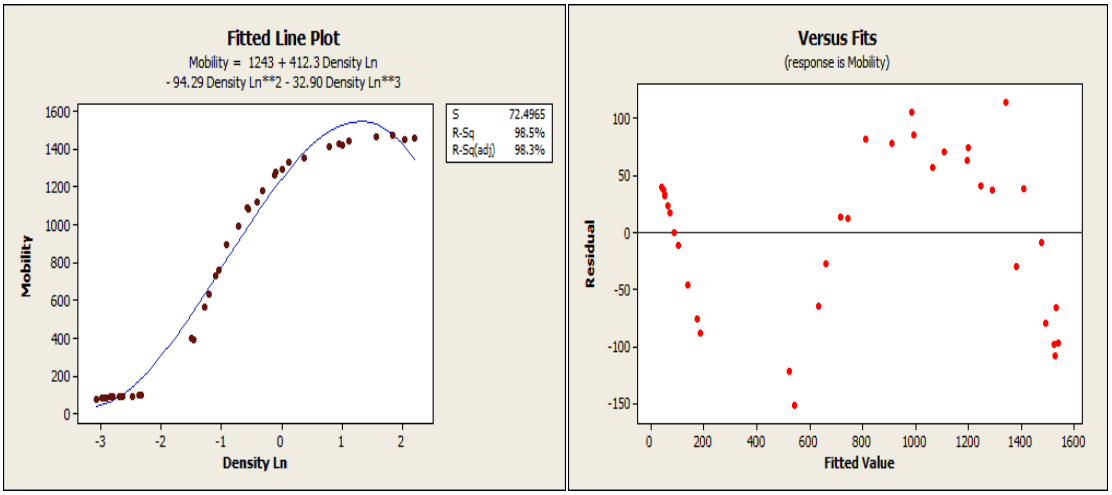
Which One is better to use – linear or Non linear regression ?

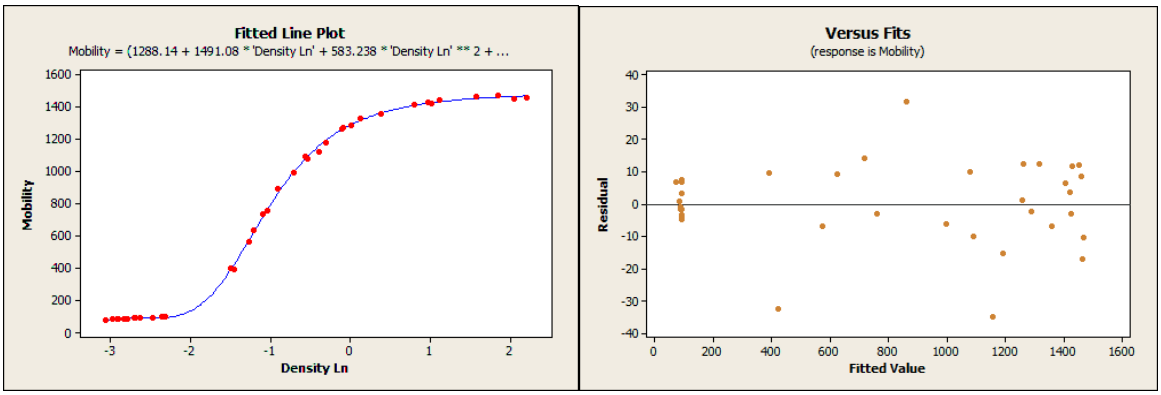
Generally speaking, you should try linear regression first. It's easier to use and easier to interpret. However, if you simply aren't able to get a good fit with linear regression, then it might be time to try nonlinear regression.

Let's look at a case where linear regression doesn't work. Often the problem is that, while linear regression can model curves, it might not be able to model the specific curve that exists in your data. The graphs below illustrate this with a linear model that contains a cubed predictor.



