Machine Learning I

Lecture 6: Iterative Methods

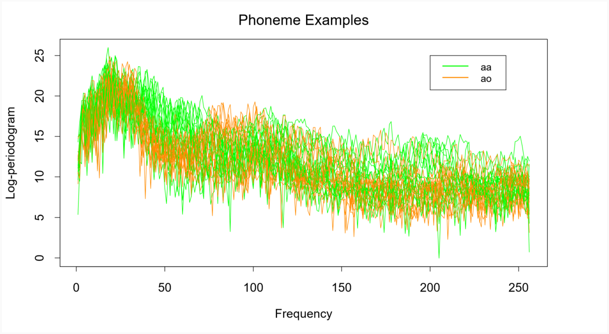
Nathaniel Bade

Northeastern University Department of Mathematics

Table of contents

The need for iterative methods

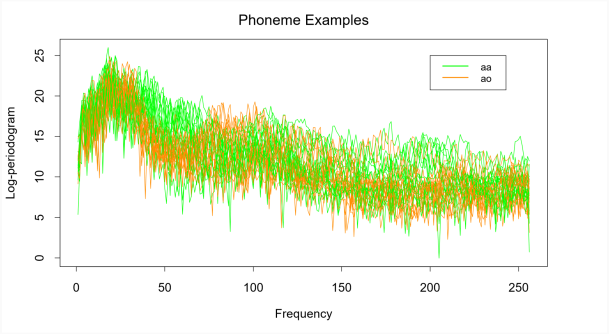
More complicated models



Not everything in the world is linear. We saw last lecture that polynomial functions can be fit by performing linear regression on synthetic features $X_{i_1}^{d_1} \dots X_{i_r}^{d_r}$. The obvious problem is that degree d polynomials on p features lead to

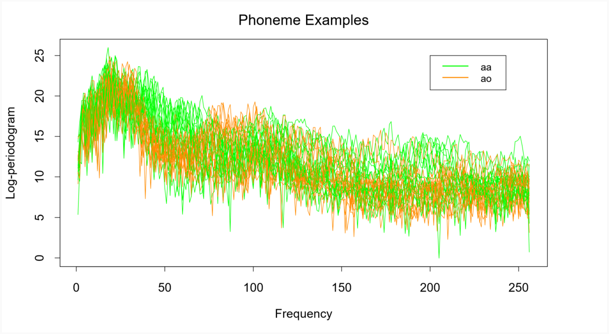
$$\textbf{Number of Features:} \qquad \binom{d+p}{d}.$$

More complicated models



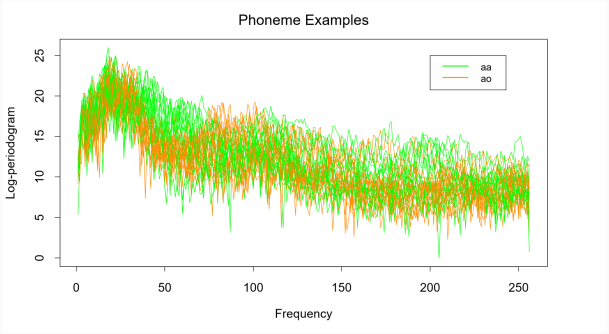
As an example, on the MNIST data set there are 308,505 degree 2 polynomials and 80,931,145 degree three polynomials. Without a principled reason, inverting a matrix to solve such problems quickly becomes prohibitive.

More complicated models



How can we fit such models when the functions are too complicated to yield a close form solution for the minimia? When the number of variables is too large to efficiently use linear methods? When the models themselves may not have closed form formula?

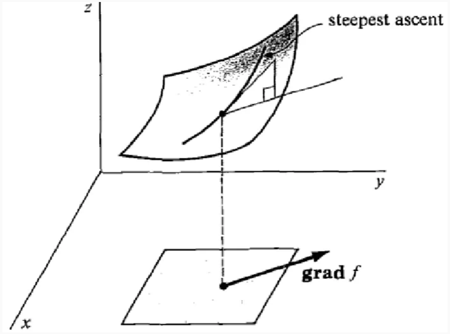
More complicated models



In this lecture we're going to being talking about how to fit more complicated functions. We will start by canvasing several methods for our linear classifiers, and then show how to extend these results to polynomials and exponential functions.

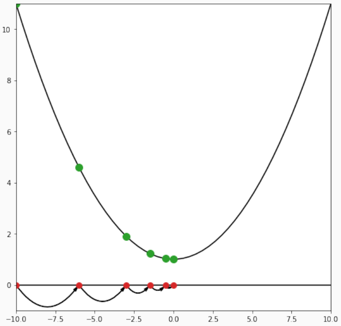
Gradient Decent

Gradient Decent



Gradient decent is an incredibly general method of finding the minima of functions. Lets recall some facts from calculus: Let $F(x)$ be a differentiable function depending on a vector of parameters x, so $F : \mathbb{R}^k \to \mathbb{R}$. Then, the gradient ∇F always points in the direction of greatest increase.

Gradient Decent

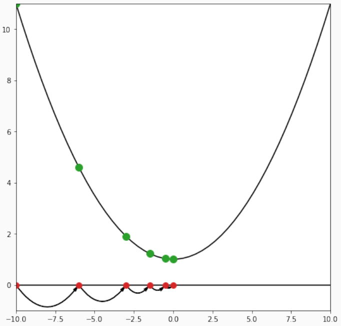


This means that if we want to find a minimum, we can follow $-\nabla F$ downhill. Concretely, for η sufficiently small,

$$F(x - \eta \nabla F) \approx F(x) - \eta \, \nabla F \cdot \nabla F = F(x) - \eta \, ||\nabla F||^2 < F(x) \, .$$

If F is always positive and not at a minimum, than a sufficiently small step in the ∇F direction will always decrease F.

Gradient Decent

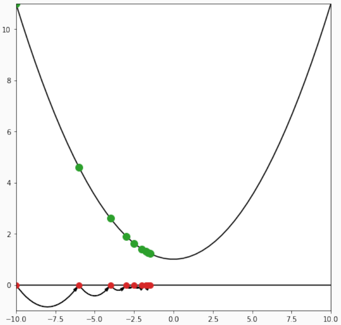


Algorithmically, at each $x^{(n)}$ we compute the next step by

$$x^{(n+1)} = x^{(n)} - \eta \nabla F \, ,$$

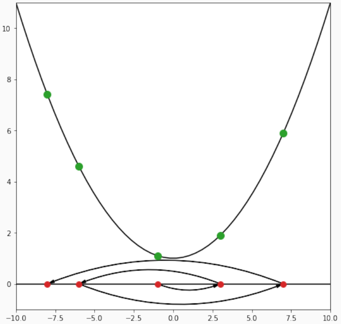
where $\eta > 0$ is the **learning rate** hyperparameter.

