Optimization

Rob McCulloch

1. Optimization
2. Derivatives
3. Logit Log Likelihood Derivatives
4. Taylor's Theorem and Local Minimums
5. Convexity
6. Gradient Descent
7. Stochastic Gradient Descent
8. Momentum
9. Newton's Method

1. Optimization

Optimization plays an important role in statistics/Machine Learning/data science.

The basic we example we have seen is maximum likelihood estimation.

Let

$$\ell(\theta) = -\log(f(y \mid \theta)) \equiv -\log(L(\theta))$$

be the negative log likelihood.

We can think of $-\log(L(\theta))$ as our "lack of fit" or *loss* on the training data used to estimate θ.

Then the maximum likelihood estimate of θ is

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}}\ \ell(\theta)$$

In machine learning we often consider regularized versions:

$$\hat{\theta} = \underset{\theta}{\text{argmin}}\ J(\theta) = \ell(\theta) + P(\theta, \lambda)$$

where

- $\ell(\theta)$ is a measure of lack of fit on training data.
- $P(\theta, \lambda)$ is a complexity penalty with tuning parameter λ.
- $P(\theta, \lambda)$ is often expressed as $\lambda\,\Omega(\theta)$, where Ω meaures complexity and λ controls the complexity penalty.

For example, in the popular Ridge regression approach

$$\theta = \beta, \ \ \text{Loss}(\theta) = ||y - X\beta||^2, \ \ P(\theta, \lambda) = \lambda \sum \beta_j^2.$$

giving,

$$\underset{\beta}{\text{minimize}}\ ||y - X\beta||^2 + \lambda \sum_{j=1}^{p} \beta_j^2.$$

In the linear regression problem with squared error loss

$$\underset{\beta}{\text{minimize}}\ ||y - X\beta||^2$$

we can best understand the solution as on orthogonal projection of y onto the subspace spanned by the columns of X.

We obtain the solution

$$\hat{\beta} = (X'X)^{-1}X'y.$$

Computational and statistical insights into this solution are gained by considering basic matrix decompostions of X or $X'X$.

For non-linear models, we need to develop alternative solutions.

Typically, these is not a closed form solution.

Most approaches involve an iterative scheme where we seek to get closer to the minimum at each iteration.

Often iterative schemes are along the lines of

- Let θ_t be the value of θ at iteration t.
- At θ_t, approximate $J(\theta)$ is some way.
- based on the approximation update θ from θ_t to θ_{t+1}.
- keep iterating $\theta_t \Rightarrow \theta_{t+1}$, and monitor $J(\theta_t)$ and/or θ_t,
  stop when $J(\theta_t)$ or θ_t does not change very much.

For example the EM algorithm works this way.

A basic approach to approximating $J(\theta)$ is Taylor's Theorem.

First order methods use the first derivative of J and second order methods use the first and second derivatives.

We will look at the key first order methods, gradient descent and stochastic gradient descent.

Stochastic gradient descent is the workhorse approach to optimization in the Machine Learning literature. For example, neural nets are typically optimized using a version of stochastic gradient descent.

Then we will look at the basic second order method, Newton's method.

2. Derivatives

We will need the first derivative:

$$f'(x) = \nabla f(x) = [\frac{\partial f(x)}{\partial x_1}, \frac{\partial f(x)}{\partial x_2}, \ldots, \frac{\partial f(x)}{\partial x_p}]$$

and the second derivative:

$$f''(x) = [\frac{\partial^2 f(x)}{\partial x_i \partial x_j}] = H(x)$$

- the first derivative is called the gradient vector. My convention is that it is a $1 \times p$ row vector.
- the second derivative is a $p \times p$ symmetric matrix. It is called the Hessian.

3. Logit Derivatives

We will use maximizing the logit likelihood as an example.

The logit is a good example because:

- logistic regression is a key statistical model that must be optimized numerically.
- it is convex.

It is a bad example because
convex ⇒ too easy.

Let's compute the gradient and Hessian for logistic regression.

The Likelihood

$Y_i \sim \text{Bernoulli}(F(x_i'\,\beta))$.

$$L(\beta) = \prod_{i=1}^{n} F(x_i'\beta)^{y_i}\,(1 - F(x_i'\beta))^{(1-y_i)}$$

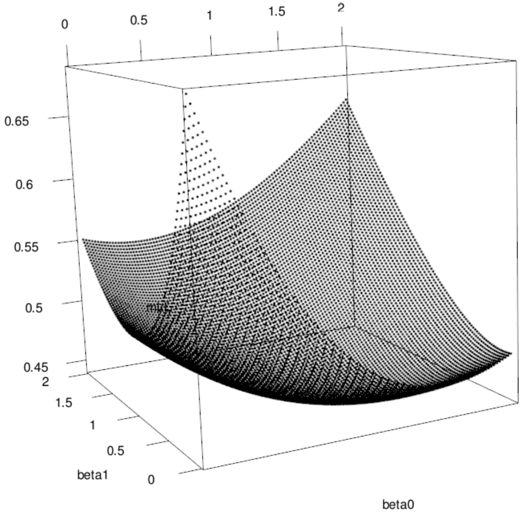
$$-\log L(\beta) = -\sum [y_i\,\log(F_i) + (1-y_i)\log(1-F_i)]$$

where $F_i = F(x_i'\beta)$.

