**Categorical Regression II**

So in the Hypothesis test file, we saw how to calculate a p-value for comparing the means of two different groups. And in the Regression file, we saw how to calculate a p-value for a regression curve. Now we’re going to kind of combine the two concepts together. For instance, we’d like to be able take a regression curve for one set of data, say group A, and a regression curve for another set of data, say group B, and calculate a p-value for whether the regressions are meaningfully different. Or another way to say it, is there any predictive power in classifying data into one of those two groups, or should we just treat them as all the same group? So this would be like doing a regression fit over independent variables, some of which are continuous, and some of which are discreet (the group index).

**Linear Regression**

Now we’ll look at comparing linear regressions between groups using a design matrix.

Chart, scatter chart

Description automatically generated

We could write categorical regression equation like this:



where f is modeling size, s, and mA,B = best fit slope for group A,B data, bA,B = best fit intercept for group A,B data, and A = 1,0, B = 0,1 for whether it’s group A,B, just like in previous file. Another, more common I think, way to write this is,



Not sure why this is preferred – maybe something about a colinearity trap thing. The blue guys are the fit parameters. w and B are the data. Accounting for random variations about this fit, we could write this in matrix form,



where,



and the ε’s are normally distributed variables with mean 0 and std σ. And X would be our **design matrix**. If we minimize the total square error of our fit,



then we should get the usual expressions for mA,B and bA,B found in the Regression file. But in general, as we saw in the multiple regression file, we will find the fit parameters are given by:



**Point Estimators for vi**

As before, we’ll recognize that **v** is really a random variable **V**, given by:



where **Y** is the random variable at the top of the page. And in the multiple regression file, we found as well that:



And finally, we found a good point estimator for σ2 was:



where n is the number of data points, and f is the number of d.o.f. of our model. And, so,



So if we know know σ2 for sure, then we can say the ZVi guy,



follows a unit normal distribution. And if we don’t know it for sure (not sure how we would), then we can say,



follows a Student’s T distribution with ν = n-f d.o.f. If ν > 30 or so, then this is just a normal distribution for all intents.

**Hypothesis Testing of Parameters**

If we are running an experiment, and get some particular values, <**V**i> = vi, for our linear regression, we may wish to see whether our results invalidate someone else’s linear regression values, or the standard/accepted linear regression values, vi\*. We can do a hypothesis test for this. We know how Vi are distributed, according to our experiments, and so I guess we could form the Z-statistics,



and then calculate the p-values, the probabilities that values at least as extreme as vi\* occur. So we’d calculate,



p(z) would be the aforementioned Student’s T distribution of course.

**Confidence Intervals for Parameters**

We can also calculate confidence intervals for the statistics Vi. Just like we’ve done with other statistics, we’d say, at the 1-α confidence level,



And the zα value would be for a Student’s T distribution.

**Goodness of Fit: R2 value**

We can define a goodness of fit just as before.



where fi are the regression curve data points, which one could say is a random variable since it depends in known way (if have formula) on the random variable Yi, the data points we’re trying to fit. SSEf = sum of squares about f, and SSEm = sum of squares about mean. This is actually a general formula that applies to all curve fits. Apropos the fraction, the numerator is the ‘variation around the fit, or f’, and the denominator is the ‘variation around the mean’). Can see R2 = 1 if there is no variation around the fit, i.e., if the curve/fit fits the data exactly. On the other hand, if the curve fits the data no better than the mean, then we get R2 = 0. If you have an R2 = 0.75, then you can say that 75% of the variation of the data is explained/predicted by the regression curve f. As usual, we can interpret R2 as:



And last,

**Hypothesis Testing Different Regression Models**

There are lots of different models we could test now. Presently, we are splitting our data into groups: A, B, C, …, G, and doing a separate regression on each one. But we could treat them all in aggregate, as just one group and do a single regression. Or we could keep them all in their groups and just fit a bunch of means to the data (basically the T-test in the previous file), or we could split into a different set of groups and do a different regression curve, etc. So we want a way to compare the models to see which is better. We’ll use a Hypothesis test framework for this. So say we have a model with f0 d.o.f. **v**(f0) = (v0, v1, …, vf) and another model with f > f0 d.o.f. **v**(f) = (v0, v1, v2, v3, …, vf). Now let’s define a Null Hypothesis.