The fitted line plot shows that the raw data follow a nice tight function and the R-squared is 98.5%, which looks pretty good. However, look closer and the regression line systematically over and under-predicts the data at different points in the curve. When you check the residuals plots (which you *always* do, right?), you see patterns in the Residuals versus Fits plot, rather than the randomness that you want to see. This indicates a bad fit, but it's the best that linear regression can do.

Let's try it again, but using nonlinear regression. It's important to note that because nonlinear regression allows a nearly infinite number of possible functions, it can be more difficult to setup. In this case, it required considerable effort to determine the function that provided the optimal fit for the specific curve present in these data, but since my main point is to explain when you want to use nonlinear regression instead of linear, we don't need to relate all of those details here. (Just like on a cooking show, on the blog we have the ability to jump from the raw ingredients to a great outcome in the graphs below without showing all of the work in between!)



The fitted line plot shows that the regression line follows the data almost exactly -- there are no systematic deviations. It's impossible to calculate R-squared for nonlinear regression, but the S value (roughly speaking, the average absolute distance from the data points to the regression line) improves from 72.4 (linear) to just 13.7 for nonlinear regression. You want a lower S value because it means the data points are closer to the fit line. What's more, the Residual versus Fits plot shows the randomness that you want to see. It's a good fit!

Nonlinear regression can be a powerful alternative to linear regression but there are a few drawbacks. In addition to the aforementioned difficulty in setting up the analysis and the lack of R-squared, be aware that:

- The effect each predictor has on the response can be less intuitive to understand.
- P-values are impossible to calculate for the predictors.
- Confidence intervals may or may not be calculable.

Linear Vs Logistic Regression

Linear regression uses the general linear equation Y=b0+∑(biXi)+ϵY=b0+∑(biXi)+ϵ where YY is a continuous dependent variable and independent variables XiXi are *usually* continuous (but can also be binary, e.g. when the linear model is used in a t-test) or other discrete domains. ϵϵ is a term for the variance that is not explained by the model and is usually just called "error". Individual dependent values denoted by YjYj can be solved by modifying the equation a little: Yj=b0+∑(biXij)+ϵjYj=b0+∑(biXij)+ϵj

Logistic regression is another generalized linear model (GLM) procedure using the same basic formula, but instead of the continuous YY, it is regressing for the probability of a categorical outcome. In

simplest form, this means that we're considering just one outcome variable and two states of that variable- either 0 or 1.

The equation for the probability of Y=1Y=1 looks like this:

P(Y=1)=11+e−(b0+∑(biXi))P(Y=1)=11+e−(b0+∑(biXi))

Your independent variables XiXi can be continuous or binary. The regression coefficients bibi can be exponentiated to give you the change in odds of YY per change in XiXi, i.e., Odds=P(Y=1)P(Y=0)=P(Y=1)1−P(Y=1)Odds=P(Y=1)P(Y=0)=P(Y=1)1−P(Y=1) and ΔOdds=ebiΔOdds=ebi. ΔOddsΔOdds is called the odds ratio, Odds(Xi+1)Odds(Xi)Odds(Xi+1)Odds(Xi). In English, you can say that the odds of Y=1Y=1 increase by a factor of ebiebi per unit change in XiXi.

Example: If you wanted to see how body mass index predicts blood cholesterol (a continuous measure), you'd use linear regression as described at the top of my answer. If you wanted to see how BMI predicts the odds of being a diabetic (a binary diagnosis), you'd use logistic regression.

Closed Form VS GD in linear regression

Normal Equations (closed-form solution)

The closed-form solution may (should) be preferred for "smaller" datasets – if computing (a "costly") matrix inverse is not a concern. For very large datasets, or datasets where the inverse of $\mathbf{X}^T\mathbf{X}$ may not exist (the matrix is non-invertible or singular, e.g., in case of perfect multicollinearity), the GD or SGD approaches are to be preferred. The linear function (linear regression model) is defined as:

$$y = w_0 x_0 + w_1 x_1 + \ldots + w_m x_m = \sum_{j=0}^{m} = \mathbf{w}^T \mathbf{x}$$

where y is the response variable, $\boldsymbol{x}$ is an m-dimensional sample vector, and $\boldsymbol{w}$ is the weight vector (vector of coefficients). Note that w_0 represents the y-axis intercept of the model and therefore $x_0=1$. Using the closed-form solution (normal **equation**), we compute the weights of the model as follows:

$$\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T y$$

2) Gradient Descent (GD)

Using the Gradient Decent (GD) optimization algorithm, the weights are updated incrementally after each epoch (= pass over the training dataset).