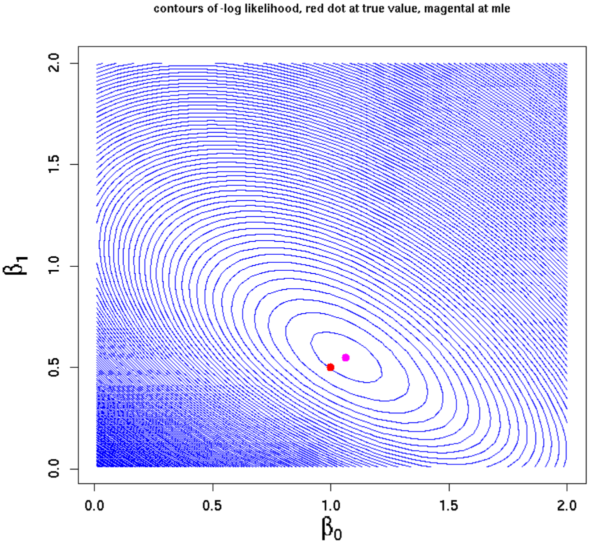
So here, "$\theta = \beta$".

Also, we could think of $F(x_i'\beta)$ as p_i with $Y_i \sim p_i$.

3D plot of $-\log L(\beta_0, \beta_1)$ simulated data, one x, slope and intercept.



Contours of $-\log L$.



We will compute the first and second derivatives of the logit log likelihood.

First, we differentiate $F(\eta) = \frac{\exp(\eta)}{1+\exp(\eta)}$:

$$F'(\eta) = \frac{(1+e^{\eta})e^{\eta} - e^{\eta}e^{\eta}}{(1+e^{\eta})^2} = F(\eta)(1-F(\eta))$$

$F_i = F(x_i'\beta)$.

$$-\log L(\beta) = -\sum [y_i \log(F_i) + (1 - y_i)\log(1 - F_i)]$$

Let's drop the i and just differentiate $y \log(F(x'\beta))$.

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_p \end{bmatrix}$$

$$\frac{\partial\left[y \log\left(F(x^T\beta)\right)\right]}{\partial \beta_j} = y \frac{1}{F(x^T\beta)} F(x^T\beta)(1-F(x^T\beta))\, x_j$$

$$\nabla\left[y F(x^T\beta)\right] = y x^T \left[1 - F(x^T\beta)\right]$$

$$\begin{aligned}
-\log L'(\beta) &= -\sum [y_i x_i' \frac{F_i(1-F_i)}{F_i} + x_i'(1-y_i)[-\frac{F_i(1-F_i)}{(1-F_i)}]] \\
&= -\sum [y_i x_i'(1-F_i) - x_i'(1-y_i)F_i] \\
&= -\sum x_i'(y_i - F_i) \\
&= -(y - F_v)'X
\end{aligned}$$

where the i^{th} of X is x_i' and F_v is the vector $[F_i]$.

To compute the second derivative, let's drop the *i* again.

$$\frac{\partial[-\log L]}{\partial \beta_j} = -x_j(y - F(x^T\beta))$$

$$\frac{\partial[-\log L]}{\partial \beta_k \, \partial \beta_j} = x_j \{F(x^T\beta)(1 - F(x^T\beta))\} x_k$$

$$H(\beta) = F(x^T\beta)(1 - F(x^T\beta)) x x^T$$

Put i back in, and sum over $i = 1, 2, \ldots, n$.

$$\begin{aligned} -\log L''(\beta) &= \sum x_i x_i' F_i (1 - F_i) \\ &= X' D X \end{aligned}$$

where,

$$D = diag(F_i(1 - F_i))$$

4. Taylor's Theorem and Local Minimums

- The first derivative give as a local linear approximation to a function.
- The first and second derivates give us a local quadratic approximation to a function.

We approximate $f(x)$ in a neighborhood of x_0,

$x \in R^p$, $f : R^p \to R$.

First order:

$$f(x) \approx f(x_0) + f'(x_0)(x - x_0)$$

Second order:

$$f(x) \approx f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2}(x - x_0)' f''(x_0)(x - x_0)$$

Or (different notation):

First order:

$$f(x) \approx f(x_0) + \nabla f(x_0)(x - x_0)$$

Second order:

$$f(x) \approx f(x_0) + \nabla f(x_0)(x - x_0) + \frac{1}{2}(x - x_0)' H(x_0)(x - x_0)$$

First and Second Order Approximation 1D

Simulated data with one x and no intercept so
$\theta = \beta,\ p(x) = F(x\beta)$.

$J(\beta) = -\log(L(\beta))$, the negative log-likelihood.

$$J'(\beta) = -\sum_{i=1}^{n} (y_i - F(x_i\beta))x_i$$

$$J''(\beta) = \sum_{i=1}^{n} (1 - F(x_i\beta))F(x_i\beta)x_i^2$$

First order:

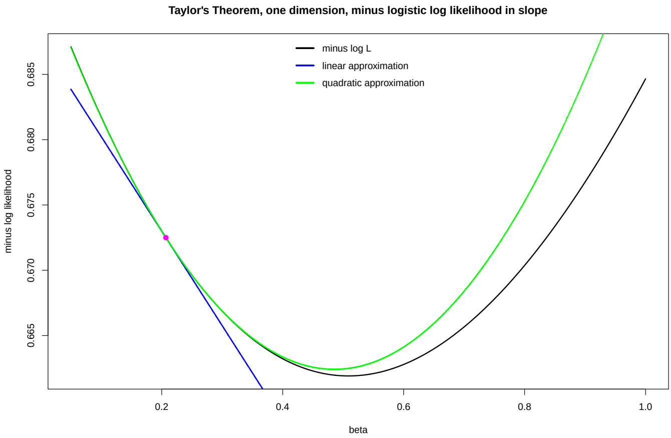
$$J(\beta) \approx J(\beta_o) + J'(\beta_o)(\beta - \beta_o)$$

Second order:

$$J(\beta) \approx J(\beta_o) + J'(\beta_o)(\beta - \beta_o) + \frac{1}{2}J''(\beta_o)(\beta - \beta_o)^2$$

Note: $J''(\beta) > 0,\ \forall\beta$.

