**Ridge and Lasso Regression**

Going to do a prelude introduction,

**Introduction**

So say we have some data yi, and multiple independent variables, say x1,2, we’d like to model it. We could do something like,



where,



And this is a linear regression because f is linearly related to the regression coefficients. And we could use sklearn, or whatever, to find a1,2, b1,2, and c. We’d just let our independent data be X = [[x1i], [x1i2], [x2i], [x2i2]], and our dependent data be y = [yi], and then do the fit. If we don’t already have an idea for the powers of x1,2 (if any) that are involved, then we run into the problem. How we know that we shouldn’t be regressing an equation like,



Or for that matter, an equation like,



The problem would come when we try to test our equation against the testing data. So say we train our data on the underfit f. I imagine we won’t get a good fit, obviously, since we’ll be missing the critical x1,22 dependence term. And when we test the fit against our testing data, we’ll again not get a good fit because we’re missing that critical dependence. On the other hand, if we train the overfit f on our training data, with all the extra parameters in there, we’ll probably have a super good fit, because we will have fit the noise. But when we test against the testing data, we’ll get a horrible result. This problem would be exacerbated if our data set is small (so that fluctuations in data don’t average out), and/or if our data is prone to have outliers (that we’re still commited to fitting). The outlier problem is severe because different training sets can have quite different outliers, and so give quite different fitting coefficients, especially for high power terms (like x2 as opposed to x), and so give really bad testing performance. I think another way the overfitting problem would manifest is that the true c1,2 fitting parameters would be small. And the c1,2 parameters we find from the training data would fluctuate between positive and negative values as we vary the training data set using cross-validation.

This problem would also arise for categorical data I think. It might be that the categories don’t matter. For instance, suppose we had 500 chipmunks variously named Alvin, Theodore, and Simon. If we were testing a drug on them, we could split them up by name, as well as other relevant features like age, gender, weight, etc. Any regression model we ran would fit the date better with the categorical name data in it, because having the names in there would give us three linear regressions we could fit, instead of just one overall linear regression. But most likely, when we split the data different ways, the coefficients for categorical terms would likely fluctuate a lot, and, I presume, be rather small, and similar to each other. And so doing cross validated train\_test\_split would not give us great results because of these fluctuations. We’d expect that eliminating the name category features altogether would give us more stable and accurate cross validated train\_test\_split results.

So we’d like a way to say to the fitting algorithm, ‘don’t use the higher power parameters to fit too closely to the data’, or, ‘don’t try too hard to fit the outliers, ‘cause the testing set’s outliers will likely be different’. And ultimately, ‘don’t use the higher power parameters to fit too closely to any new data, ‘cause it’s outliers will be a bit different too’. And even more ultimately, don’t use the higher powers, or superfluous categorical features at all, if they’re not necessary, and they’re just fitting noise, rather than the underlying pattern. The mathematical way to do this is by adding a penalty for the higher power terms (or categorical terms). So for instance, if we’re fitting the polynomial:



to the data. Instead of minimizing,



we’d minimize, generically, something like:



where p is some power, and λ is a regularization parameter. Setting p = 1 is called Lasso regression. Setting p = 2 is called Ridge regression. There’s no fundamental reason for prefering one power to the other, I don’t think. Anyway, so the extra term would impose a cost on having a large coefficient for the higher power terms. And this ought to discourage overfitting. And accordingly, it will encourage c1,2 to be closer to zero, than regular regression will. So note if we don’t expect c1,2 to be ‘small’, then we may not want to use Ridge/Lasso regression. Or we might just find that the best hyperparameter is λ close to 0.

So we’d run the regression algorithm on the training data using the new SSEf,λ, and this would return a series of coefficients: a1,2, b1,2, c1,2, d. And then we would cross validate the regression curve on the testing data. But note that when running the regression curve on the testing data, we’d evaluate its fit using the old SSEf. This is because SSEf is the true measure of goodness of fit. And we were only using SSEf,λ to try to return a regression curve that didn’t fit the training data too closely, so that it would fit the testing data better. Different values of λ would return different regression curves, and therefore different testing data fits. So we will, unfortunately, need to try many different values of λ, until we found the one, λ = λ\*, that returns the best regression curve, as measured by the average fitting score overall all validations. We should also find that the variance in our fitting parameter c1,2 (and maybe a1,2, b1,2, d as well) across the different validations goes down as we approach the optimum λ = λ\*. I would expect the same for the variance of R2. The min variance won’t be precisely at λ = λ\* I presume, but should be in the ball park.