The cost function $J(\cdot)$, the sum of squared errors (SSE), can be written as:

$$J(\mathbf{w}) = \frac{1}{2}\sum_i (\text{target}^{(i)} - \text{output}^{(i)})^2$$

The magnitude and direction of the weight update is computed by taking a step in the opposite direction of the cost gradient

$$\Delta w_j = -\eta \frac{\partial J}{\partial w_j},$$

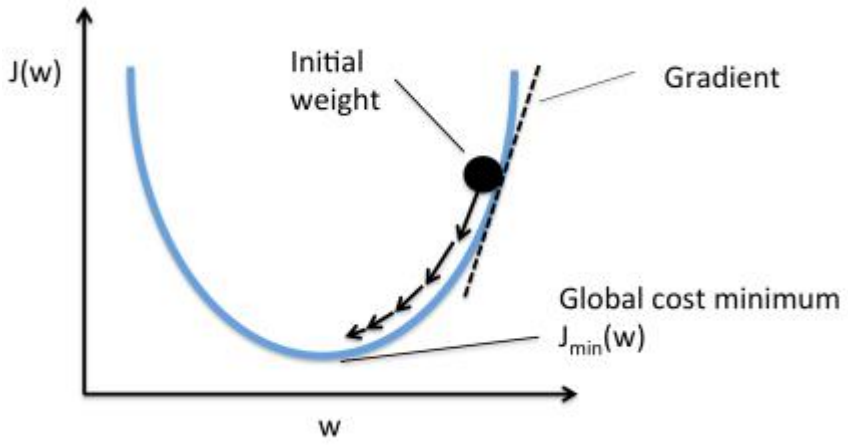
where η is the learning rate. The weights are then updated after each epoch via the following update rule:

$$\mathbf{w} := \mathbf{w} + \Delta \mathbf{w},$$

where $\Delta \mathbf{w}$ is a vector that contains the weight updates of each weight coefficient w, which are computed as follows:

$$\begin{aligned} \Delta w_j &= -\eta \frac{\partial J}{\partial w_j} \\ &= -\eta \sum_i (\text{target}^{(i)} - \text{output}^{(i)})(-x_j^{(i)}) \\ &= \eta \sum_i (\text{target}^{(i)} - \text{output}^{(i)}) x_j^{(i)}. \end{aligned}$$

Essentially, we can picture GD optimization as a hiker (the weight coefficient) who wants to climb down a mountain (cost function) into a valley (cost minimum), and each step is determined by the steepness of the slope (gradient) and the leg length of the hiker (learning rate). Considering a cost function with only a single weight coefficient, we can illustrate this concept as follows:



3) Stochastic Gradient Descent (SGD)

In GD optimization, we compute the cost gradient based on the complete training set; hence, we sometimes also call it *batch GD*. In case of very large datasets, using GD can be quite costly since we are only taking a single step for one pass over the training set – thus, the larger the training set, the slower our algorithm updates the weights and the longer it may take until it converges to the global cost minimum (note that the SSE cost function is convex).