Here are the first order (linear) and second order (quadratic) approximation to -loglikelihood for the logit.



Local Minimums, 1D

x_o is a *local minimum of* $f(x)$ if small changes in x_o make f bigger.

If x_o is a local min, we must have $f'(x_o) = 0$, otherwise we would know how to change it to make f smaller.

If $f''(x_o) > 0$, x_o is a local min.

First Order Approximation More than 1D

In higher dimensions (more than 1), things get much more interesting in the we have to think about directions rather than just up or down.

The gradient is a multivariate derivative in that (skipping some technical details):

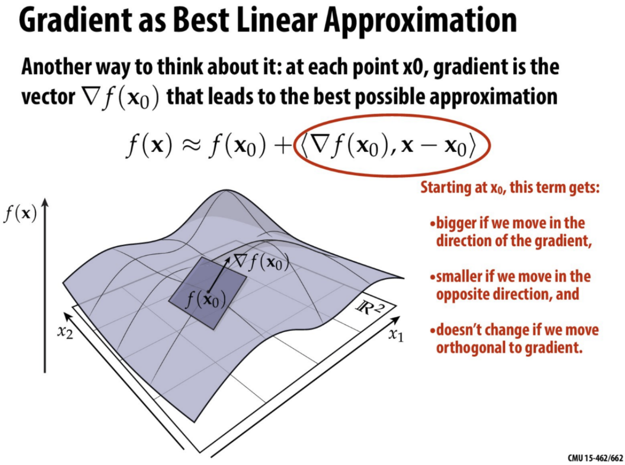
$$f(x) \approx f(a) + \nabla f(a)(x - a)$$

Note that $\nabla f(x)$ is a row vector so that the product above makes sense with x a column vector.

An alternative notation is:

$$f(x) \approx f(a) + < \nabla f(a), (x - a) >$$

Stolen off the web:



Gradient as Best Linear Approximation

Another way to think about it: at each point x0, gradient is the vector $\nabla f(\mathbf{x}_0)$ that leads to the best possible approximation

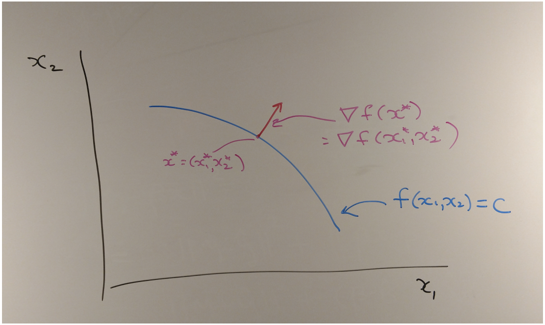
$$f(\mathbf{x}) \approx f(\mathbf{x}_0) + \langle \nabla f(\mathbf{x}_0), \mathbf{x} - \mathbf{x}_0 \rangle$$

Starting at x_0, this term gets:

- bigger if we move in the direction of the gradient,
- smaller if we move in the opposite direction, and
- doesn't change if we move orthogonal to gradient.

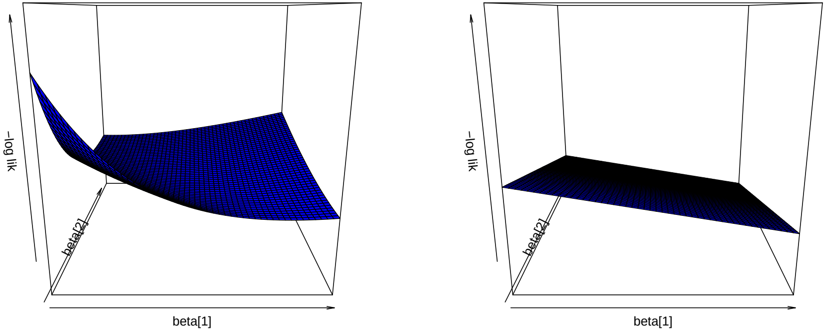
CMU 15-462/662

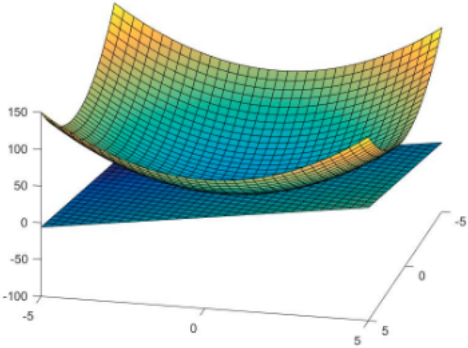
We can visualize the gradient using the *contours* of f.
A *contour* is the set $\{x : f(x) = c\}$.



