**Multivariable Logistic Classification**

Seems that now we are just generalizing our curve to include multiple independent variables **x**i = (x1i, x2i, x3i, etc.).

Chart, line chart

Description automatically generated

And our curve would generalize to, in d = 3 for instance:



The best fit curve can be ascertained by adjusting the parameters to maximize the cumulative likelihood:



or minimizing the log-loss,



And like before,

Looks like we can take LLf to be analogous to the SSEf from linear models. It will be zero when there is perfect fit, and ∞ for really bad fit.

Note logistic regression is a *linear* classifier because the decision boundary is a hyperplane. This is because when z = m1x1 + m2x2 + m3x3 + b > 0, then f > 0.5, and the decision is positive (usually we set the threshold to p = 0.5, but could be otherwise). But when z < 0, then f < 0.5, and the decision is negative. So z = m1x1 + m2x2 + m3x3 +b = 0 is our decision boundary. And this is the formula for a hyperplane. So logistic regression ought to be appropriate if we can come close to physically separating the yes’s and no’s in the d-dimensional phase space (d = # independent x variables), like this:

A diagram of a function

Description automatically generated

As suggested in the examples in the previous file, if we *can* linearly separate all the no’s and yes’s into separate volumes of feature space, then we ought to be able to perfectly fit the data. In that case the data is linearly separable. In this cae LLf = 0. On the other hand, the ‘worst’ fit scenario should still be when the yes’s and no’s are completely inseparable. In this case our logistic regression curve becomes a simple straight line (or rather, hyperplane), which we obtain by setting all the m’s = 0. Then we have f(x) = 1/(1+e-(0·x\_1 + 0·x\_2 + 0·x\_3 + b) = p. Should find again that p = n1/(n0 + n1) where n0,1 = total number of no’s, yes’s. And LLf will be in this case, like before,



Okay. This is just some large positive number. Is this the entropy of mixing? Anyway…

**Goodness of Fit: R2 value**

R2 is a measure of the goodness of fit, of the regression curve. There’s apparently no consensus on a definition of R2. But one analogous to the linear models expression is this. We define



where LLf is the log-loss of the logistic regression curve, and LLm is the log-loss of the curve that just fits the ‘mean’ of the data, i.e., it is f(**x**) = p = n1/(n0 + n1), where n1 is the total number of yi = 1’s and n0 = the total number of yi = 0’s. As we argued in the example above, LLm forms the upper bound of the log-loss for any logistic function. So we should always have LLf < LLm. And so 0 < R2 < 1.

**Hypothesis Testing**

Now that we are allowing multiple fitting parameters, it’s useful to know how to tell if the fitting parameters are predictive – if they are fitting the behavior, or just the noise. In analogy with the multivariable linear regression stuff, let’s say we have a model with f d.o.f./fitting parameters, which is taken as the standard model so-to-speak. Then let our Null Hypothesis be as follows:

H0 = assumption that the data is described by model f(**x**) which has f0 degrees of freedom.

And let LLf0 be the log-loss for this model.Then let’s compare to another model with f > f0 degrees of freedom. And let LLf be its log-loss. We would anticipate this to be smaller of course, i.e., LLf0 > LLf. The alternative hypothesis would be:

HA = assumption that at least one of the extra f – f0 parameters in the new model is non-zero.

Given the Null Hypothesis, turns out LLf follows a known probability distribution. Well we can form a test statistic,



And this follows a χ2 distribution with f – f0 d.o.f..



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).

**Exploring the Model and Hyperparameters**

So here’s logistic regression run on a linear boundary with 0% and 10% outliers.

A diagram of a log regress

Description automatically generated A diagram of a log regress

Description automatically generated

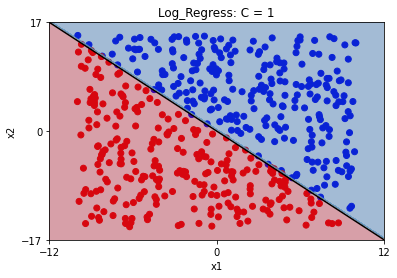
Our results are pretty similar to the linear SVC. One could alter the performance of the model by adjusting the probability cutoff, pcut, that it uses to classify as 0 or 1. This cutoff is by default pcut = 0.5. And while pcut is not a hyperparameter per se´, we could have the model output the probabilities, pi, of all the points, and then tell it to classify the points as +1 if pi > pcut and 0 if pi < pcut. And we could make pcut = 0.55 or something. I’m thinking this would have the same effect as altetering the parameter b in f(x)? So really, changing pcut, leaving b fixed, is probably the same as leaving pcut fixed (at say 0.5) and changing b. So b is probably already adjusted to optimize the performance of the model with the default cutoff. And indeed, changing pcut wouldn’t change the angle the red-blue border makes on the axes, it would just back it up or move it forward, in the direction normal to the plane.