H0 = assumption that the data is described by model **Y** = X**v**(f0) + ε, which has f0 degrees of freedom.

And let SSEf0 be the sum of the square errors for this model.Then let’s compare to another model with f (> f0) degrees of freedom, fitting **Y** = X**v**(f) + ε. And let SSEf be its sum of squared errors. We would anticipate this to be smaller of course, i.e., SSEf0 > SSEf. The alternative hypothesis would be:

HA = assumption that at least one of the extra f – f0 parameters in the new model Y = Xv(f) + ε is non-zero.

Turns out SSEf follows a known probability distribution, presuming the truth of H­­0. We can form a test statistic,



and n = number of data points. We might recall SSEf/(n-f) is just the point estimator for σ2 in the f-model. Might interpret Z\* as:



Turns out this follows an F-distribution (B is the β function).



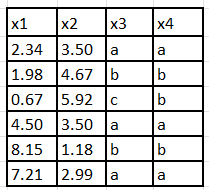
which is the probability density of getting an Z-value of x, given the null hypothesis is true. So we can calculate a p-value,



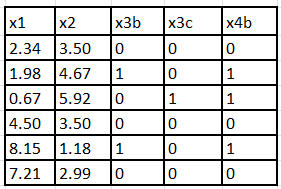
which would be the probability that we’d get an Z-statistic Z\* or higher, out of the new model, if the Null hypothesis were true. So if the f model has true explanatory power, then we should find Z\* >> 0 and the p-value should be small (less than 0.05 at the 95% significance level). But note for instance that if f = n, i.e., if the number of fit parameters equals the number of data points (allowing an exact fit of f to our data points), then Z\* = 0, and so our p-value would be 1 I think. And this would mean that our extra parameters are meaningless.

**Categorical (Polynomial) Regression**

Just noticed something. Say we have some data X, comprised of four columns: x1,2 (numerical columns), and x3,4 (categorical columns with values equal to a,b,c, or a,b).



When we one-hot-encode this, we’d get:



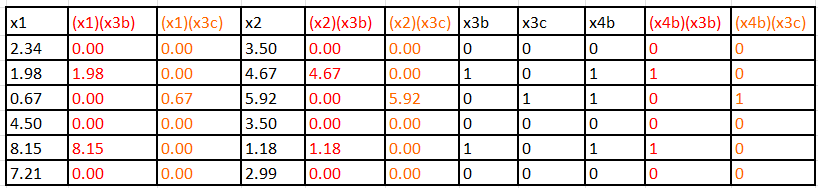
Will write this table as *df = [x1, x2, x3b, x3c, x4b]*, for short. When we do linear regression with categorical variables in sklearn, we get an equation like this:



which we’ll note is a little less general than the formula we’ve been exploring above. That’s because when x3b = 0,1 and x3c = 0,1, x4b = 0,1 respectively, we have:



And so this equation only changes *intercept* when x3b, x3c, x4b are ‘turned on’. But the slopes with respect to the numerical columns remains the same. Having a different slope per category seems sensible (and our formalism above allowed for it). Consider housing data for instance. Different categories could represent different neighborhoods, and maybe in expensive neighborhoods they use more expensive flooring, and so the price per square foot would be higher in such neighborhoods. Or if you’re looking at tips, people might tip at a higher rate at certain expensive restaurants, or on certain days, than others. How could we implement this? There is a problem though. To have a slope and intercept for every possible categorical variable combination, we’d need a lot of variables. Say we have two numerical variables, x1,2 and 3 categorical variables, x3,4,5, each with 4 options. Then the number of categorical variables combinations is 43 = 64. And then we need for each of these, a slope for x1,2 and an intercept, which is 3 parameters. So we need a total of 3∙64 = 192 free parameters. In general, if have n numerical variables, and categorical variables c1, c2, c3 etc., with number of outcomes o1, o­2, o3 each (for example), then we’d need (n+1)∙o1o2o3… parameters. This could be huge. And we probably wouldn’t even have enough data points for each of the o1o2o3 categorical combinations to get a decent non-volatile regression. Or even a *single* data point for every categorical combination. So out of all the categorical data we have, we might want to open up only a few categories to the possibility of different slopes. Okay, so how do we do this in sklearn? Well, we can multiply the numerical data by our one-hot-encoded columns I think. For instance, suppose we wanted to allow for different slopes according to the value of x3. Then we’d make copies of the x1, x2, x4 columns, multiply them by the x3b, and x3c columns, and add them to our datatable. Helps to read the columns in order of black (original df), red (multiplying black by x3b), orange (multiplying black by x3c).



Can write this is as:

*df = [x1, x2, x3b, x3c, x4b] + [x1, x2, x4b](x3b, x3c)*.

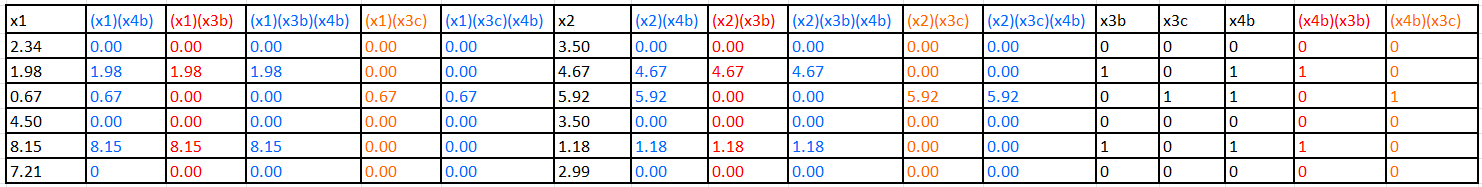
This has a total of 11 features, which sklearn would model with 12 parameters (including the intercept). This matches our formula (n+1)o3 = (3+1)(3) = 12, where 3 is the number of ‘numerical columns’ (including x4b in this because we’re not expanding it), and 3 is the number of outcomes of feature x3. If we fit this using sklearn, it would do a regression of form:



Now when x3 = a, b, c respectively, we have:



So now we get a new intercept *and* slope when we change categorical values. A nice thing about this setup is that we can do Lasso regression on our table. And if there is no support for treating the categories as following separate slopes, then we should find the coefficients of the (x1∙x3b), (x1∙x3c), (x2∙x3b), (x2∙x3c) columns should go to zero. And if there’s no support for splitting x4b into subgroups according to x3b,3c, then we should find the coefficients of the column x4bx3b and x4bx3c to go to zero as well. And if that’s the case, we can just eliminate these columns, and redo normal linear regression. What if we wanted to split all different slopes and intercepts according to all possible combinations of categories of x3 and x4? Then we’d mentally take out the x3b,3c columns (because we already have products of them with x4b), multiply everything by x4b, and add it to the data table. I think it’d look like this. Helps to read the columns in order of black (original df), red (multiplying black by x3b), orange (multiplying black by x3c), blue (multiplying black, red, orange by x4b).



Can write this as:

*df = [x1, x2, x3b, x3c, x4b] + [x1, x2, x4b](x3b, x3c) + {[x1, x2] + [x1, x2](x3b, x3c)}(x4b).*

This also matches our formula, we should have (n+1)o3o4 = (2+1)(3)(2) = 18 parameters, where 2 is number of numerical columns, 3 the options of x3 and 2 the options of x4. And indeed we have 17 columns, which, when include the intercept, comes to 18 free parameters. As we can see, the number of free parameters grows quickly. In our simple example, we already have a lot more parameters than datapoints. So the general rubric is:

1. One-hot-encode the categorical features of the dataframe, and put the features that you want to expand in the expanding\_features\_list (efl). Might have something like this, where we have two numerical columns n1, n2, and where we expand categorical columns c1 (three outcomes), c2 (two outcomes), c3 (three outcomes), but leave c4 (two outcomes) alone.