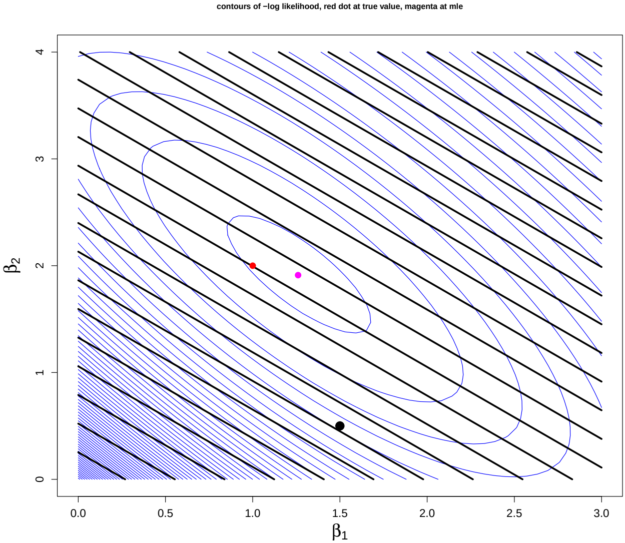
- If you want to increase f as fast as possible, go in the direction of the gradient ∇f.
- If you want to decrease f as fast as possible, go in the direction of the negative gradient $-\nabla f$.
- If you want to move without changing f go in a direction orthogonal to the gradient.

Left: - log lik, logit. Right: linear approx.




150
100
50
0
-50
-100
-5
0
5
-5
0
5

Contours of logit -logL and contours of linear approximation.



First Order Condition of a local Min/Max

Neccessary Condition for a local min/max:

If x^* is a local min (or max) then we must have

$$\nabla f(x^*) = 0$$

Otherwise, you would know which way to move to make f bigger or smaller.

Second Order Approximation, More than 1D

$$f(x) \approx f(x_o) + \nabla f(x_o)(x - x_o) + \frac{1}{2}(x - x_o)' H(x_o)(x - x_o)$$

The Hessian is a symmetric matrix.

We can gain insight into the nature of this approximation using the eigen vector/ eigen value decompostion of H.

We do a reparametrization in terms of the orthonormal eigen vectors.

$$f(x) \approx f(x_0) + \nabla f(x_0)(x - x_0) + \frac{1}{2}(x - x_0)^T H(x_0)(x - x_0)$$

$$H(x_0) = P D P^T$$

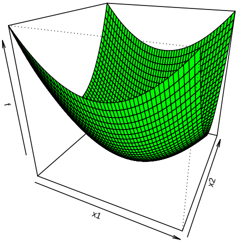
$$z = P^T(x - x_0) \qquad (x - x_0) = P z$$

$$g(z) = g(0) + \nabla f(x_0) P z + \frac{1}{2} z^T D z$$

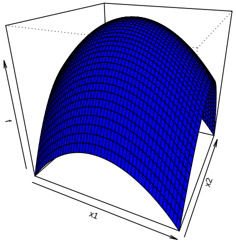
$$= a + \sum b_i z_i + \frac{1}{2} \sum z_i^2 d_i$$

$$= a + \sum \left[b_i z_i + \frac{1}{2} d_i z_i^2 \right]$$

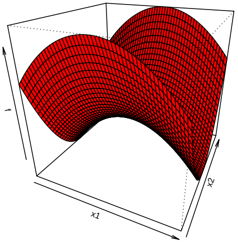
positive eigen values



negative eigen values



positive and negative eigen values



What are the eigen values for our logit problem?

$$\begin{aligned} H = -\log L''(\beta) &= \sum x_i x_i' F_i(1 - F_i) \\ &= X'DX \end{aligned}$$

where,

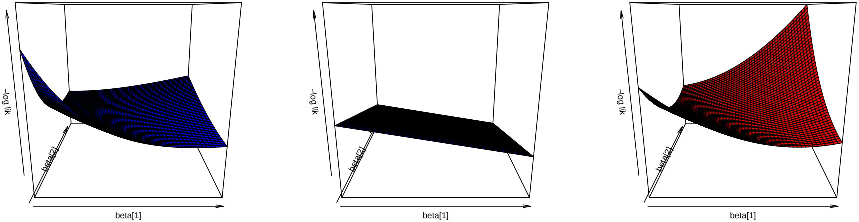
$$D = diag(F_i(1 - F_i))$$

Let $w = Xv$, for any v.

$v'Hv = (Xv)'D(Xv) = \sum D_i w_i^2 > 0$ as long as X has full rank.

Simulated logit data with 2 highly correlated x variables.

True, linear approx, quadratric approx.



Expand around beta1o = 1.5 beta2o = .5.

Gradient:

```
          [,1]
x1 -0.03511730
x2 -0.03727138
```

Hessian:

```
           x1         x2
x1 0.07029308 0.04867294
x2 0.04867294 0.04590770
```

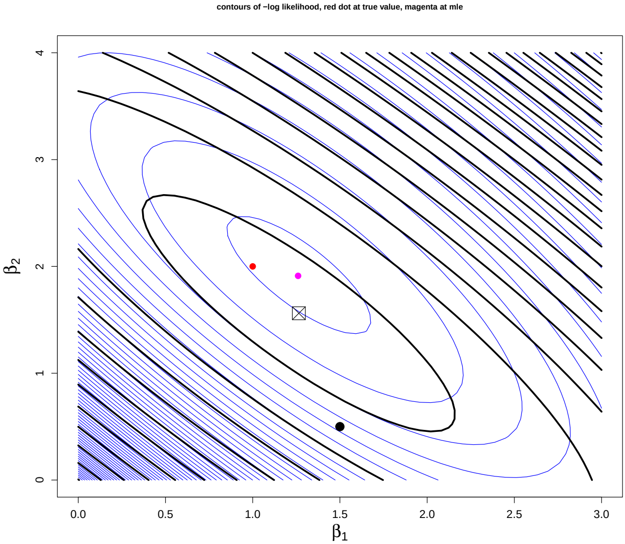
Eigen:

```
eigen() decomposition
$values
[1] 0.108277245 0.007923534

$vectors
           [,1]       [,2]
[1,] -0.7883509  0.6152259
[2,] -0.6152259 -0.7883509
```

-logL contours and contours of the quadratic approximation.