**Hyperparameter: C**

Just like with linear regression, we can add an L1 or L2 penalty term to the loss function. Can alternatively add an elasticnet regularization, which is apparently some combination of L1 and L2. I’m not sure L1 and L2 would literally go into f like λ1|m|, or λ2m2, respectively, e.g., for L2



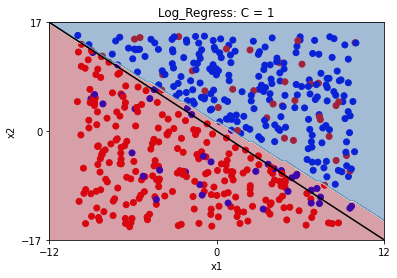
But however they do, they serves the same purpose – to tamp down on the reliance of certain wildly fluctuating features values. And in the case of L1 regression, we can sometimes get mj = 0, which will tell us which features we can effectively ignore, for modeling purposes. I should say, more than *can* ignore, it tells us if we *should* ignore the features, because they are irrelevant to the underlying phenomenon. In all cases, the regularization parameter is controlled by C, which, like with the linear SVC, is accounted as inversely proportional to λ (λ also called alpha). And like in SVC, higher C would mean that the model would try to fit the data more closely, to make avoiding misclassifications paramount. And of course this could cause overfitting. While, lower C would emphasize keeping the distance between classes and the border hyperplane at pcut = 0.5 as large as possible. And the drawback of this is possible underfitting. Here’s adjusting C on a pure model, using L2 regularization.

 A diagram of a log graph

Description automatically generated with medium confidenceA diagram of a graph

Description automatically generated

Doesn’t matter. And we get perfect fit of course, since it’s a linear classifier and we have a linear boundary. Here’s adjusting C on 10% outliers,

 A diagram of a graph

Description automatically generated A diagram of a graph

Description automatically generated

Doesn’t seem to do much in this case. Interesting to me that no matter how much I ramp up C, it doesn’t want to adjust the angle of the hyperplane to align with the black-border, even though doing so would eliminate some misclassifications.

**Hyperparameter: kernels**

I read that, like with SVM, we can use kernels to map our feature space into a higher dimensional space, which might be more amenable to linear separation that before. Don’t know how we do that.

**Comparing to other Models**

Then we can try to use the logistic regression fit to try to classify a new test point, **x**, as either 0 or 1. To do so, we’d usually make some cut-off like p = 0.5. And if f(**x**)>0.5, we’ll classify it as 1; otherwise 0. The logistic regression method might work better than say KNN if, as above, there is no sharp well defined boundary separating the two cases, and there are a lot of ‘outliers’. Looks like it would also require the 0’s and 1’s to be roughly separated by a d-1 dimensional hyperplane.

**Categorical Polynomial Logistic Regression**

It’s worthwhile to consider Categorical Polynomial Logistic Regression. Can see the Categorical Regression II file for the basic regression case. But we basically augment the dataframe by adding to it columns of powers of the numerical variables and products of the categorical variables. This basically gives us, instead of a hyperplane, a hypersurface. For instance, suppose we start off with a df (just showing X values, not y values), with two numeric columns and one binary categorical column.

|  |  |  |
| --- | --- | --- |
| x1 | x2 | c |
| 2.34 | 3.50 | 1.00 |
| 1.98 | 4.67 | 0 |
| 0.67 | 5.92 | 0 |
| 4.50 | 3.50 | 1.00 |
| 8.15 | 1.18 | 1.00 |
| 7.21 | 2.99 | 0 |

Then our normal logistic regression function would be:



And all points for which h > 0 would be classified as 1, and all points for which h < 0 would be classified as 0 (presuming usual p = 0.5 cutoff). And the boundary h = 0 is a plane in three dimensional space. But we could augment the numerical columns to allow for a quadratic hypersurface,