A graph of a function

Description automatically generated

This is good because a good model should return approximately the same regression coefficients for the different validation folds. If it doesn’t, then there is no reason to suspect that with the different data we might deploy our model on, we wouldn’t get markedly different regression coefficients there too. This would make our model unreliable and inaccurate, as there should be just one underlying behavior for all the data.

The problem of overfitting or underfitting a regression is analogous to the overfitting/underfitting problem with decision trees. When we fit all the way a decision tree to the training data, that is like fitting a curve with a whole bunch of parameters. And we obviously get a really good fit. But then when we test the DT against the testing data, we get poor results. So we have to prune the tree. And a mathematically well defined way to prune the tree is to use cost-complexity analysis, where we put the pruned trees in a one-to-one correspondence with the cost complexity parameter, α. When α is small, we have very little pruning, and when it’s large, a lot of pruning. And there will be a sweet spot, α = α\*, where the pruning is just right, and we get the best performance against the testing data.

**Back to our regularly scheduled programming**

So going back to normal univariate regression, there are two powers of interest p = 1, 2, which go by the names Lasso and Ridge Regression.



Ridge Regression is also called L2 regression. Lasso Regression is also called L1 regression. The 1 and 2 refer to the power of the slope in the SSE. So for instance regular linear regression,

Chart, scatter chart

Description automatically generated

fits the data with a line,



and the Ridge and Lasso Regression would minimize,



Multivariable regression would fit the data with:



and the Ridge and Lasso Regression would minimize,



Categorical regression looks like,

Chart, scatter chart, box and whisker chart

Description automatically generated

and would try to fit the data with, say,



and the Ridge, Lasso regressions would minimize,



Finally, categorical, slopey regression looks like,

Chart, scatter chart

Description automatically generated

and would try to fit the data with something like,



using Ridge, Lasso Regression which would minimize,



The higher λ is, the more the slope is penalized, and consequently the lower will be the fit’s slope. This has the virtue of supressing large fluctuations in the slopes that comes from having only a few data points in the training sets. Turns out that while Ridge regression will likely never return a fit where slope = 0, Lasso regresion will. I guess this comes about due to the Lasso Regression formula’s non-analytic dependence on m via the |m|. So turns out that when λ > λcritical, it will return fits with slope = 0. Will note that Lasso Regression penalizes small slopes more than Ridge Regression does, so it seems plausible that this could happen. Below I’m stealing a plot from Stat Quest of SSEL and SSER vs slope for a linear regression. Can see that the values of m (slope) that minimizes the error gets smaller and smaller as λ increases. But it only asymptotically approaches zero for SSER, while it actually equals 0 past a λcritical ~ 40 for SSEL. Interesting.

A picture containing chart

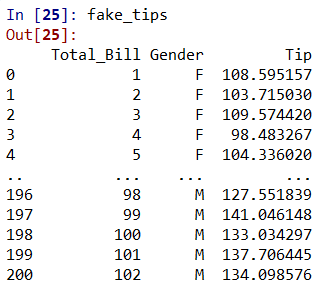
Description automatically generated A picture containing histogram

Description automatically generated

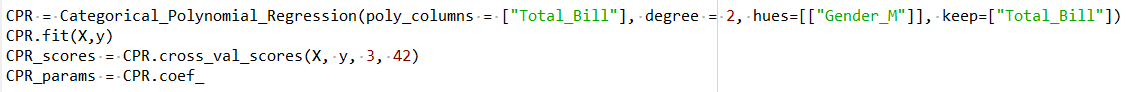
So the features corresponding to the coefficients whose slopes are zero can be said to deselected from the regression model, as they effectively don’t appear in it. And so again, only those features whose coefficients are non-zero would be effectively ‘kept’. So this can tell you what the important features of the dataset are. For instance, going back to the 500 chipmunks example of names Alvin, Simon, Theodore, we’d find that the coefficients multiplying the name category variable would go to zero at some point. And we should find that having eliminated this categorgy gives us a better overall fit, better cross-validated train\_test\_split. This is an advantage of the Lasso Regression approach. It gives you a way to ascertain whether the feature should even be in the model at all.