1. Now take first categorical feature tuple, (c1b, c1c) out of efl1 and out of df1. Then multiply it with all the other columns in df1, and add the generated new columns to df1. Notation works out best if multiply on the right.



This checks out as we should have (n+1)o1 = (6+1)(3) = 21 free parameters. And in our df2 we

have 20 parameters, + 1 for the intercept when do the regression.

1. Then repeat for feature (c2b). We would take (c2b) out of the efl2, and remove all columns in df2 that begin with (c1b, c1c) or (c2b), multiply on right by (c2b), and add the new generated columns to df2.



This works out as we should have (n+1)o1o2 = (5+1)(3)(2) = 36 free parameters. And in our df3

we have 35 free parameters + 1 intercept = 36. So we’re good.

1. And last repeat for (c3b, c3c). We’d remove (c3b, c3c) from efl3, and then remove all columns that begin with (c1b, c1c), (c2c), or (c3b, c3c) from df3, multiply on the right by (c3b­, c3c), and add the new generated columns to the df3.



And this works out as we should have (n+1)o1o2o3 = (3+1)(3)(2)(3) = 72 free parameters. And in

our df4 we have 71 free parameters + 1 intercept = 72 free parameters.

So I *think* this is the procedure. Revisiting our df4,



how would we interpret the column’s coefficients that we’d get for the linear regression? If we stop at the black row, then the coefficients are just the slopes for the respective variables of course. We have coefficient of n1 is slope of n1 (regardless of c’s), and coefficient of n2 is slope of n2 (regardless of c’s), and coefficients of c’s are the slopes of c’s, etc.

If we stop at black and red row (just expanding c1), then coefficients of the black rows are implicitly those with c1 retricted to c1a. Except, c1b and c1c, which are obviously not implicitly restricted to c1a, as they would then we be zero due to incompatibility. If look at example above, can see that the c1b, c1c coefficients end up being changes to the intercepts when c1b and c1c are turned on.

If we stop at black, red, and blue rows, then coefficients are broken down by c1 and c2. I guess the black and red rows are now implicitly restricted to c2a. And the blue rows contain the c2b additional contributions.

Similarly for the purple row.

One concern is that this procedure has ‘memory’. If we expand column c1, and then later decide to expand column c2, we would have to know that we had already expanded c1. I feel like we shouldn’t need to know this beforehand.

**Training and testing on datasets with few instances per category**

Sometimes we have datasets with a category, like house location, which can take on multiple values, even hundreds. And there may be hundreds of rows corresponding to one location, and only one row corresponding to another. When we split the dataset for training and testing, our training dataset may have instances that our testing dataset does not have, and vice versa. For instance, say, we split the dataset below into training (grey-blue) and testing (white).

|  |  |  |  |
| --- | --- | --- | --- |
| **Dosage** | **Age** | **Sex** | **Effectiveness** |
| 10 | 25 | Female | 98 |
| 20 | 73 | Male | 14 |
| 35 | 54 | Female | 6 |
| 5 | 12 | Male | 44 |
| 25 | 44 | Female | 73 |

If we one-hot-encode our Sex category (Female = 0, Male = 1), and expand Dosage by Sex, then we’d get:

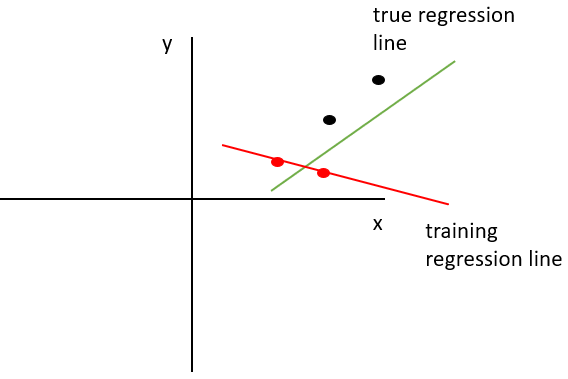
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Dosage** | **Dosage∙Sex** | **Age** | **Sex** | **Effectiveness** |
| 10 | 0 | 25 | 0 | 98 |
| 20 | 20 | 73 | 1 | 14 |
| 35 | 0 | 54 | 0 | 6 |
| 5 | 5 | 12 | 1 | 44 |
| 25 | 0 | 44 | 0 | 73 |