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| x1 | x2 | c | x1^2 | x2^2 | x1\*x2 |
| 2.34 | 3.50 | 1.00 | 5.48 | 12.25 | 8.19 |
| 1.98 | 4.67 | 0 | 3.92 | 21.81 | 9.25 |
| 0.67 | 5.92 | 0 | 0.45 | 35.05 | 3.97 |
| 4.50 | 3.50 | 1.00 | 20.25 | 12.25 | 15.75 |
| 8.15 | 1.18 | 1.00 | 66.42 | 1.39 | 9.62 |
| 7.21 | 2.99 | 0 | 51.98 | 8.94 | 21.56 |

Then our logistic regression function would be:



And still, all points for which h > 0, < 0 would be classified as 1, 0 respectively. But now we can have more varied behavior with our quadratic hypersurface. And of course we could do more than just a quadratic hypersurface, but also a cubic, etc. So the degree of our polynomial hypersurface could be considered a hyperparameter. We could further augment our X to allow for different behavior per c-value. Say we just allow the hypersurface to change its x1 parameterization per c-value. Then our X would be:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| x1 | x2 | c | x1^2 | x2^2 | x1\*x2 | x1\*c | x1^2\*c | x1\*x2\*c |
| 2.34 | 3.50 | 1.00 | 5.48 | 12.25 | 8.19 | 2.34 | 5.48 | 67.08 |
| 1.98 | 4.67 | 0 | 3.92 | 21.81 | 9.25 | 0.00 | 0.00 | 0.00 |
| 0.67 | 5.92 | 0 | 0.45 | 35.05 | 3.97 | 0.00 | 0.00 | 0.00 |
| 4.50 | 3.50 | 1.00 | 20.25 | 12.25 | 15.75 | 4.50 | 20.25 | 248.06 |
| 8.15 | 1.18 | 1.00 | 66.42 | 1.39 | 9.62 | 8.15 | 66.42 | 92.49 |
| 7.21 | 2.99 | 0 | 51.98 | 8.94 | 21.56 | 0.00 | 0.00 | 0.00 |

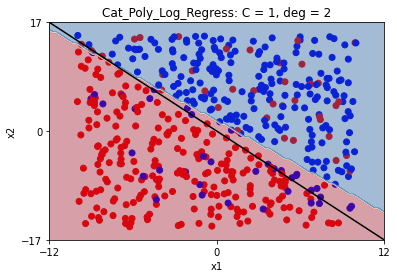
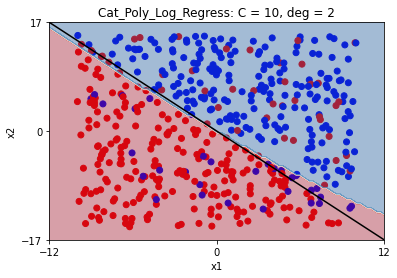
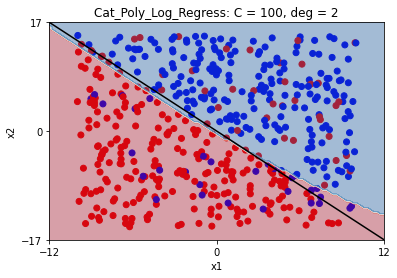
And our logistic regression function would go to:



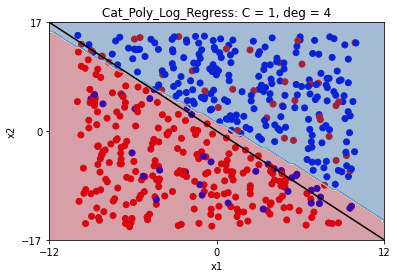
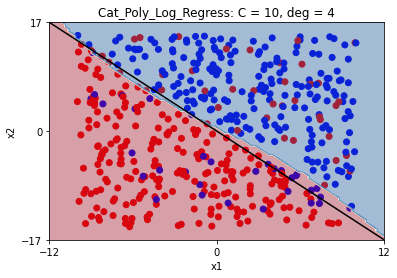
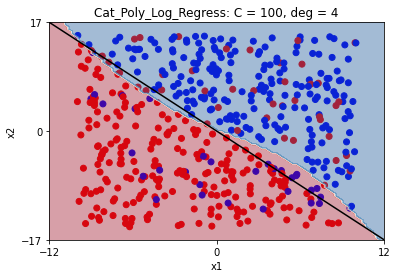
Now we’d have something like a decision tree hypersurface. We get one hypesurface for c = 0 (the default) and another for c = 1. Let’s look at a few cases:

**Hyperparameters: C and degree**

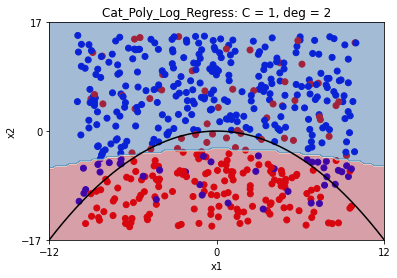
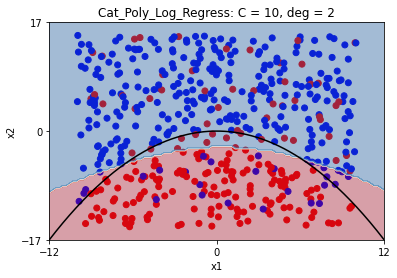
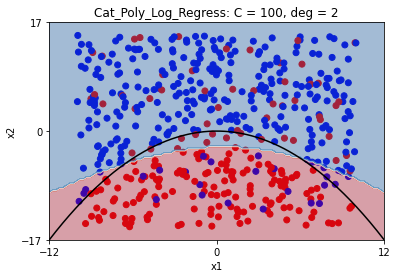
I’ll explore a few parameter combinations. Here’s C = 1, 10, 100 for a quadratic surface (deg = 2), with 10% outliers. And by the way, I’m using L1 regularization here. Can see that we’re slightly better off with a linear surface.

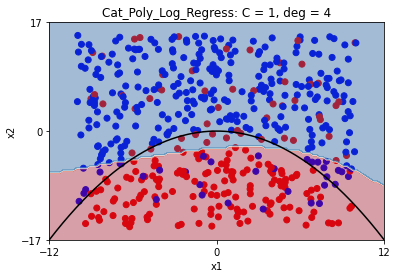
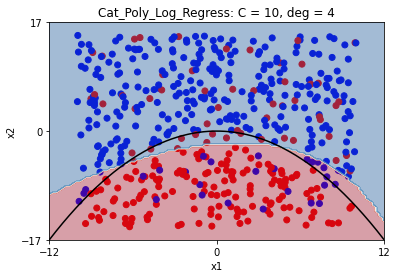
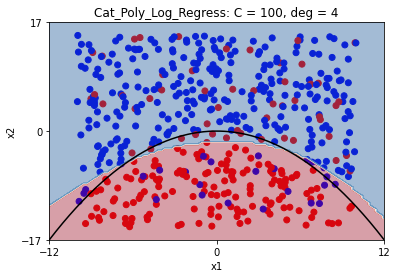
And I’ll do a cuartic surface now (deg = 4). We can see higher degrees allow for a little bit of overfitting.

The real advantage of this approach is with non-linear surfaces though, as discussed. Consider the same set of hyperparameters with degree 2,

and degree 4,

We get pretty good fit, when regularization is tuned down. Only SVC does better I’d say. Seems we get only marginally better results by going to even higher degrees and C’s.

**Multinomial Logistic Classification**

We can generalize our formula to accommodate classification into multiple categories. Formula would work like this, in say d = 3:



where i = 1,2,...,n denumerates the possibilities. So fi is interpreted as the probability of outputing variable i. And whichever fi is largest would be accounted the prediction. I think that logistic regression with this formula is often implemented via neural networks. How does this compare to the two-outcome case? Then we have:



And can see fb = 1 - fa. Also, can write fa as:



And can then redefine the parameters to write,



So this, along with the knowledge that fb = 1 - fa brings us back to the original two-outcome setup.

But assuming we have multiple possible outcomes, now we have to update the loss function. So…first let’s rewrite our LL in a form that makes the generalization more plausible. So to start, we have:



Now let’s call f(xi) = f1(xi), the calculated probability of a yes outcome. And we’ll write 1 - f(xi) = f2(xi) is the probability of a no outcome. And let’s write Yi = p1(xi), the probability of a yes outcome, and then 1 – Yi = p2(xi) will be the probability of a no outcome. p1(xi) and p2(xi) are either 1 or 0 of course. In this language we can say,



It then seems plausible that if we had three possible outcomes, with associated probabilities p1,2,3(xi) and fitted probabilities f1,2,3(xi), that we could generalize our loss function to:



Note p1,2,3(xi) will be 1 for one of the outcomes, and 0 for the other two. Since two of the outcomes is always zero, we can simplify our expression. Let the outcome which is certain to occur be denoted with the subscript outcome, and the other two with not-outcome. Note outcome could be any *one* of 1, 2, 3, and not-outcome would be the other two. Then we can write:



So we have:



This formula would apply equally well to the previous binary output case (case with single output node). Should introduce a formula it’s good to be aware of. If we just look at what’s inside the bracket of the log-loss, we will see the so-called cross entropy,



It’s measure the information loss (or something) when we replace an exact probability distribution pj=1,2,3 with an approximation fj=1,2,3. Note xi is just a bystander variable in this equation. pj and fj are probability distribution w/r to the parameter j, such that Σjpj = Σjfj = 1. And we can write the log-loss in terms of this as:



So yeah.