In Stochastic Gradient Descent (SGD; sometimes also referred to as *iterative* or *on-line* GD), we **don't** accumulate the weight updates as we've seen above for GD:

- for one or more epochs:
  - for each weight j
    - $w_j := w + \Delta w_j$, where: $\Delta w_j = \eta \sum_i (\text{target}^{(i)} - \text{output}^{(i)}) x_j^{(i)}$

Instead, we update the weights after each training sample:

- for one or more epochs, or until approx. cost minimum is reached:
  - for training sample i:
    - for each weight j
      - $w_j := w + \Delta w_j$, where: $\Delta w_j = \eta(\text{target}^{(i)} - \text{output}^{(i)}) x_j^{(i)}$

Here, the term "stochastic" comes from the fact that the gradient based on a single training sample is a "stochastic approximation" of the "true" cost gradient. Due to its stochastic nature, the path towards the global cost minimum is not "direct" as in GD, but may go "zig-zag" if we are visualizing the cost surface in a 2D space. However, it has been shown that SGD almost surely converges to the global cost minimum if the cost function is convex (or pseudo-convex)[1]. Furthermore, there are different tricks to improve the GD-based learning, for example:

- An adaptive learning rate η Choosing a decrease constant d that shrinks the learning rate over time: $\eta(t+1) := \eta(t)/(1 + t \times d)$

- Momentum learning by adding a factor of the previous gradient to the weight update for faster updates: $\Delta \mathbf{w}_{t+1} := \eta \nabla J(\mathbf{w}_{t+1}) + \alpha \Delta w_t$

- For a differential real valued function of several variables, its gradient is a vector that points in the direction of maximum increase of that function, while it's negative (the vector pointing in the opposite direction) points in the direction of maximum decrease. For ML problems, the function represents the error, which we wish to minimize. Thus we should nudge our parameters towards the minimum error, by taking a small step in the direction of the negative gradient. That is

gradient descent. Were we following the gradient rather than its negative, we would be using gradient ascent.

-
- With logistic regression, we know the error function is convex, which implies a single minimum (which is nice!). So we won't get stuck in a local minimum, and so the iterative process converges to the unique global minimum.
-
- For a very large dataset, one should use stochastic gradient descent, updating one variable at a time instead of computing the gradient on the entire set, or batch gradient descent, which updates small batches of variables at each iteration.
- It is not different. Gradient ascent is just the process of maximizing, instead of minimizing, a loss function. Everything else is entirely the same. Ascent for some loss function, you could say, is like gradient descent on the negative of that loss function.

Orthogonal : Complex-valued random variables C1C1 and C2C2 are called orthogonal if they satisfy cov(C1,C2)=0cov(C1,C2)=0

Learning Rate Vs Decay Rate

The learning rate is a parameter that determines how much an updating step influences the current value of the weights. While weight decay is an additional term in the weight update rule that causes the weights to exponentially decay to zero, if no other update is scheduled.

So let's say that we have a cost or error function E(**w**)E(w) that we want to minimize. Gradient descent tells us to modify the weights **w**w in the direction of steepest descent in EE:

wi←wi−η∂E∂wi,wi←wi−η∂E∂wi,

where ηη is the learning rate, and if it's large you will have a correspondingly large modification of the weights wiwi (in general it shouldn't be too large, otherwise you'll overshoot the local minimum in your cost function).

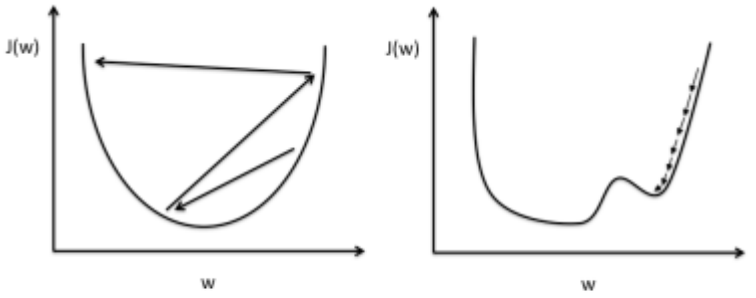
In order to effectively limit the number of free parameters in your model so as to avoid over-fitting, it is possible to regularize the cost function. An easy way to do that is by introducing a zero mean Gaussian prior over the weights, which is equivalent to changing the cost function to E~(**w**)=E(**w**)+λ2**w**2E~(w)=E(w)+λ2w2. In practice this penalizes large weights and effectively limits the freedom in your model. The regularization parameter λλ determines how you trade off the original cost EE with the large weights penalization.