Gradient Decent



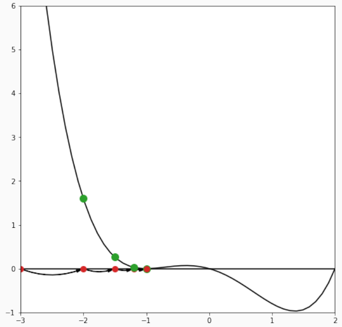
The learning rate η determines proportionally how far of a step we take in the $-\nabla F$ direction. If η is too small we may find that the algorithm appears to converge but does so to a point that isn't an actual minimum.

Gradient Decent



On the other hand, if η is too large we may actually walk away from the minimum. We will see an example for linear regression explicitly.

Gradient Decent



Of course, gradient decent is a greedy method and so if η is too small it may miss a global minimum in favor of a local one. If we want to guarantee that gradient decent can actually find the minimum, we will need some extra assumptions on our hypothesis class and loss function. RSS for Linear Regression turns out to only have a single minimum.

Batch Gradient Decent

If we compute the gradient using all of the training data it is called **batch gradient decent**. For mean squared error,

$$MSE(\beta) = \frac{1}{N}(\mathbf{y} - \mathbf{X}\beta)^T(\mathbf{y} - \mathbf{X}\beta)$$

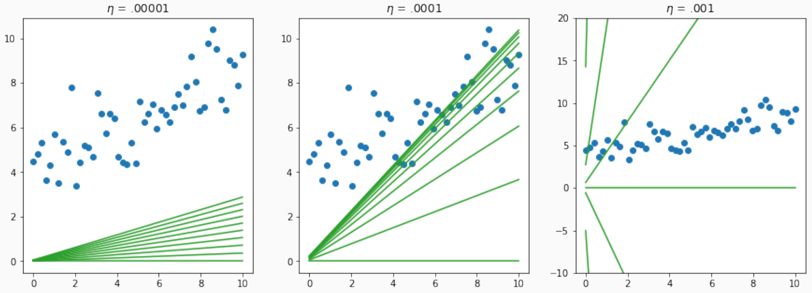
a formula is readily available. Choosing to represent $\nabla_\beta MSE(\beta)$ as a $p+1$ column vector, the gradient is

$$\nabla_\beta MSE = -\frac{2}{N}\mathbf{X}^T(\mathbf{y} - \mathbf{X}\beta)\,.$$

The update step for the parameters β is

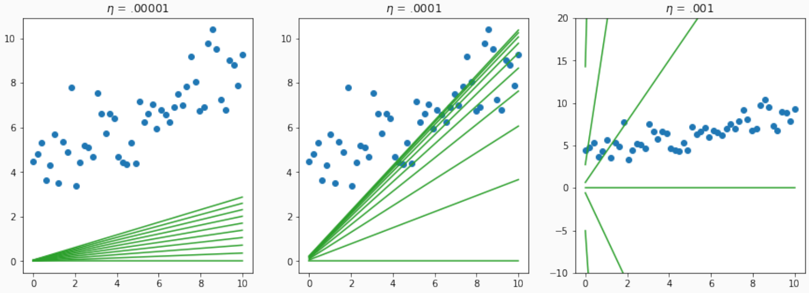
$$\beta^{(n+1)} = \beta^{(n)} + \eta\frac{2}{N}\mathbf{X}^T(\mathbf{y} - \mathbf{X}\beta^{(n)})\,.$$

Batch Gradient Decent



We see the results for MSE and linear regression above. If η is too low, convergence is slow (possibly to the point of being undetectable). If it is too large, the fit is actually repelled by the minimum.

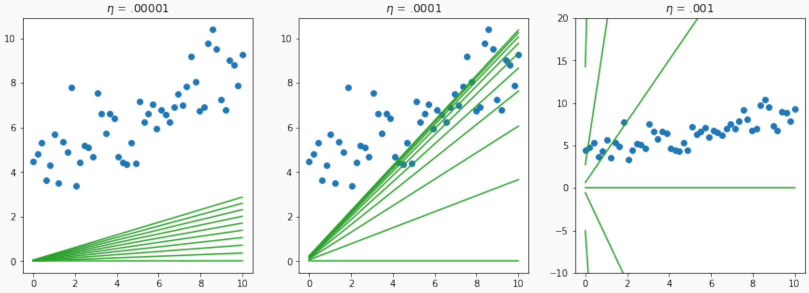
Batch Gradient Decent



In addition to η, we must decide how many iterations we will use. Optimally, we would like to find a learning rate that causes convergence in a small number of iterations.

The learning rate can be fixed (say by performing a grid search to find the best η) or variable, starting large and then lowering to attempt to avoid both falling into and being repelled by local minimia.

Batch Gradient Decent



Similarly, the number of iterations can be fixed, or we can choose to stop when, say, $||\nabla\beta|| < \epsilon$ for some **tolerance** ϵ.

It can be show that for *MSE* with fixed η, the convergence rate is approximately $O\left(\frac{1}{\text{itter}}\right)$ .That is, to decrease the tolerance ϵ by $\frac{1}{2}$ we must double the number of iterations.

Checklist for implementing gradient decent

Checklist for implementing gradient decent:

Let $f(X, \beta)$ be a positive function that depends on data $\mathbf{X}$ and parameters β. Let $F(\mathbf{X}, \beta) = \sum_{i=1}^{N} f(x_i, \beta)$. Then GD on F is

Step 1: Find an expression for $\nabla_\beta F$.

Step 2: Set an initial guess for $\beta^{(0)}$ and fix a learning rate η.

Step 3: For each n, compute $\beta^{(n+1)} = \beta^{(n)} - \eta \nabla_\beta F$

Step 4: Stop when $\eta ||\nabla_\beta F||$ is sufficiently small, after a fixed number of steps, of when F is sufficiently small. Alternatively, define a **learning schedule** for η which gradually lowered η as n gets large.