**Exploring the Model and Hyperparameters**

So here’s logistic regression run on a linear-ish boundary with 0% and 15% outliers.

A diagram of a log graph

Description automatically generated A diagram of a graph

Description automatically generated

Our results are pretty similar to the linear SVC. One could alter the performance of the model by adjusting the probability cutoff, pcut, that it uses to classify as 0 or 1. This cutoff is by default pcut = 0.5. And while pcut is not a hyperparameter per se´, we could have the model output the probabilities, pi, of all the points, and then tell it to classify the points as +1 if pi > pcut and 0 if pi < pcut. And we could make pcut = 0.55 or something. I’m thinking this would have the same effect as altering the parameter b in f(x)? So really, changing pcut, leaving b fixed, is probably the same as leaving pcut fixed (at say 0.5) and changing b. So b is probably already adjusted to optimize the performance of the model with the default cutoff. And indeed, changing pcut wouldn’t change the angle the red-blue border makes on the axes, it would just back it up or move it forward, in the direction normal to the plane.

**Hyperparameter: C**

Just like with linear regression, we can add an L1 or L2 penalty term to the loss function. Can alternatively add an elasticnet regularization, which is apparently some combination of L1 and L2. I’m not sure L1 and L2 would literally go into f like λ1|m|, or λ2m2, respectively, e.g., for L2



But however they do, they serves the same purpose – to tamp down on the reliance of certain wildly fluctuating features values. And in the case of L1 regression, we can sometimes get mj = 0, which will tell us which features we can effectively ignore, for modeling purposes. In all cases, the regularization parameter is controlled by C, which, like with the linear SVC, is accounted as inversely proportional to λ (λ also called alpha). And like in SVC, higher C would mean that the model would try to fit the data more closely, to make avoiding misclassifications paramount. And of course this could cause overfitting. While, lower C would emphasize keeping the distance between classes and the border hyperplane at pcut = 0.5 as large as possible. And the drawback of this is possible underfitting. Here’s adjusting C on a pure model, using L2 regularization.

A diagram of a graph

Description automatically generated A diagram of a graph

Description automatically generated A diagram of a graph

Description automatically generated

So the higher C looks better, probably better by a little than the SVC guys. Here’s adjusting C on 15% outliers,

 A diagram of a graph

Description automatically generated A diagram of a graph

Description automatically generated

Hmmm. Now SVC is doing better.

**Hyperparameter: kernels**

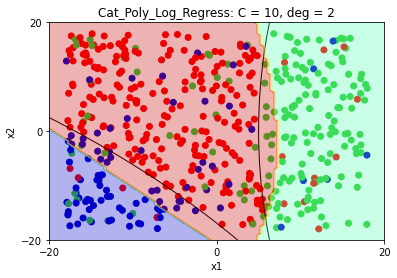
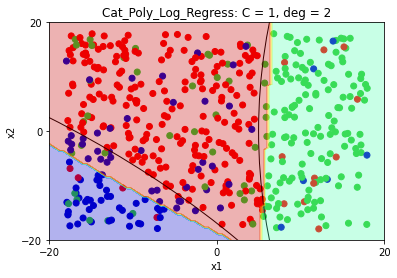
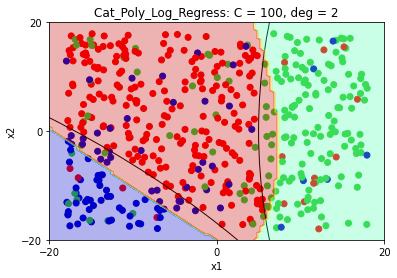
I read that, like with SVM, we can use kernels to map our feature space into a higher dimensional space, which might be more amenable to linear separation that before. Don’t know how we do that