**Categorical Polynomial (Lasso) Regression**

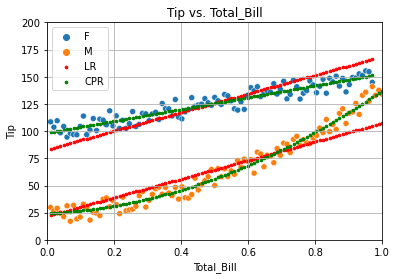
Let’s recall the example I did of the case of two tips separated by gender (see Categorical Regression II file). I made the female tip rate linear (Tip = 100 + 0.5\*Bill), and the male tip rate quadratic (Tip = 25 + 0.1\*Bill + 0.01\*Bill2),



and then fitted a Categorical\_Polynomial\_Regression model to it, which adds the necessary columns to do polynomial regression, and then adds the necessary columns on top of that to do categorical regression. Setting the polynomial degree to 2,

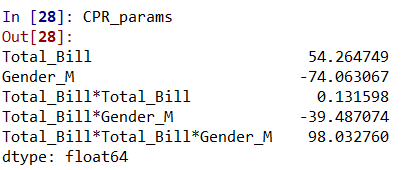


I get,

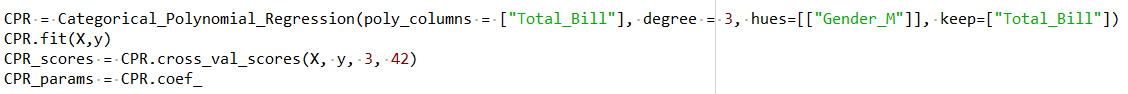


The red double lines are what a normal LinearRegression would give. The green lines are Categorical\_Polynomial\_Regression curve. The coefficients are:

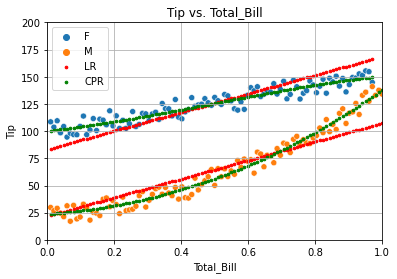




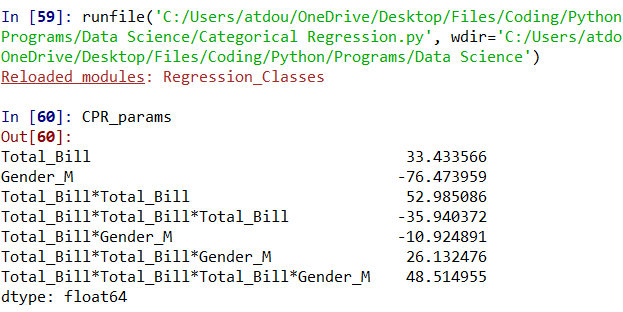
And this worked out pretty well. What if we had tried a cubic polynomial? Then we’d have said,



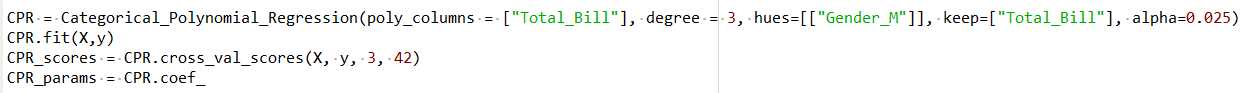
and the fit would look like,



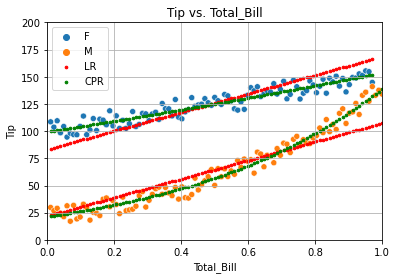
which is almost indistinguishable from the quadratic case, except we can see a little more curviness in the green line. And we’d obtain the following coefficients.



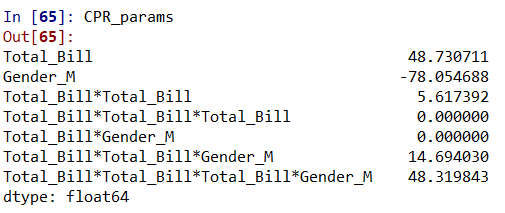
Can see the green line now has significant quadratic and cubic coefficients, which it shouldn’t have. But we can mitigate this with Lasso regression. If we fit λ = 0.025,



we get:



Can see the green line has been straightened out a little bit. And this is corroborated when we print out the coefficients,



Now the Female curve has no cubic part. It *does* have a quadratic part, which it shouldn’t have. But this is much reduced from what it was. I wasn’t able to entirely eliminate the quadratic part. Problem is, as λ increases, the linear slope decreases as well, which further necesitates a quadratic part of the curve to ‘catch up’ to the rest of the values. Also unfortunately, the Male part of the regression curve picked up a cubic part. Couldn’t get this to go away either. But luckily, the cubic part has little effect on the data for the most part as cubing a number between 0 and 1 generally makes it pretty small.