Stochastic Gradient Decent

Batch Gradient Decent

The main problem with batch gradient decent is that all the training data is used to compute the parameters on each iteration. This slows down training considerably when the dataset is large.

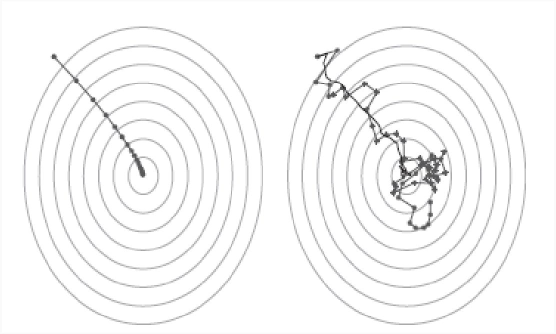
At the opposite end, **Stochastic Gradient Decent** (SGD) picks a random element of the training set and uses that to update, like the perception algorithm. SGD a much faster than batch GD since it only needs a single element to train on. The tradeoff is it is much more random.

If $f(X, \beta)$ is a positive function that depends on data $\mathbf{X}$ and parameters β. Let $F(\mathbf{X}, \beta) = \sum_{i=1}^{N} f(x_i, \beta)$. Then SGD updates $\beta^{(n)}$ by randomly sampling $x \sim \mathbf{X}$ and computing

$$\beta^{(n+1)} = \beta^{(n)} - \eta \nabla f(x) .$$

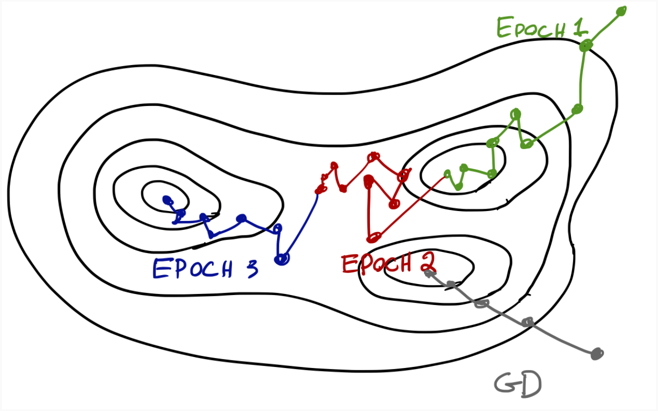
Of course, β will take many more steps to converge, but the computation at each step will be much simpler.

Batch Gradient Decent



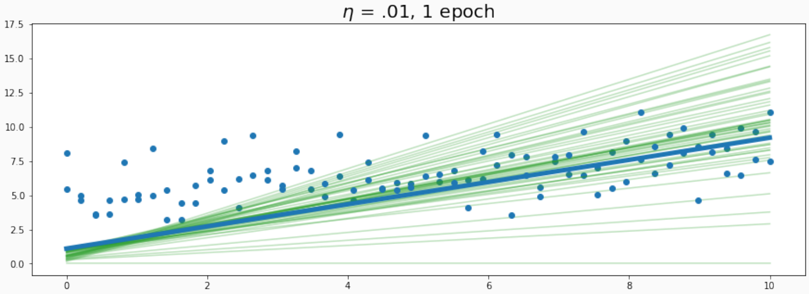
Comparing gradient decent and stochastic gradient decent, gradient decent will be much more stable, usually plodding along directly towards to nearest local minimum and getting stuck there. SGD on the other hand is much more random, and may walk towards or away at each step.

Stochastic Gradient Decent



SGD is much more likely to avoid getting stuck in local minimia, but on the other hand it doesn't tend to settle down (even in a global minimum). As a result it is almost necessary to gradually lower the learning rate η as we train. The function that determines η is called the **learning schedule**.

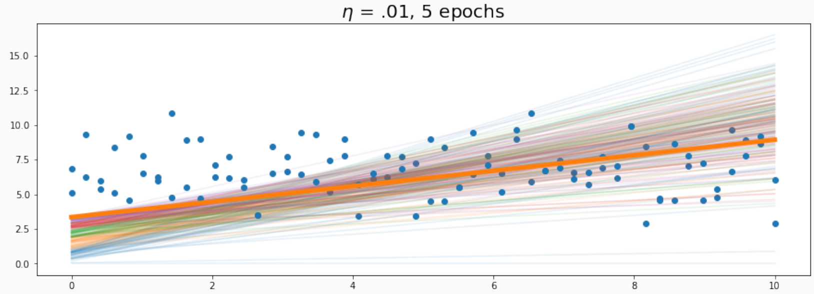
Stochastic Gradient Decent



Training on the entire data set all at once (GD) or one at a time (SGD) called a **training epoch**. It is customary to train multiple epochs, resetting the learning schedule each time.

Although this produces many training steps, each step is computationally simple.

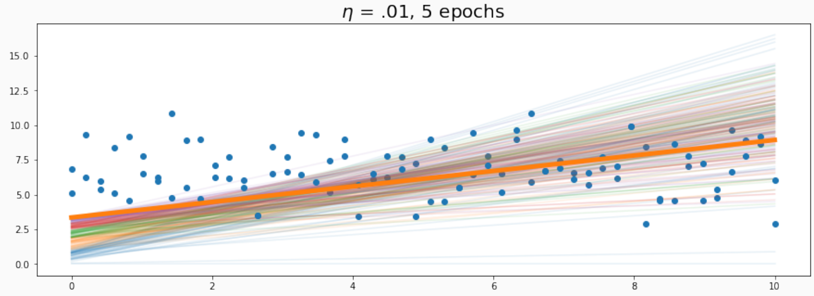
Stochastic Gradient Decent



Finally **minibatch gradient decent** splits the difference between SGD and GD, training in epochs of m subsets of size n, where $m \times n = N$. All three methods are implemented in sci-kit learn and (for convex problems) yield the same results.

The batch size m provides a nonlinear trade off between the number of iterations to convergence and the number complexity of each iterative step.

Stochastic Gradient Decent



In the above, the lines are colored by epochs. Notice that after 5 epochs we are closer to the true center line, but it will take about 10 more epochs to get within 1% of the RSS fit.

Checklist for implementing batch gradient decent

Checklist for implementing batch gradient decent or SGD:

Let $f(X, \beta)$ be a positive function that depends on data $\mathbf{X}$ and parameters β. Let $m < N$ be the batch size and let $F_M(\mathbf{X}, \beta) = \sum_{i=1}^{M} f(x_i, \beta)$. Then GD on F is

Step 1: Find an expression for $\nabla_\beta F_M$.