Expand around black dot.



Second Order Conditions for a local Min/Max

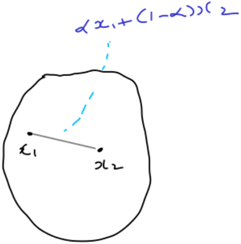
If $\nabla f(x_o) = 0$, x_o is a *critical point*

If $H(x_o)$ is positive definite, than it is a local min.

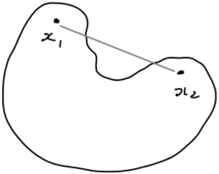
5. Convexity

A set S is convex if:

$$x_1, x_2 \in S \Rightarrow \alpha\, x_1 + (1-\alpha)\, x_2 \in S, \forall \alpha \in [0,1].$$

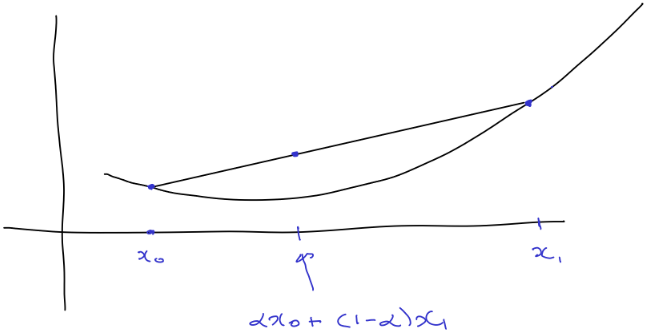


S convex



S not convex

Recall: a function is convex if, for $\alpha \in [0, 1]$:



$$f(\alpha x_0 + (1-\alpha)x_1) \leq \alpha f(x_0) + (1-\alpha) f(x_1)$$

The function is *strictly convex* if

$$f(\alpha x_0 + (1-\alpha)x_1) < \alpha f(x_0) + (1-\alpha)f(x_1)$$

Convexity is key in optimization because if f is strickly convex then,

a local min is a global min !!!

If we are optimizing a convex function over a convex set, life tends to be easier !!!!

Sufficient condition for Convexity:

If the Hessian is postitive definite everywhere , then the function is strictly convex.

Our key example is the logit $-\log L$.

$$
\begin{aligned}
H = -\log L''(\beta) &= \sum x_i x_i' F_i (1 - F_i) \\
&= X' D X
\end{aligned}
$$

where,

$$D = diag(F_i(1 - F_i))$$

Let $w = Xv$, for any v.

$v' H v = (Xv)' D (Xv) = \sum D_i w_i^2 > 0$ as long as X has full rank.

6. Gradient Descent

Typically it is difficult to solve

$$\nabla J(\theta) = 0$$

To find a local minimum.

In large problems (e.g. neural nets) it may be out of the question to compute the Hessian.

A neural net could have a million parameters and a million squared is big.

Hence simple iterative approaches that just depend on computing the gradient are very useful.

Iterative methods are a basic tool in optimization.
We want to minimize $J(\theta)$.

Start at θ_0.

```
For $t$ in 1:nit:
    Figure out something about $J$ at $\theta_t$.
    Based on what you figured out, $\theta_{t+1} \leftarrow \theta_t$ somehow.
```

Choose nit big enough that "you have converged", that is θ_t or $J(\theta_t)$ does not seem to be changing much.

Maybe a "while not converged loop" instead of a for loop.

If you can compute the gradient, the obvious thing to do is head of the direction $-\nabla J(\theta_t)$.

The objective function $J(\theta)$ goes down fastest in direction $-\nabla J(\theta)$, so why don't we head off in that direction.

$$\theta_{t+1} = \theta_t - \alpha_t \nabla J(\theta_t), \;\; \alpha_t > 0.$$

We have to pick how far to go which is the role of α_t which is called the *learning rate*.

There are various schemes for making the learning rate depend on the iteration and this is a major part of the game. "optimization" in keras means choosing the scheme for adjusting the learning rate.

```
#compile model
nn1.compile(loss=’mse’,optimizer=’rmsprop’,metrics=[’mse’])
```

Alternatively, this kind of algorithm is often written as:

$$\theta \leftarrow \theta - \alpha \nabla J(\theta)$$

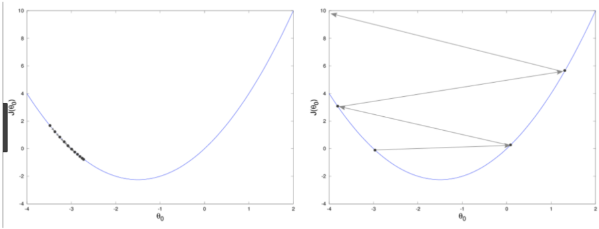
which is very clean looking, but then it is hard to note things like an iteration dependent learning rate.

Let's use both notations.

“Gradient descent” in one dimension:

This picture shows “gradient” descent in 1-d and illustrates the role of the learning rate.

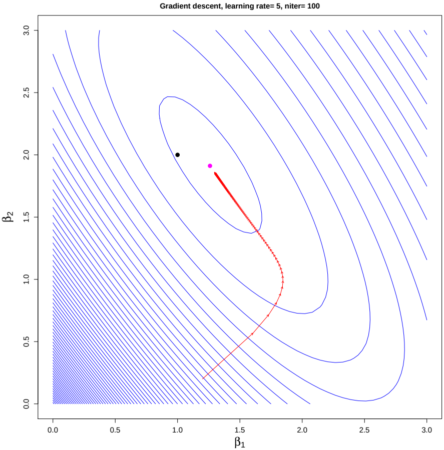
$$x \to x - \epsilon_k f'(x)$$



At left we have a small fixed ϵ_k.