Applying gradient descent to this new cost function we obtain:

wi←wi−η∂E∂wi−ηλwi.wi←wi−η∂E∂wi−ηλwi.

The new term −ηλwi−ηλwi coming from the regularization causes the weight to decay in proportion to its size.

n addition to Brent's answer, I would add that varying your learning and decay rates affects how quickly your learner converges. Take a look at this illustration:



Large learning rate: Overshooting.

Small learning rate: Many iterations until convergence and trapping in local minima.

In your logistic regression, you are attempting to minimize some loss function. To do this, you change your theta values to move in the direction towards the gradient. Your alpha value decides *how far* you move in that direction or the *step size* of your regression. As in the diagram to the left, if your alpha values are too large they will cause your regression to jump around the minimum value without actually converging on it. Therefore, we decay the alpha value by multiplying it a number $0 < x < 1$ so that it slowly decreases. If you choose this number, x, to be very small, your alpha will decay quickly to zero and you may have a problem which is shown in the right image. The step size (alpha value) is too small to get over the local minimum and your regression will converge on a suboptimal value.

Some people have reported that they have not needed to decay their learning rate (alpha) because they start with an adequately small alpha that is still large enough to avoid local minima.
You are not required to have a decay rate if you choose a good alpha, but it may help your regression converge quicker.

- $\partial 2L(\theta)\partial\theta 2 \geq 0 \Leftrightarrow$ L is convex in θ
- $\partial 2L(\theta)\partial\theta 2 > 0 \Leftrightarrow$ L is strictly convex in θ
- $\partial 2L(\theta)\partial\theta 2 \leq 0 \Leftrightarrow$ L is concave in θ
- $\partial 2L(\theta)\partial\theta 2 < 0 \Leftrightarrow$ L is strictly concave in θ

Brent trained a linear classifier on 10,000 data points and found his training error was just 2%. He tested the classifier on a test set and found the error was 10%. Explain the problem Brent is facing, places where Brent may have gone wrong during his training, and how he might address the problem to improve his model.

First of all, the problem doesn't include the number of features that exist on the data set. Usually, 10,000 records isn't very many. If there are just a handful of features, then d<<n, and a model might be able to be made. If, however, there are 10,000 features, there will likely not be enough rows to accurately train a model.

Let's assume there are only 5 features, then we can look at other causes for the error rate.

Initially, I was thinking that the error could be due to the fact that maybe the data isn't related to the response, but since the error is quite low on the training set, this is probably not the case.

Another reason why the test error could be much higher is because the training data is not an accurate representation of the distribution or makeup of the test data. For example, if the train data has 90% class A and 10% class B, but the test data is 90% class B, it will be very hard to make an accurate prediction.

So let's assume this is also not the case. What could be happening?

It's possible that Brent has overfitted the model on the training data. I think a good next step would be to look into regularization to help with overfitting. If that doesn't help, more analysis should be done on the test set or model to understand the discrepancy in the accuracy.

Describe the relationship between Bias and Variance in linear regression. How might they relate to overfitting / underfitting data? How does regularization attempt to balance the two?

The formula for a linear model can be thought of as MSE = $Bias^2$ + Variance + Error. Bias describes the systemic bias of the model. Variance describes the variability of the model estimates. One can think of Bias as how specifically the model is fit to the data, whereas variance can be descried as the "spread" of predictions.

The relationship between the two can be seen in the equation above. We can trade some Bias for some variance in order to get a lower overall MSE.

They relate to overfitting / underfitting in that having a high variance can mean the model might have underfitted, whereas having too low bias can mean the model is overfitted.

Regularization attempts to minimize the sum of the Bias^2 and Variance by making tradeoffs between the two. Different regularization methods do this in different ways, but the best methods (Ridge and Lasso) use the data to determine the regularization values.