Step 2: Set an initial guess for $\beta^{(0)}$ and fix a learning rate η.

Step 3: Shuffle $\mathbf{X}$ and split it into k batched of M unique data points denoted $\mathbf{X}_j$. For each n, compute $\beta^{(kn+j)}$ by $\beta^{(n+1)} = \beta^{(n)} - \eta \nabla_\beta F_M(\mathbf{X}_j)$.

Step 4: Stop when $\eta||\nabla_\beta F||$ is sufficiently small, after a fixed number of steps, of when F is sufficiently small. Alternatively, define a **learning schedule** for η which gradually lowered η as n gets large.

Newtons Method

Newtons Method

Newtons Method is an iterative method of finding zeros of differential functions. By applying it to the first derivative of a function we can use it to find minima, maxima and saddle points.

Newtons method is considerably faster than gradient decent but relies on finding second derivatives of the loss function. This means that not only must the loss function be twice differentiable, but a formula should be available for all second derivatives.

As a result, Newtons method isn't as widely used as GD, SGD and other solvers, but when it can be implemented it tends to be superior.

Newtons Method

In one variable, we try to find a zero of $f(x)$ by successive approximations of $f(x)$ by it's Taylor polynomial.

Starting with x, we try to find an improved guess $x+\eta$ by Taylor expanding $f(x+\eta)$ around x:

$$f(x+\eta) \approx f(x) + \eta f'(x) = 0\,.$$

Solving for $\delta = -f(x)/f'(x)$, we have an "improved" guess $x_{new} = x - f(x)/f'(x)$ for the location of the zero. We then iterate until we have arrived at a zero. It can be proved that under reasonable assumptions on $f(x)$ we always will converge to a zero.

Multivariate Newtons Method

For higher dimensional functions, we proceed exactly as before: Let $f(x)$ be differentiable. Expanding $f(x+\eta)$ around x, we find

$$f(x+\eta) \approx f(x) + J(x)\eta = 0\,,$$

where $J_{ij} = \frac{\partial f_i}{\partial x_j}$ is the Jacobean matrix (gradient if $f(x) \in \mathbb{R}$). If $J(x)$ is invertible, we can solve for

$$x_{new} = x - J^{-1}(x)f(x)\,.$$

Note, unless $f : \mathbb{R}^d \to \mathbb{R}^d$ $J(x)$ will almost certainly not be even partially invertible.

Optimization via Newtons Method

For a single valued multivariate function $f(x)$, we can use Newtons method to perform optimization. Since optimization is just finding $\nabla f(x) = 0$, we write

$$\nabla f(x+\eta) \approx \nabla f(x) + H_f(x)\eta = 0$$

where $H_f(x) = (\frac{\partial f}{\partial x_i \partial x_j})_{ij}$ is the Hessian matrix. At each iteration then we update x_{old} to

$$x_{new} = x_{old} - H_f^{-1}(x)\nabla f(x)\,.$$

Note, Newtons Method converges much faster than gradient decent, but requires computing higher derivatives which might not exist, or may be hard to compute. As a result, it often cannot be used.

Example: Linear Regression Via Newtons Method

Lets solve linear regression using Newtons Method. The gradient of $MSE = \frac{1}{N}(\mathbf{y} - \mathbf{X}\beta)^T(\mathbf{y} - \mathbf{X}\beta)$ is

$$\nabla MSE = -\frac{2}{N}\mathbf{X}^T(\mathbf{y} - \mathbf{X}\beta)\,,$$

and the Hessian is the $(p+1) \times (p+1)$ matrix

$$H_{MSE} = \frac{2}{N}\mathbf{X}^T\mathbf{X}\,.$$

Given an initial $\beta^{(0)}$, the update rule is

$$\beta^{(n+1)} = \beta^{(n)} + (\frac{2}{N}\mathbf{X}^T\mathbf{X})^{-1}\frac{2}{N}\mathbf{X}^T(\mathbf{y} - \mathbf{X}\beta^{(n)}) = \beta^{(n)} + (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T(\mathbf{y} - \mathbf{X}\beta^{(n)})\,.$$

Question: How many iterations does Newtons method take to arrive at the answer?

Fitting Logistic Regression

Recall Logistic Regression on a binary labeling. Let y_i take probabilities $k \in \{0, 1\}$. Under the logistic assumption we set

$$\delta_1(x) = \frac{\exp(x^T\beta)}{1 + \exp(x^T\beta)}, \quad \delta_0(x) = \frac{1}{1 + \exp(x^T\beta)},$$

where we have again absorbed β_0 into β. Then

$$\begin{aligned} \ell(\beta) &= \sum_{i=1}^{N} \log \mathbb{P}(G = y_i | X = x_i; \beta) \\ &= \sum_{i=1}^{N} y_i \log(\delta_1(x_i)) + (1 - y_i) \log(1 - \delta_0(x_i)) \\ &= \sum_{i=1}^{N} y_i x_i^T \beta - \log(1 + e^{x_i^T \beta}). \end{aligned}$$

Fitting Logistic Regression

Denoting by $\mathbf{d}$ the vector whose i'th coordinate is $\delta_1(x_i)$ fitted to β^{old}, the gradient of

$$\ell(\beta) = \sum_{i=1}^{N} y_i x_i^T \beta - \log(1 + e^{x_i^T \beta}$$

is

$$\nabla \ell(\beta) = \mathbf{X}^T (\mathbf{y} - \mathbf{d}) \,.$$

Letting $\mathbf{W}$ be the $N \times N$ diagonal matrix with i'th entry $\delta_1(x_i)$ fitted to β^{old}, the Hessian is

$$H = \frac{\partial^2 \ell(\beta)}{\partial \beta \partial \beta^T} = -\mathbf{X}^T \mathbf{W} \mathbf{X} \,.$$

Then

$$\beta^{new} = \beta^{old} + (\mathbf{X}^T \mathbf{W} \mathbf{X})^{-1} \mathbf{X}^T (\mathbf{y} - \mathbf{p}) \,.$$

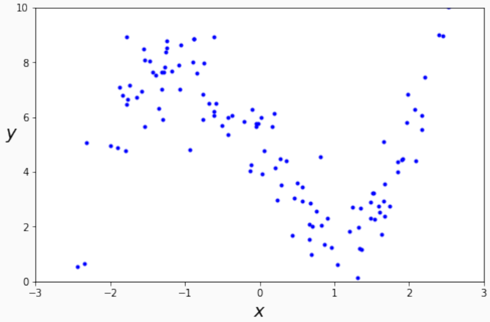
Fitting Logistic Regression

Part of the power of logistic regression comes from the fact that Newtons Method can be directly applied. In addition, it is an explanatory tool, since in general it's coefficients have readily available, explicit interpretations.