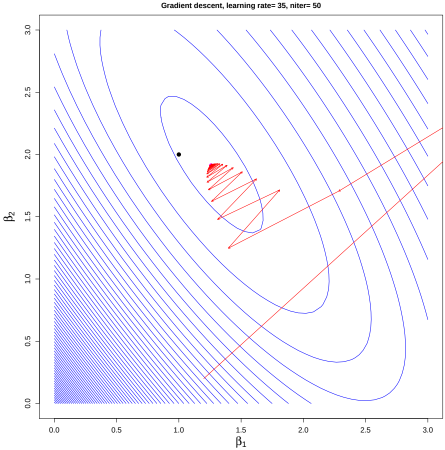
At right we have a big fixed ϵ_k.

Here is an example using data simulated from logit model with 2 correlated x variables and no intercept.



The learning rate is small in that we make small moves, it works, but it might take a while to get there.

Same data but a larger (fixed) learning rate.



With correlated *x* variables, -logL has narrow contours so that gradient descent goes in the wrong direction.

We have *zig-zagging !!*

7. Stochastic Gradient Descent

In statistical/machine learning applications our objective function often has the particular structure that is a sum over observations in our training data.

For our logit example

$$L(\theta) = \frac{1}{n} \sum_{i=1}^{n} (-\log(p(y_i|x_i,\theta)))$$

where "$\theta = \beta$".

$-\log(p(y_i|x_i,\theta))$ measures our error or *loss* at training observation (x_i, y_i) and parameter θ.

We seek a θ that minimizes our average (or total) loss on the training data.

More generally, we can formulate a problem with an action $f(x,\theta)$ and a loss $L(f(x,\theta),y)$ which depends on our action and the outcome y.

In our logit example, $f(x,\theta) = p(y=1|x,\theta)$.
Our loss is $-\log(f(x,\theta))$ if $y=1$ and $-\log(1-f(x,\theta))$ if $y=0$.

In our general problem, we see to minimize the average (or total) loss on the training data:

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n} L(f(x_i,\theta),y_i)$$

In this case our gradient is

$$\nabla J(\theta) = \frac{1}{n} \sum_{i=1}^{n} \nabla L(f(x_i, \theta), y_i)$$

We can also think of this as an estimate of the expected gradient under the joint distribution of (X, Y).

Recall that for our logit example, we did exactly this, that is we differentiated for each observation and then added them up.

Stochastic Gradient Descent

Rather then using all the data to estimate the gradient and then making a move, we divide the data up into "batches" and make a move for each gradient estimate from each batch.

We divide the training data up into K batches.

Let F_k be the set of indices for the k^{th} batch, $k = 1, 2, \ldots, K$.

That is

$$\cup_{k=1}^{K} F_k = \{1, 2, \ldots, n\}, \;\; F_k \cap F_l = \emptyset, k \neq l.$$

where n is the number of observations in the training data.

Let n_k be the number of observations in batch k.

```
for each epoch:
    For each batch $k$:
        $\hat{g} = \frac{1}{n_k} \sum_{i \in F_k} \nabla L(f(x_i, \theta), y_i)$
        $\theta \leftarrow \theta - \alpha \, \hat{g}$.
```

Here, an *epoch* is a pass through the entire data set.

You could try a fixed number of epochs, or keep going until you have converged.

For example, here is the python code to fit a neural net in keras.

You have to choose:

- the loss
- the optimizer, which corresponds to the learning rate
- the size of the batches
- the number of epochs

```
#compile model
nn1.compile(loss=’mse’,optimizer=’rmsprop’,metrics=[’mean_absolute_error’])

# fit
nepoch = 1500
nhist = nn1.fit(Xtr,ytr,epochs=nepoch,verbose=1,batch_size=32,
  validation_data=(Xte,yte))
```

The batch scheme does not have to be a partition, but that seems to be a common simple way to make up the batches.

More generally, you just need a scheme for selecting the next batch and you keep updating θ using the gradient information from each batch. But usually you schedule the batches so that you pass through the entire data set with a batch sequence so that you have an epoch.

Commonly the learing rate is decreased as you iterate and there are a variety of schemes:

$$\alpha_t = (1 - \frac{t}{\tau})\alpha_o + \frac{t}{\tau}\,\alpha_\tau, t = 1, 2, \ldots, \tau, \;\; \alpha_t = \alpha, t > \tau.$$

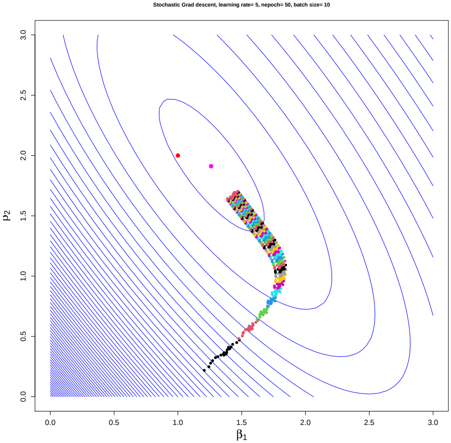
$$\alpha_t = \alpha_o \; \text{exp}(-kt).$$

$$\alpha_t = \frac{\alpha_o}{1 + kt}$$

We will see that there a more sophisticated adjustments of the descent such as momentum.

Here is 50 epochs with batch size 10.

The epochs are color coded.



8. Momentum

Since gradient descent just depends on the first derivative in can encounter a variety of problems:

- slow down in flats.
- stuck in local min/max.
- zig zag.