Now our training regression equation would be something like,

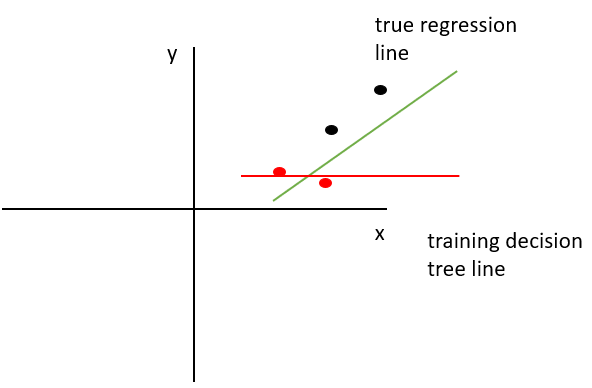


So Dosage∙Sex and Sex just wouldn’t matter to the training. And the values of mDS and mS wouldn’t be updated from their initial values, whatever those are. And since the initial values are *some* number (zero in the case of Lasso regression), our training dataset regression curve would still be able to make predictions on the testing data. They just probably wouldn’t be very good predictions.

A related problem with Categorical Regression is that even if it doesn’t reduce the number of rows per value to zero, it will still be quite small. And so the training algorithm will create a regression curve off of data that might be quite different from the testing data, resulting it really bad fits. Can see that if our training data is the red two points, then it will be a really bad predictor of the testing data black points.



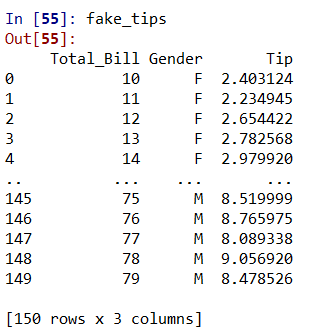
Linear (Quadratic, Polynomial, etc.) regressions seem really bad with this compared to, say, Decision Trees, which only estimate mean values, not slopes too. Can see the error here is not as bad.



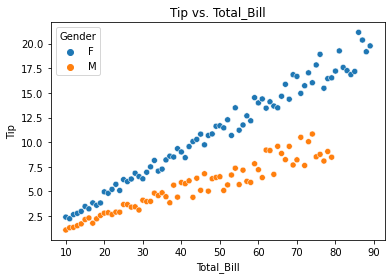
This is one reason why Regularized Regression (Lasso, Ridge) is useful.

**Exploring the Model and Hyperparameters**

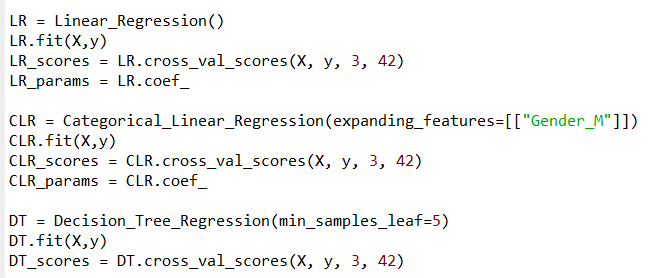
So just to illustrate, I made a fake\_tips dataset,



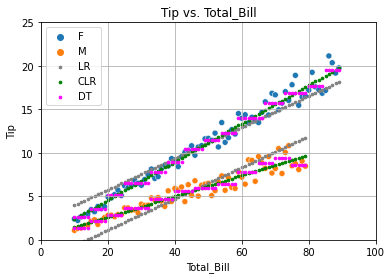
which has women tipping at around 20%, and men at around 10%. Plotting the data Tip vs. Total\_Bill, we get:



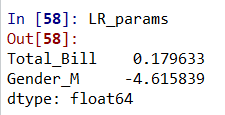
Then I ran a linear regression program, a categorical regression program as described above, separating things by gender, and finally a decision tree program, and trained it on the data,



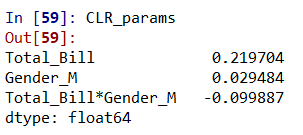
and plotted the regression curves.