Indeed, Newtons Method is often used in data analysis and situation in which one would like to actually explain outcomes, there tends to be a tight correlation between models with explicable parameters and models with computable derivatives.

Example: Polynomial Fitting

Polynomial Regression

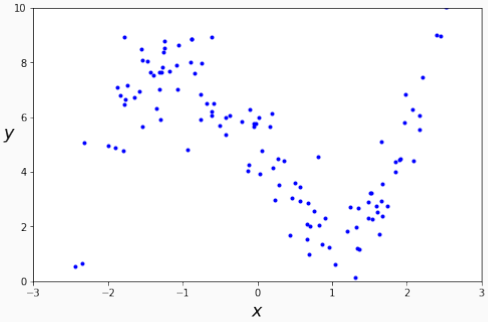


Let us now consider the problem of fitting a polynomial function to a dataset with numeric labels. In one variable, a degree d polynomial can be written

$$Y = \beta_0 + \beta_1 X + \beta_2 X^2 + \ldots + \beta_d X^d \,,$$

where β_ℓ are constant parameters to be fit.

Polynomial Regression



The RSS loss function takes the form

$$RSS(\beta) = \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 = \sum_{i=1}^{N} (y_i - \hat{\beta}_0 - \hat{\beta}_1 X - \ldots - \hat{\beta}_d X^d)^2$$

We can fit a polynomial function by defining a loss function and solving iteratively or fit it by solving for the derivative (although the derivative takes care to put in a linear form).

Polynomial Regression

The other option is to engineer new features. Let $\mathbf{X}^{poly}$ be the matrix whose columns are compute from the monomials in the columns of $\mathbf{X}$:

$$\mathbf{X}^{poly} = \begin{bmatrix} x_1 & x_1^2 & x_1^3 & \dots & x_1^d \\ \vdots & \vdots & \vdots & & \vdots \\ x_N & x_N^2 & x_N^3 & \dots & x_N^d \end{bmatrix}.$$

The polynomial fitting on $\mathbf{X}$ can now be written in terms of a linear fitting on $\mathbf{X}^{poly}$. Writing $X^{poly} = [X, X^2, \dots, X^d]$, we can write the polynomial model as

$$Y = \beta_0 + \beta_1 X + \ldots + \beta_d X^d = \beta^T X^{poly}.$$

Polynomial Regression

The RSS loss

$$RSS(\beta) = \sum_{i=1}^{N} (y_i - \hat{\beta}_0 - \hat{\beta}_1 X - \ldots - \hat{\beta}_d X^d)^2 = (\mathbf{y} - \mathbf{X}^{poly}\beta)^T (\mathbf{y} - \mathbf{X}^{poly}\beta)$$

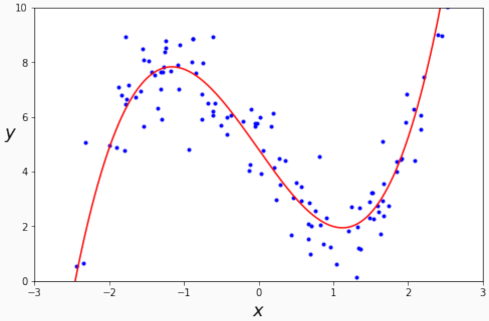
is minimized explicitly by $\beta = (\mathbf{X}^{poly\ T}\mathbf{X}^{poly})^{-1}\mathbf{X}^{poly\ T}\mathbf{y}$.

This are some huge advantages to this:

The monomial vectors only need to be computed once.

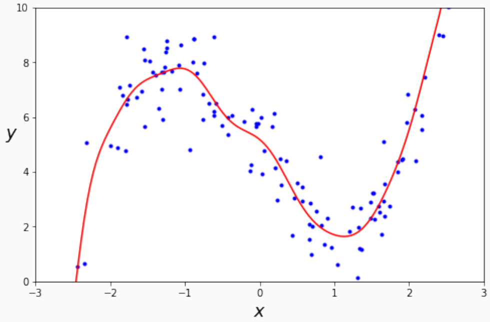
This is a unified expression for all orders of polynomial.

Polynomial Regression



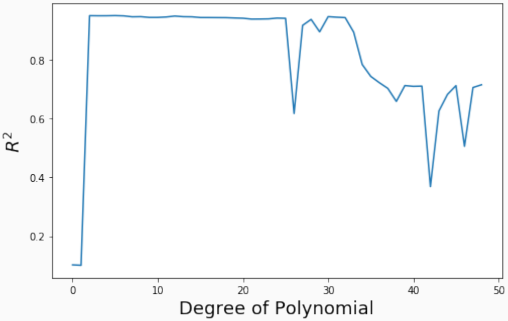
Performing linear regression on the constructed features in $\mathbf{X}^{poly}$ yields a polynomial model as above. Of course in most cases we will not know the optimal degree of the polynomial in question. In these cases (computational time permitting) we perform a train/test split and tune d using the standard cross validation.

Polynomial Regression



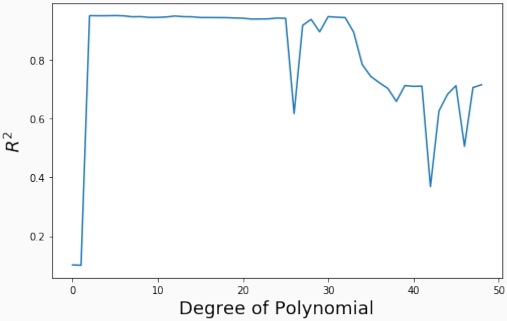
Performing linear regression on the constructed features in $\mathbf{X}^{poly}$ yields a polynomial model as above. Of course in most cases we will not know the optimal degree of the polynomial in question. In these cases (computational time permitting) we perform a train/test split and tune d using the standard cross validation.