Elastic net regularization often does better than either Ridge and Lasso alone, being an optimized combination of them.

Why is the bias zero for Linear Regression?

Bias is defined as $E[\hat{y}-\bar{y}]$ where $\hat{y}=X\theta$ (the model) and $\bar{y}=\frac{1}{n}\sum_{i=1}^{n} y_i$ is the mean of y such that $y=\bar{y}+\epsilon$ and ϵ is a random normal variable with mean $\bar{y}$ and variance $\sigma=var[\epsilon]=var[y-\bar{y}]=var[y]$.

The MSE is defined as $\sum_{i=1}^{n}(y_i-\hat{y})^2$. What model $\hat{y}$ minimizes the MSE? The one such that $\partial MSE \partial \hat{y}=0$ or

$$\partial\partial\hat{y}\sum_{i=1}^{n}(y_i-\hat{y})^2=-2\sum_{i=1}^{n}(y_i-\hat{y})=0 \text{ or}$$

$$\sum_{i=1}^{n}\hat{y}=\sum_{i=1}^{n}y_i \Rightarrow n\hat{y}=\sum_{i=1}^{n}y_i \Rightarrow \hat{y}=\frac{1}{n}\sum_{i=1}^{n}y_i=\bar{y}$$

So the model that minimizes MSE is $\hat{y}=\bar{y}$ and since $Bias=E[\hat{y}-\bar{y}]$ then Bias=0.

OR,

think the idea is that in the ideal case of $n\to\infty$ the estimation of $\hat{\theta}$ reaches the unbiased ground truth $\hat{\theta}^*$. In the case of $n<\infty$, there is an error in the estimation $e=\hat{\theta}^*-\hat{\theta}$ and if there are too many variables, this error adds up in the MSE quickly

$$MSE=\sum_{i=1}^{n}(y-\sum_{k=1}^{d}\hat{\theta}k x_{ik})^2=\sum_{i=1}^{n}(y-\sum_{k=1}^{d}(\hat{\theta}^*-e)x_{ik})^2$$

and regularization is a way to minimize this estimation error.

Describe the relative pros and cons of ridge and lasso regularization. Which one would you use (you may qualify your answer if you would use one in some cases the other in other cases).

Lasso is good at eliminating unnecessary features. So if you throw tons of features at it, it will zero out the ones that provide less of a benefit. Ridge will reduce the magnitude of unnecessary features but never eliminate them so that can lead to having too many features to deal with. Too many features lead to overfitting and excessive computation.

However Lasso uses absolute value which is non-differentiable and doesn't work with gradient descent. So Lasso can be slower to converge or may not converge.

Ridge regression will reduce the values of the weights and for two highly correlated features it will distribute the weights among them. But Lasso which pick one and eliminate the other.

Get it from Note

1. Consider the two following encoding for a categorical variable that takes three values A, B, C: (a) (1,0,0) for A (0,1,0) for B and (0,0,1) for C and (b) (1,0,0) for A (1,1,0) for B and (1,1,1) for C. Describe the pros and cons of the two approaches in the context of linear regression and which one would you use (you may qualify your answer if you would use one in some cases the other in other cases).

Dummy coding[edit]

Dummy coding is used when there is a control or comparison group in mind. One is therefore analyzing the data of one group in relation to the comparison group: *a*represents the mean of the control group and *b* is the difference between the mean of the experimental group and the mean of the control group. It is suggested that three criteria be met for specifying a suitable control group: the group should be a well-established group (e.g. should not be an "other" category), there should be a logical reason for selecting this group as a comparison (e.g. the group is anticipated to score highest on the dependent variable), and finally, the group's sample size should be substantive and not small compared to the other groups.[3]

In dummy coding, the reference group is assigned a value of 0 for each code variable, the group of interest for comparison to the reference group is assigned a value of 1 for its specified code variable, while all other groups are assigned 0 for that particular code variable.[2]