**Categorical Polynomial Logistic Regression**

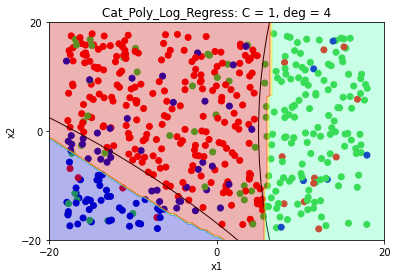
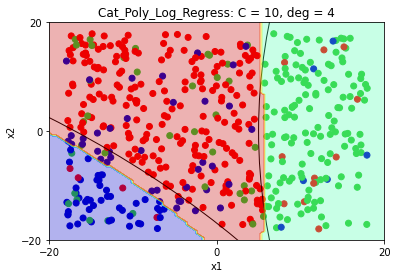
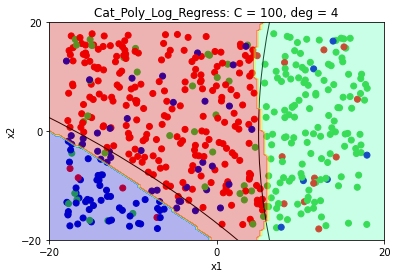
Now I’ll look at the performance of the categorical polynomial logistic regression model.

**Hyperparameters: C and degree**

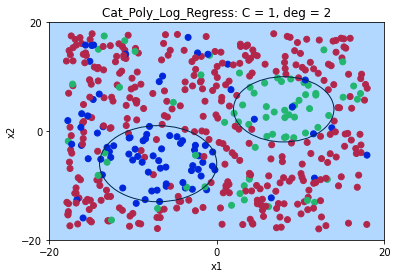
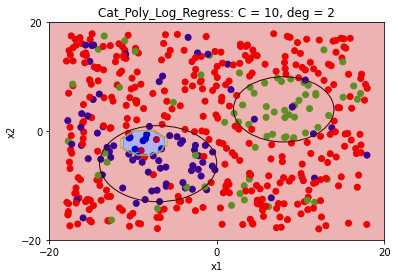
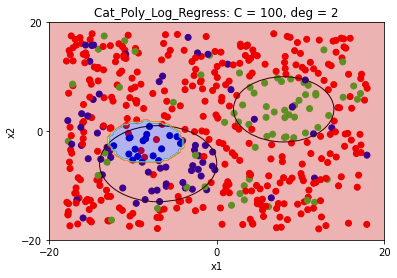
I’ll explore a few parameter combinations. Here’s C = 1, 10, 100 and deg = 2 for the linear surfaces with 15% outliers. And by the way, I’m using L1 regularization here. Can see that we’re better off with a deg = 1 fit.

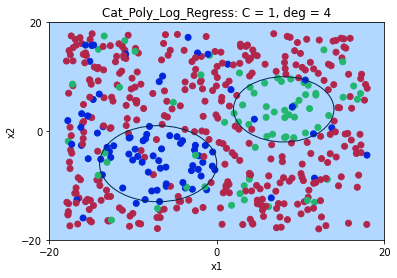
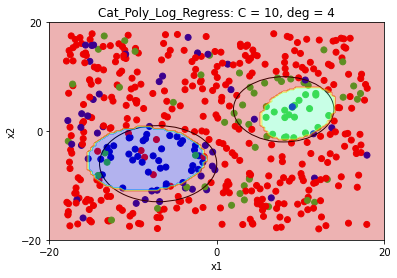
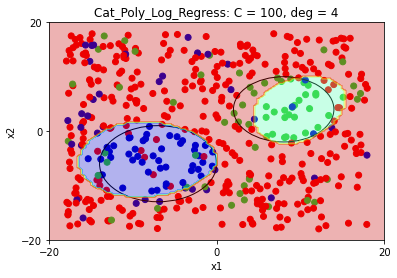
and here’s deg = 4. Always have that problem of getting the blue guy to edge up to the black line.

Now let’s look at the circular multi-class case. Here’s deg = 2 again. Pretty bad, as it only picked up the red and blue guys, and not very well.

Here’s deg = 4. If we lower the regularization enough, then we begin to get decent results. Still not better than SVM’s though.

And further increasing C and degree seems not to effect much.

**Another Appendix**

Since f(x1, x2, x3) → probabilities. Can we say f(x1,x2,x3) = P(y|x1x2x3)? Yes I think so. Then what is P(x1,x2,x3,y)? Well,



and I guess P(x1,x2,x3) would just be 1/n, where n = # data points, presuming there are no duplicate xi’s. So then we could say,