Polynomial Regression



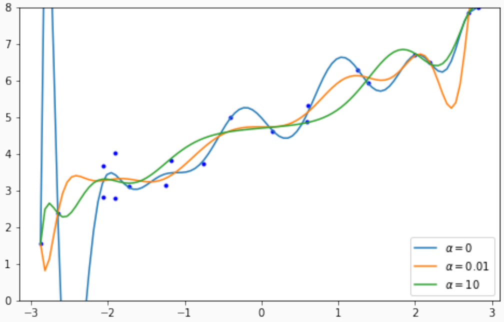
Testing on validation data, we see that the R^2 value reaches a maximum when the degree of the model matches the degree of the generated data. It then sticks there until it has enough degrees of freedom to begin badly overfitting. In general, this overfitting is a function of the number of datapoints and the number of dimensions.

Polynomial Regression



Testing on validation data, we see that the R^2 value reaches a maximum when the degree of the model matches the degree of the generated data. It then sticks there until it has enough degrees of freedom to begin badly overfitting. In general, this overfitting is a function of the number of datapoints and the number of dimensions.

Polynomial Regression



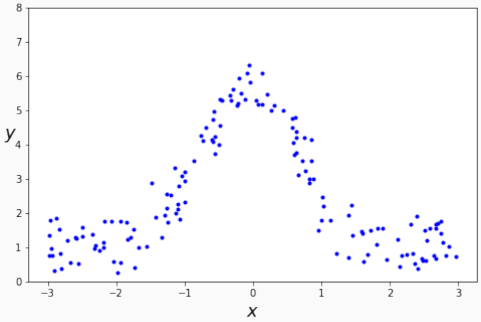
Polynomial fitting can also be used with Ridge and Lasso regression to regulate the size of the beta. In the example above, ridge regression

$$\text{Ridge} = \frac{1}{N}\text{RSS}(\beta) + \alpha\beta^T\beta\,,$$

is used to fit the higher degree polynomial, resulting in a dampening of the β terms.

Example: Fitting Nonpolynomial Functions

Fitting Gaussians



Lets consider the fitting of a function to roughly bell shaped dataset. Our proposed model will be the Gaussian exponential

$$Y = f(X) = \beta \exp\left(-\frac{(X-\mu)^2}{\sigma}\right) .$$

This function is not a probability distribution, so we can ignore the usual normalization.

Fitting Gaussians

Lets fit this function by minimizing the RSS. We will first try to find an exact minimum. For f, the RSS loss is

$$RSS(\beta, \sigma, \mu) = \sum_{i}^{N} \left(y_i - \beta e^{-\frac{(x_i - \mu)^2}{\sigma}} \right)^2 .$$

We set the parameter gradient to zero, yielding three expressions:

$$0 = \frac{\partial RSS}{\partial \beta} = -2 \sum_{i}^{N} e^{-\frac{(x_i - \mu)^2}{\sigma}} \left(y_i - \beta e^{-\frac{(x_i - \mu)^2}{\sigma}} \right)^2$$

$$0 = \frac{\partial RSS}{\partial \mu} = -4 \sum_{i}^{N} \beta \frac{(x_i - \mu)}{\sigma} e^{-\frac{(x_i - \mu)^2}{\sigma}} \left(y_i - \beta e^{-\frac{(x_i - \mu)^2}{\sigma}} \right)^2$$

$$0 = \frac{\partial RSS}{\partial \sigma} = -2 \sum_{i}^{N} \beta \frac{(x_i - \mu)^2}{\sigma^2} e^{-\frac{(x_i - \mu)^2}{\sigma}} \left(y_i - \beta e^{-\frac{(x_i - \mu)^2}{\sigma}} \right)^2$$

Fitting Gaussians

It turns out the first expression can be meaningfully solved for β. Indeed, we can always minimize

$$RSS(\beta) = (\mathbf{y} - \beta f(\mathbf{X}))^T(\mathbf{y} - \beta f(\mathbf{X}))$$

by solving

$$0 = f(\mathbf{X})^T(\mathbf{y} - \beta f(\mathbf{X}))$$

exactly as in the linear case. (**Exercise**: What is the solution?). However, the other two expressions are relatively intractable.

In general we you will not be able to find an expression for the derivative of the loss function for some arbitrary model, even if the model is differentiable. There are two options: Change the loss function, or use an iterative fitting method.

Polynomial Regression

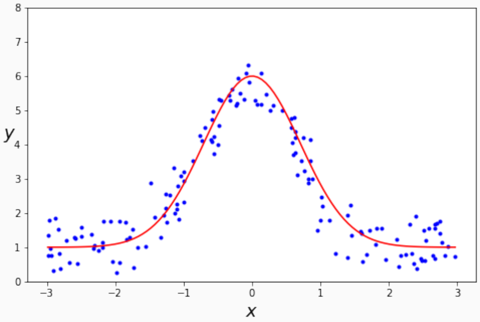
Gradient Decent:

For gradient decent we only need the expressions for the derivatives derived a few slides ago. Starting with a guess of the parameters, we fit the bell curve by computing itterativly. For $\theta = [\beta, \sigma, \mu]^T$,

$$\theta^{(n+1)} = \theta^{(n)} - \eta \frac{\partial RSS}{\partial \theta} ,$$

for some learning rate η.

Fitting Gaussians



Provided the derivatives can be computed, gradient decent will generally be usable, even for more complicated models. For example, the sum of d Gaussians model can be computed in the same manner, with $|\theta| = 2d$ parameters to be fit. For prohibitive amounts of data, SGD may be more time effective.

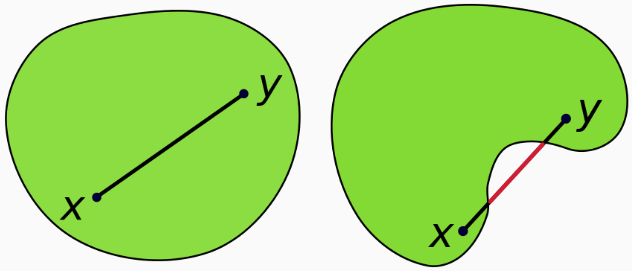
Convex Learning

Convex-Smooth-Lipschitz

Gradient decent is used widely throughout machine learning, mathematical modeling and optimization so we would like to nail down its effectiveness. In particular, is there a criteria for loss functions and hypothesis classes such that GD/SGD is guaranteed to converge?

It turns out the concepts we will want are **convexity**, **Lipschitzness** and **smoothness**.