A variety of methods use the history of gradients to improve performance.

This may be done at the level of an individual parameter. Parameters may have large partial derivatives associated with zig-zagging.

8.1. Momentum

The goal is to modify simple gradient descent so that we are able to avoid slowing down in local flat spots and get out of local mins.

Implicitly, we use second order information by using information from previous gradient evaluation.

But we do it in a computationally cheap way.

Momentum based methods address the problems with local optima, flat spots, and zigzagging by incorporating the overall direction of past moves.

Our next step is the a weighted combination of the previous step and the current gradient information.

$$v_t = \gamma\, v_{t-1} - \eta\, \nabla_\theta J(\theta_{t-1})$$

$$\theta_t = \theta_{t-1} + v_t$$

v_{t-1} will capture information from past gradients.

γ is the "momentum parameter" or the "friction parameter".

$\gamma \in [0, 1)$, e.g. .8. $\gamma = 0$ is simple gradient descent.

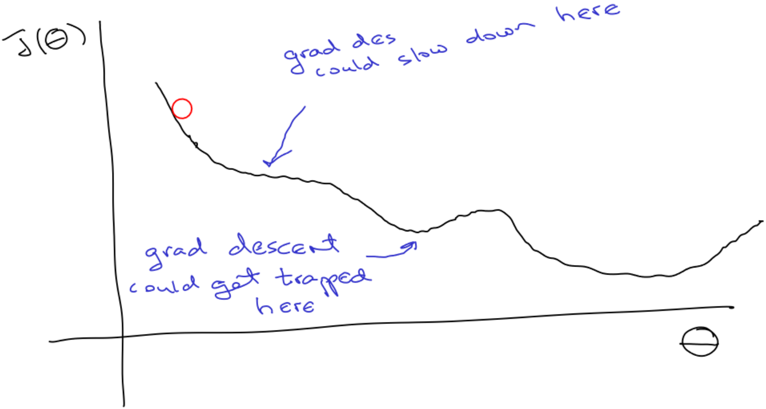
Useful to look at this by simply replacing v_{t-1} with $\theta_{t-1} - \theta_{t-2}$.

$$\theta_t = \theta_{t-1} + v_t$$

$$= \theta_{t-1} + \gamma\, v_{t-1} - \eta\, \nabla_\theta J(\theta_{t-1})$$

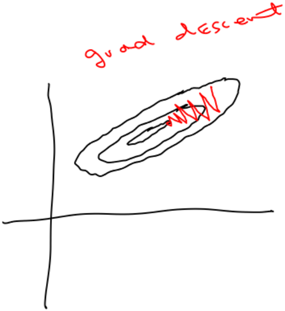
$$= \theta_{t-1} + \gamma\, (\theta_{t-1} - \theta_{t-2}) - \eta\, \nabla_\theta J(\theta_{t-1})$$

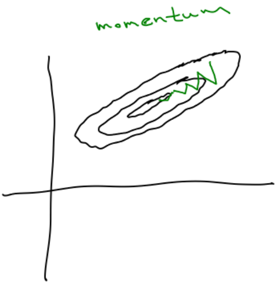
Think of a marble rolling down a surface, it's momentum will enable it roll past local shallow spots.


J(Θ)
grad des could slow down here
grad descent could get trapped here
Θ

We use our overall, consistent direction rather than the one totally dependent on our current position.

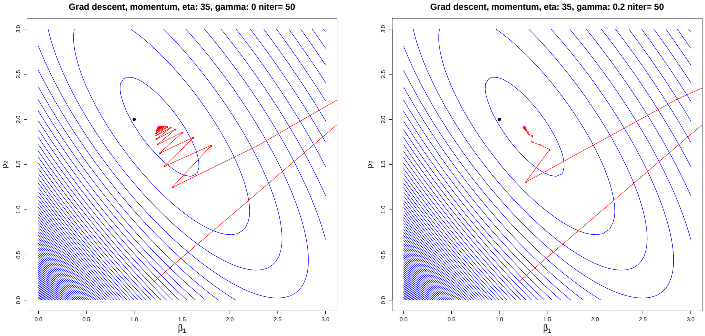
Momentum captures the overall direction, mitigating the the zig zag you get with contours that are long and narrow.





Left: gradient descent

Right: with momentum.



Obviously, how you intitialize v is an issue.

Here I started at $v_0 = 0$.

8.2. AdaGrad and RMSProp

AdaGrad

This approach will tailor the update mechanism to individual parameters.

The idea is that zig-zagging in the θ_j in basic SGD will be driven by large values of $(\frac{\partial J(\theta)}{\partial \theta_j})^2$.

In AdaGrad we keep track of the sum of the values over iterations and use it to adjust our step size on an individual parameter basis.

$$A_{it} = A_{i,t-1} + (\frac{\partial J(\theta_{t-1})}{\partial \theta_i})^2$$

$$\theta_{it} = \theta_{i,t-1} - \frac{\alpha}{\sqrt{A_{it}}} \frac{\partial J(\theta_{t-1})}{\partial \theta_i}.$$

- Initialize A at 0.
- Use $\sqrt{A_{it} + \varepsilon}$ instead of $\sqrt{A_{it}}$.