Can see that the linear regression program can’t change slope based on category, only intercept, which is a liability here. The categorical regression program can change slope, and fits the data quite well. The decision tree regression can of course respond to different categories, and so it does a better job than the linear regression. But as always, the downside of the decision tree is its choppiness, so it doesn’t do as well as the categorical linear regression. When we print the LR and CLR coefficients, we get:

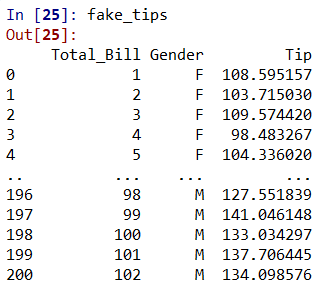


and,

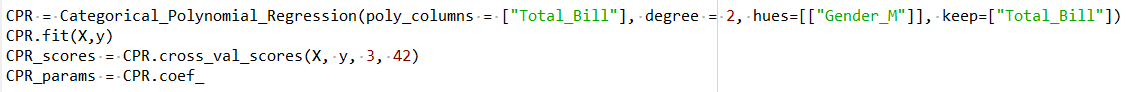


As can see, there is only one slope for LR (coefficient of Total\_Bill), and its roughly an average of the 10% and 20% tip rates of the two genders. But do we have two slopes for CLR. We have 22% for the default Female gender, and then a -10% correction for switching to the Male gender, which changes the tip rate to 22% - 10% = 12% for Male. (I didn’t include the Regression intercepts in these printouts).

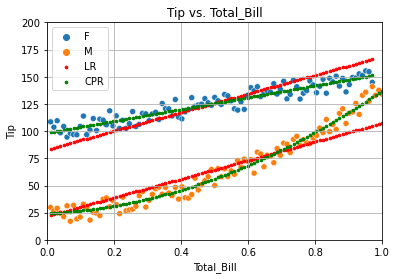
I did another example where I made the female tip rate linear (Tip = 100 + 0.5\*Bill), and the male tip rate quadratic (Tip = 25 + 0.1\*Bill + 0.01\*Bill2),



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,

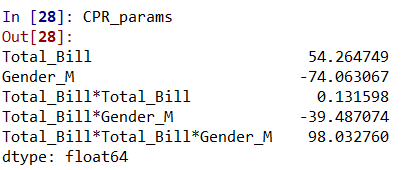


I get,



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





And this works out pretty well. Accounting for having normalized the data by dividing Total\_Bill by 100, and Total\_Bill\*Total\_Bill by 10000), the Female (default) intercept is 98 vs. 100. The slope is 0.54 vs. 0.50. And it also gives a Female quadratic coefficieint of 1.3×10-5 vs. 0. For the Male gender, we have the intercept is 98 – 74 = 24 vs. 25. The slope is 0.54 – 0.39 = 0.15 vs. 0.10, and the quadratic coefficient is 98×10-4 = 0.0098 vs. 0.01. So these results are pretty good! BTW, we can also reproduce the CLR model using a neural network – this just requires a bit of work to fine tune, and the interpretation of the results isn’t as clear as it is here with the CLR.

**Appendix**

When doing regression on a data set with dummy variables that have been translated to vector thing,

Table

Description automatically generated Table

Description automatically generated

you’re supposed to drop one of the columns when doing a linear regression, to avoid a ‘colinearity’ trap thing. If you drop monroe township, then you’d refer to the towns as monroe township = (0,0), west windsor = (1,0), robbinsville = (0,1)