Convex-Smooth-Lipschitz

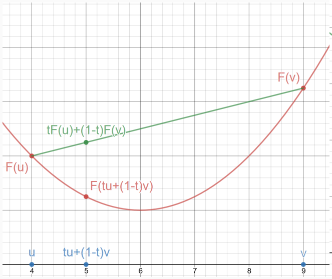


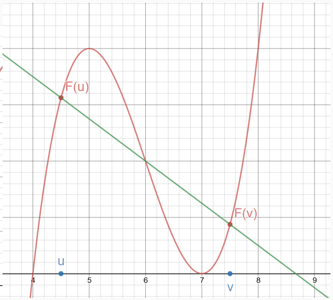
A set C in a vector space is **convex** if for any vectors $u, v \in C$ the line segment

$$tu + (1-t)v\,, \quad t \in [0,1]\,,$$

joining u to v is fully contained in C.

Convex-Smooth-Lipschitz





Similarly, let C be a convex set. A function $F : C \to \mathbb{R}$ is **convex** if for every $u, v \in C$ and $t \in [0, 1]$

$$F\big(\, tu + (1-t)v \,\big) \leq tF(u) + (1-t)F(v)\,.$$

The function on the left is convex, the function on the right is not.

Convex-Smooth-Lipschitz

The utility of convex functions in optimization come from the fact that every local minimum of a convex function is global minimum.

Furthermore, assume that $F : \mathbb{R}^d \to \mathbb{R}$ can be written $F(u) = g(\langle u, x\rangle + y)$ for some $x \in \mathbb{R}^d$, $y \in \mathbb{R}$ and $g : \mathbb{R} \to \mathbb{R}$. Then, the convexity of g implies the convexity of F.

(**Exercise**) We can show several functions are convex:

For convex functions $F_i : \mathbb{R}^d \to \mathbb{R}$, both $\max_i F_i(x)$ and $\sum_i F_i(x)$ are convex.

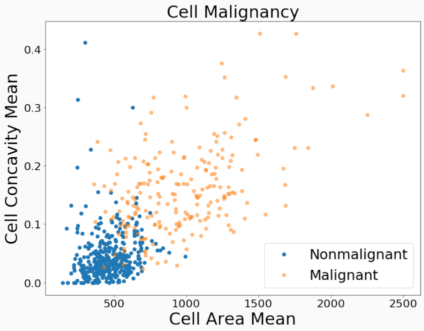
The function $g(x) = |x|$ is convex.

For $x \in \mathbb{R}^d$ and $y \in \mathbb{R}$, the function $F(u) = (\langle u, x\rangle + y)^2$ is convex.

The $RSS(\beta) = \sum_i^N (\langle \beta, x_i\rangle + y_i)^2$ is a convex function.

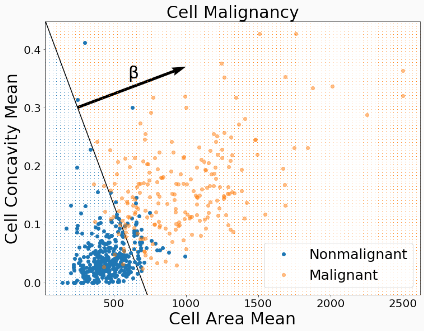
Binary Classification by a Halfspace and Linear Programming

Binary Classification



In binary classification, the target space $\mathcal{Y}$ is a categorical variable with two labels. For example, the data above (UW Breast Cancer Dataset) is attempting to classify tumor cells as malignant or nonmalignant based on the cells average area and average concavity.

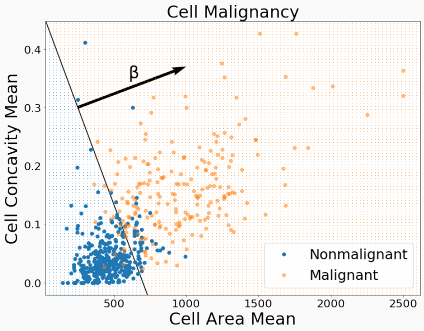
Halfspace Classifiers



The first and simplest hypothesis class to consider is the class of linear half-space classifiers. For binary variables, we can always pick $\mathcal{Y} = \{-1, +1\}$. For an input vector X, an a parameter vector β, the halfplace classifiers can be written

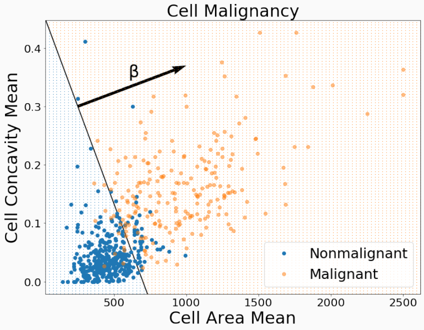
$$\hat{f} = \text{sign}(\beta_0 + X_1\beta_1 + \ldots X_p\beta_p) = \text{sign}(\beta_0 + X^T\beta)\,.$$

Halfspace Classifiers



Then, if x_i lies on the side of $\hat{f}$ that β points towards, sign$(\beta_0 + X^T\beta)$ assigns it the label $\hat{y}_i = 1$. Otherwise it is assigned the label $\hat{y}_i = -1$.

Halfspace Classifiers



In linear and logistic regression, we tried to minimize a continuous loss function, in QDA we estimated the label distributions directly. It turns out we can also directly fit the 0-1 loss directly using iterative methods, provided the data set actually can be separated by a hyperplane.

In this case, we call the labels **separable** and use **Linear Programming.** Unfortunately, implementing the ERM rule in the **nonseperable** case (as above) is known to be computationally hard.

Linear Programming

Linear programs are problems that can be expressed as maximizing a linear function, subject to linear inequalities. For $U \in \mathbb{R}^p$ a fixed input, $M \in \mathbb{R}^m$ a fixed offset vector and A a $m \times p$ matrix, we want to find

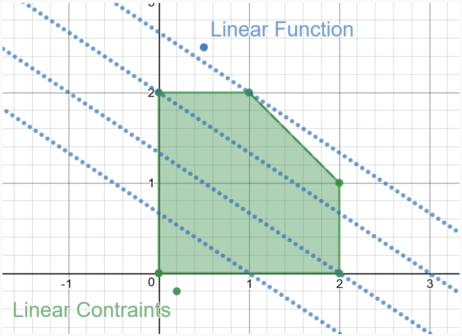
$$\max_{\beta \in \mathbb{R}^p} \langle U, \beta \rangle = \max_{\beta \in \mathbb{R}^p} U^T \beta \,,$$

subject to

$$A\beta \geq M \,.$$

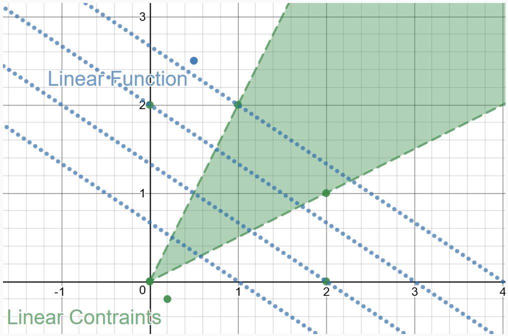
Linear programs can be solved efficiently and are implemented in a variety of packages in R, C, Matlab and Python.