RMSProp

First note:

Observe sequence of values y_t which depend on previous y_{t-1} and "new information" ϵ_t.

$$y_t = \rho y_{t-1} + (1-\rho)\varepsilon_t$$

Start at y_0

$$y_1 = \rho y_0 + (1-\rho)\varepsilon_1$$

$$y_2 = \rho^2 y_0 + \rho(1-\rho)\varepsilon_1 + (1-\rho)\varepsilon_2$$

$$y_3 = \rho^3 y_0 + \rho^2(1-\rho)\varepsilon_1 + \rho(1-\rho)\varepsilon_2 + (1-\rho)\varepsilon_3$$

$$y_4 = \rho^4 y_0 + \rho^3(1-\rho)\varepsilon_1 + \rho^2(1-\rho)\varepsilon_2 + \rho(1-\rho)\varepsilon_3 + (1-\rho)\varepsilon_4$$

$$y_t = \rho^t y_0 + \sum_{j=0}^{t-1} \rho^j (1-\rho)\varepsilon_{t-j}$$

The current y is an exponential smooth of the past ϵ.

RMSProp

$$A_{it} = \rho\, A_{i,t-1} + (1-\rho)\,(\frac{\partial J(\theta_{t-1})}{\partial \theta_i})^2$$

$$\theta_{it} = \theta_{i,t-1} - \frac{\alpha}{\sqrt{A_{it}}}\,\frac{\partial J(\theta_{t-1})}{\partial \theta_i}.$$

- Initialize A at 0.
- $\rho \in (0,1)$.
- Use $\sqrt{A_{it}+\varepsilon}$ instead of $\sqrt{A_{it}}$.

8.3. Adam

Adam builds on AdaGrad and RMSprop.

RMSprop has an exponential smoothed summary of the past squared partial derivates.

Adam adds an exponential smoothed summary of the past partial derivates giving a momentum type flavor.

In addition, an iteration dependent learning rate is used to dampen the initial moves which may be suspect due to the initialization of the smoothed values.

Adam

$$A_{it} = \rho\, A_{i,t-1} + (1-\rho)\,(\frac{\partial J(\theta_{t-1})}{\partial \theta_i})^2$$

$$F_{it} = \rho_f\, F_{i,t-1} + (1-\rho_f)\,\frac{\partial J(\theta_{t-1})}{\partial \theta_i}$$

$$\theta_{it} = \theta_{i,t-1} - \frac{\alpha_t}{\sqrt{A_{it}}}\, F_{it}$$

$$\alpha_t = \alpha\,\frac{\sqrt{1-\rho^t}}{(1-\rho_f^t)}.$$

- Initialize A and F values at 0.
- ρ, $\rho_f \in\ (0,1)$, e.g. $\rho = .9$, $\rho_f = .999$.
- Use $\sqrt{A_{it}+\varepsilon}$ instead of $\sqrt{A_{it}}$.

Note that

$$\rho^t,\ \rho_f^t\ \to 0,\ \text{as } t \to \infty.$$

so that the effect on α_t goes away as the number of iterations increases.

This adjustment is designed to migitgate the effects of early values of A and F due to the crude intialization.

8.4. Momentum and SGD

With SGD we can make our adjustments on a batch basis.

For example, SGD with momentum would be:

Let n_k be the number of observations in batch k.

Initialize v, choose γ, η.

```
for each epoch:
    For each batch $k$:
        $\hat{g} = \frac{1}{n_k} \sum_{i \in F_k} \nabla L(f(x_i, \theta), y_i)$
        $v \leftarrow \gamma v - \eta \hat{g}$
        $\theta \leftarrow \theta + v$
```

9. Newton's Method

Newton's method is iterative.

Let β_i the value at iteration i.

- approximate f at β_i by a quadratic using Taylors's theorem.
- optimize the quadratic: the solution is β_{i+1}.
- repeat until converged.

Taylor approximation:

$$f(\beta) \approx \tilde{f}(\beta) = f(\beta_i) + f'(\beta_i)(\beta - \beta_i) + \frac{1}{2}(\beta - \beta_i)' f''(\beta_i)(\beta - \beta_i)$$

Now to optimize the quadratic, we compute its gradient and set it equal to 0.

$$\nabla \tilde{f}(\beta) = f'(\beta_i) + (\beta - \beta_i)' f''(\beta_i)$$

We can solve $\nabla \tilde{f}(\beta) = 0$ with

$0 = f'(\beta_i) + (\beta - \beta_i)' f''(\beta_i)$

$-f'(\beta_i)[f''(\beta_i)]^{-1} = \beta' - \beta_i'$

$\beta' = \beta_i' - f'(\beta_i)[f''(\beta_i)]^{-1}$

$$\beta_{i+1} = \beta_i - [f''(\beta_i)]^{-1}[f'(\beta_i)]'$$

Newton's method and the logit mle

We will maximize the the logit log-likelihood.

$$\begin{aligned}\beta_{i+1} &= \beta_i - [-X'DX]^{-1}X'(y - F_v)\\ &= \beta_i + [X'DX]^{-1}X'(y - F_v)\end{aligned}$$

Iteratively Reweighted Least Squares

Recall weighted least squares

$$Y = X\beta + \epsilon, \;\; \epsilon \sim N(0, \Sigma)$$

then,

$$\hat{\beta} = (X'\Sigma^{-1}X)^{-1}X'\Sigma^{-1}y$$

It may be helpful to rewrite the Newton iteration as a series of weighted regressions:

Let $\Sigma^{-1} = D$ and

$$Z = X\beta_i + D^{-1}(y - F_v)$$

then,

$(X'\Sigma^{-1}X)^{-1}X'\Sigma^{-1}Z = (X'\Sigma^{-1}X)^{-1}X'\Sigma^{-1}(X\beta_i + D^{-1}(y - F_v))$

$= \beta_i + [X'DX]^{-1}X'(y - F_v)$

Hence doing an iteratively (re)weighted least squares problem (IRLS) gets you the mle.