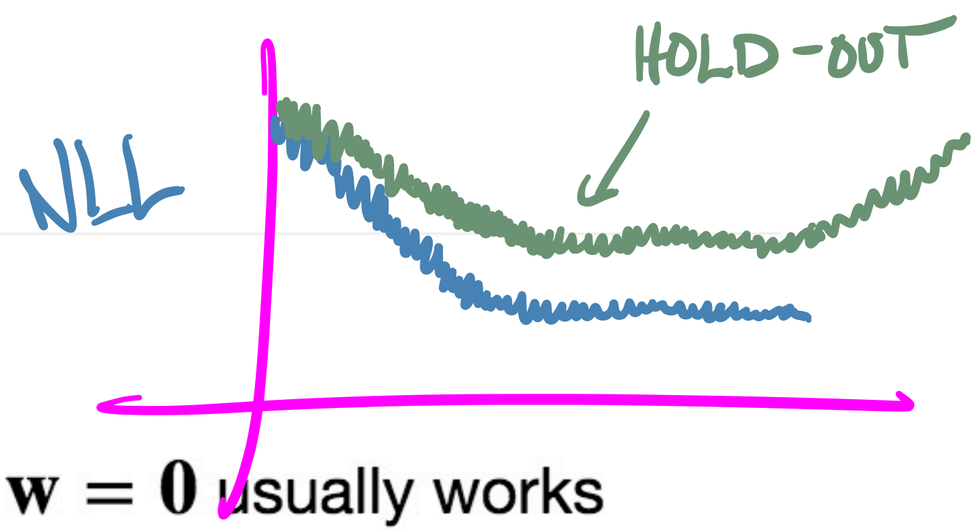


Choosing the Learning Rate

So what should we do?

- Good constant learning rates can be chosen by testing a range of reasonable values and monitoring the decrease in the negative log-likelihood function
- With a good initial learning rate, most implementation of SGD will implement some kind of a schedule for the learning rate, usually decreasing slowly as you increase the number of epochs
- Some advanced techniques even allow for different learning rates for different features

Starting and Stopping



What should my initial guess for $\mathbf{w}$ be?

- If there is no colinearity in your data, $\mathbf{w} = \mathbf{0}$ usually works fine. If you're worried about colinearity then start with small random values. (**Safe Bet**: Initialize randomly just in case)

When should I stop?

- When the NLL on a **hold-out set** stops decreasing
- When you've done "ten" passes through the data

One More Important Change! - Regularization

It's common in many learning algorithms to add a penalty term to the log-likelihood function that penalizes large weights.

$$\text{NLL}(\mathbf{w}) = -\sum_{i=1}^{m} \left[y_i \log \text{sigm}(\mathbf{w}^T \mathbf{x}_i) + (1 - y_i) \log(1 - \text{sigm}(\mathbf{w}^T \mathbf{x}_i)) \right] + \lambda \sum_{k=1}^{D} w_k^2$$

Loss =

Weights that grow too large in magnitude during training can be a sign that you're overfitting your training set.

Penalty term balances minimizing NLL with shrinking weights

Don't penalize the bias weight w_0

One More Important Change! - Regularization

How does this affect our SGD algorithm?

Minor changes to the update rules

$$w_0 \leftarrow w_0 - \eta[\ \text{sigm}(\mathbf{w}^T\mathbf{x}_i) - y_i\ ]x_{i0}$$

$$w_k \leftarrow w_k - \eta\{[\ \text{sigm}(\mathbf{w}^T\mathbf{x}_i) - y_i\ ]x_{ik} + 2\lambda w_k\}, \quad \text{for } k \geq 1$$

Talk loads more about regularization when we get to theory

In Class

- **Get out your laptops and get into groups!**
- Let's practice on a simple text problem
- Talk about a cool trick to make SGD loads more efficient for text learning

In Class

$y=1$ POS: AAAA BBBC $\Rightarrow$

$y=0$ NEG: BBCCCD $\Rightarrow$

$$X_1 = \begin{bmatrix} 1 \\ 4 \\ 3 \\ 1 \\ 0 \end{bmatrix} \begin{matrix} \text{BIAS} \\ A \\ B \\ C \\ D \end{matrix}$$

$$X_2 = \begin{bmatrix} 1 \\ 0 \\ 2 \\ 3 \\ 1 \end{bmatrix}$$

$$W = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$\eta = 1$

$y_1 = 1 \quad \mu_i = [\text{sigm}(w^T x_i) - y_i] = [\frac{1}{2} - 1] = -\frac{1}{2}$

$$\text{sigm}(w^T x_i) = \frac{1}{1+e^{-0}} = \frac{1}{2}$$

$$\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} - 1 \cdot -\frac{1}{2} \begin{bmatrix} 1 \\ 4 \\ 3 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} .5 \\ 2 \\ 1.5 \\ .5 \\ 0 \end{bmatrix} \qquad w^{(1)} = \begin{bmatrix} .5 \\ 2 \\ 1.5 \\ .5 \\ 0 \end{bmatrix}$$

In Class

$$w^{(1)} = \begin{bmatrix} .5 \\ 2 \\ 1.5 \\ .5 \\ 0 \end{bmatrix} \quad \longleftrightarrow \quad x_2 = \begin{bmatrix} 1 \\ 0 \\ 2 \\ 3 \\ 1 \end{bmatrix} \quad y_2 = 0$$

$$h_2 = [\text{sigm}(w^T x_2) - y_2] \qquad w^T x = .5 + 0 + 3 + 1.5 + 0 - .5 = 4$$

$$[\text{sigm}(5) - 0] = 0.993$$

$$\begin{bmatrix} .5 \\ 2 \\ 1.5 \\ .5 \\ 0 \end{bmatrix} - 1.0993 \begin{bmatrix} 1 \\ 0 \\ 2 \\ 3 \\ 1 \end{bmatrix} = \begin{bmatrix} -.493 \\ 2.0 \\ -4.87 \\ -2.430 \\ -.993 \end{bmatrix} = w^{(1)}$$

In Class

In Class

In Class

In Class

In Class