Linear Programming



Linear programs can be compact problems or noncompact problems, but they are always **convex**, meaning any straight line between two points stays in their domain. Optimization over convex domains tends to be easier than over nonconvex domains since any pair of points can be interpolated between.

Linear Programming



Linear programs can be compact problems or noncompact problems, but they are always **convex**, meaning any straight line between two points stays in their domain. Optimization over convex domains tends to be easier than over nonconvex domains since any pair of points can be interpolated between. Since the function to be maximized is linear, this means the maxima are on the boundary.

Linear Programming

Idea: In the separable case, the 0-1 loss for halfspace classifiers can be minimized by solving a linear program.

Let (x_i, y_i), $i = 1, \ldots, N$, with $x_i \in \mathbb{R}^p$, $y_i \in \{\pm 1\}$. By separability, there is separating hyperplane with 0 errors on the training set. Assuming $\beta_0 = 0$ for simplicity, we are looking for $\beta \in \mathbb{R}^p$ such that

$$\operatorname{sign}(x_i^T \beta) = y_i \quad \forall i = 1, \ldots N$$

But this is equivalent to finding a β such that

$$y_i(x_i^T \beta) > 0 \quad \forall i = 1, \ldots N\,.$$

We need a strict inequality to have a linear program since the power comes from assuming the max is on the boundary.

Linear Programming

By separability, there does exist a β_* with

$$y_i(x_i^T \beta_*) > 0 \quad \forall i = 1, \ldots N \, .$$

Setting $\gamma = \min_i(x_i^T \beta_*)$, the inequality above is equivalent to

$$\frac{1}{\gamma} y_i(x_i^T \beta_*) \geq 1 \quad \forall i = 1, \ldots N \, .$$

so the problem of finding an ERM classifier is a linear program.

Letting $A_{ij} = y_i x_{ij}$ and $M = \langle 1, \ldots, 1 \rangle$, the constraint becomes $A\beta \geq M$. Since any β that satisfies this equation is an equal candidate, we set $U = 0$. This system can be solved using any available LP solvers.

Binary Classifacation via the Perception Algorithm

Perception

One of the most important algorithms in machine learning is Rosenblatts perceptron algorithm. The perceptron algorithm works iteratively to minimize the distance of misclassified points to the decision boundary.

Initially, we set all $\beta^{(j)}$ to 0 and then at each step the perception finds an example of a mislabeled instance x_i. We then generate $\beta^{(j+1)}$ by

$$\beta^{(j+1)} = \beta^{(j)} + y_i x_i \,.$$

Since

$$y_i \langle x_i, \beta^{(j+1)} \rangle = y_i \langle x_i, \beta^{(j)} + y_i x_i \rangle = y_i \langle x_i, \beta^{(j)} \rangle + ||x_i||^2 > 0 \,,$$

the new halfplane "more correctly" classifies x_i. The algorithm stops when all points are classified.
`https://www.youtube.com/watch?v=xpJHhHwR4DQ`

Perception

Theorem: Assume that (x_i, y_i) are separable. Let $R = \max_i ||x_i||$ and

$$B = \min\{||\beta|| \, : \, y_i \langle x_i, \beta \rangle \geq 1 \, \forall i\} \, .$$

Then the perceptron algorithm stops after at most $(RB)^2$ iterations.

A couple of points about the perceptron:

- When the data is separable, there are many solutions and which one is found depends on the starting value.
- The finite number of steps can be large, practically, if the gap is small the time to find it is large.
- When the data are not seperable, the algorithm does not converge, and instead falls into a cycle.

Perception

The perceptron is neither an explicit solution to a vanishing derivative, a direct gradient decent nor Newtons method. If we consider the loss function

$$\ell(\beta, \beta_0) = -\sum_{i=1}^{N} y_i(x_i^T \beta + \beta_0)\,,$$

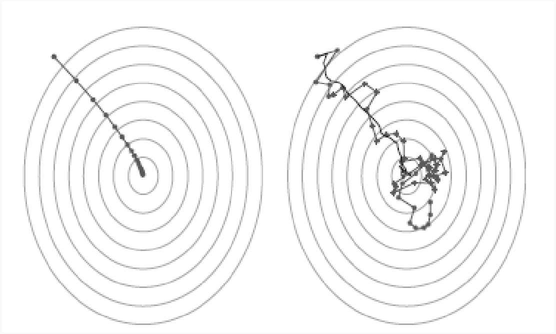
the gradient is

$$\frac{\partial \ell}{\partial \beta} = -\sum_{i=1}^{N} y_i x_i\,, \qquad \frac{\partial \ell}{\partial \beta} = -\sum_{i=1}^{N} y_i\,.$$

In gradient decent, we would update β and β_0 using the full gradient:

$$\begin{pmatrix} \beta^{(j+1)} \\ \beta_0^{(j+1)} \end{pmatrix} = \begin{pmatrix} \beta^{(j)} \\ \beta_0^{(j)} \end{pmatrix} + \rho \begin{pmatrix} \sum_{i=1}^{N} y_i x_i \\ \sum_{i=1}^{N} y_i \end{pmatrix}\,, \quad \rho > 0\,.$$

Perception



The perceptron's update picks out only a single derivative from a privileged subset:

$$\begin{pmatrix} \beta^{(j+1)} \\ \beta_0^{(j+1)} \end{pmatrix} = \begin{pmatrix} \beta^{(j)} \\ \beta_0^{(j)} \end{pmatrix} + \begin{pmatrix} y_i x_i \\ y_i \end{pmatrix} .$$

This is a form of **stochastic gradient decent**, where we only require the expected value of the update to be in the gradient direction.