**Preprocessing**

Going to consider a few approaches to handling missing data, outlier data, sparse data, and imbalanced data.

**Missing Data**

So first let’s consider some ways to handle missing data. Consider a table with data,

|  |  |  |  |
| --- | --- | --- | --- |
| **Loves Popcorn** | **Loves Soda** | **Age** | **Loves Cool as Ice** |
| Yes | Yes | 7 | No |
| Yes | No | 12 | No |
| No | Yes | 18 | Yes |
| No | Yes | 35 | Yes |
| Yes | Yes | 38 | Yes |
| Yes | No | 50 | No |
| No | No | 83 | No |

But suppose some of it were missing, like Loves Popcorn in row 4, and Age in row 6.

|  |  |  |  |
| --- | --- | --- | --- |
| **Loves Popcorn** | **Loves Soda** | **Age** | **Loves Cool as Ice** |
| Yes | Yes | 7 | No |
| Yes | No | 12 | No |
| No | Yes | 18 | Yes |
|  | Yes | 35 | Yes |
| Yes | Yes | 38 | Yes |
| Yes | No |  | No |
| No | No | 83 | No |

How do we handle it?

**Dropping Data**

One viable option, if we have a lot of data, is to simply drop the rows with missing data. In general, we can drop rows only if we’re sure the missing data occurs at random. A case where we might not have random missing data is if all the old people in a survey didn’t want to indicate their age. So if we dropped them, we’d lose any entire age group. Another problem with dropping data is that you might not have a lot of data to begin with, so dropping the bad data might leave you with not enough complete data to train on. Another problem with this is that, not having a way to handle missing data besides dropping the data (row) means that when we get incomplete data in the real world, that we have to predict on, we would have no recourse but to drop *it* too. But in the real world, we probably have to make some prediction, perfect or not, on imperfect data. So, we might need to impute data.

**Imputing Data**

If we need to impute data, there are a few options. If it’s numerical data we can impute with the mean or median, or mode of that column. Or maybe can see which column the numerical column data is most highly correlated with and linearly impute based off of that correlation.

Apparently there is a technique using Decision Trees and Random Forest to infer the missing data’s most probable value. So might want to review those files….but anyway, the main idea is this. If row x has missing data, then we quantify, with weights ranging from 0 to 1, how similar all other rows are to row x. And then take row x’s value to be the weighted average of the other rows’ values.

**1st Iteration**

But anyway, presuming our outcome, y, is binary, we start by segregating all rows by their outcome. So gonna highlight all the Loves Cool as Ice = Yes’s.

|  |  |  |  |
| --- | --- | --- | --- |
| **Loves Popcorn** | **Loves Soda** | **Age** | **Loves Cool as Ice** |
| Yes | Yes | 7 | No |
| Yes | No | 12 | No |
| No | Yes | 18 | Yes |
|  | Yes | 35 | Yes |
| Yes | Yes | 38 | Yes |
| Yes | No |  | No |
| No | No | 83 | No |

We could create a similarity matrix, inserting a 1 for every pair of rows that are similar, i.e., have the same outcome. Note if row a is similar to row b, then row b is similar to row a. So we get a symmetric matrix. But we evidently don’t count a row as similar to itself, for these purposes. I’ve highlighted the two rows which have missing data.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | row 1 | row 2 | row 3 | row 4 | row 5 | row 6 | row 7 |
| row 1 |  | 1 |  |  |  | 1 | 1 |
| row 2 | 1 |  |  |  |  | 1 | 1 |
| row 3 |  |  |  | 1 | 1 |  |  |
| row 4 |  |  | 1 |  | 1 |  |  |
| row 5 |  |  | 1 | 1 |  |  |  |
| row 6 | 1 | 1 |  |  |  |  | 1 |
| row 7 | 1 | 1 |  |  |  | 1 |  |

And then we fill in the missing values with a weighted average of the values within that class. The weight is the ‘similarity’ of the row to the one with the missing value. For instance, we’ll have for the missing Popcorn value:



This is borderline. Typically we’d round up to 1 and say P4 → yes, which happens to be the incorrect answer. And for the missing Age value:



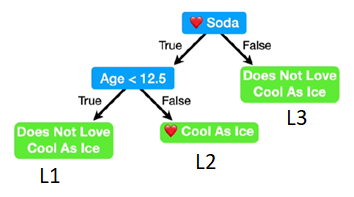
So in our first order iteration we’d say,

|  |  |  |  |
| --- | --- | --- | --- |
| **Loves Popcorn** | **Loves Soda** | **Age** | **Loves Cool as Ice** |
| Yes | Yes | 7 | No |
| Yes | No | 12 | No |
| No | Yes | 18 | Yes |
| Yes | Yes | 35 | Yes |
| Yes | Yes | 38 | Yes |
| Yes | No | 34 | No |
| No | No | 83 | No |

which is quite off, but it’s a start.

**2nd Iteration**

Now we create a random forest of decision trees, based off of the filled in guess data. Say our random forest consists of these two trees,

 Diagram

Description automatically generated

And we run all the data through each of the decision trees, and work out where the rows end up. For the first tree, we have:

row 1 → L1

row 2 → L3

row 3 → L2

row 4 → L2

row 5 → L2

row 6 → L3

row 7 → L3

So rows 3, 4, 5 all end up in L2, and rows 2, 6, 7 all end up in L3. Rows that end up in the same leaf are accounted ‘similar’, and we add 1 to the similarity matrix (starting over on that guy). So we add 1 to elements (3,4) (4,3) (3,5) (5,3) (4,5) (5,4). And we add 1 to elements (2,6) (6,2) (2,7) (7,2) (6,7) (7,6):

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | row 1 | row 2 | row 3 | row 4 | row 5 | row 6 | row 7 |
| row 1 |  |  |  |  |  |  |  |
| row 2 |  |  |  |  |  | 1 | 1 |
| row 3 |  |  |  | 1 | 1 |  |  |
| row 4 |  |  | 1 |  | 1 |  |  |
| row 5 |  |  | 1 | 1 |  |  |  |
| row 6 |  | 1 |  |  |  |  | 1 |
| row 7 |  | 1 |  |  |  | 1 |  |

Now we run the data rows through the second decision tree:

row 1 → L3

row 2 → L3

row 3 → L3

row 4 → L1

row 5 → L1

row 6 → L2

row 7 → L2

So row 1, 2, 3 all end up in L3, and rows 4, 5, all end up in L1, and rows 6, 7 all end up in L2. So we add 1 to elements (1,2) (2,1) (1, 3) (3,1) (2,3) (3,2). And we add 1 to elements (4,5) (5,4), and 1 to elements (6,7) (7,6):

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | row 1 | row 2 | row 3 | row 4 | row 5 | row 6 | row 7 |
| row 1 |  | 1 | 1 |  |  |  |  |
| row 2 | 1 |  | 1 |  |  | 1 | 1 |
| row 3 | 1 | 1 |  | 1 | 1 |  |  |
| row 4 |  |  | 1 |  | 2 |  |  |
| row 5 |  |  | 1 | 2 |  |  |  |
| row 6 |  | 1 |  |  |  |  | 2 |
| row 7 |  | 1 |  |  |  | 2 |  |

Might note that initially, two rows were accounted similar if they had the same outcome. And they were dissimilar otherwise. Now two rows will be similar only if they not only have the same outcome, but also have it in the same place (leaf). So just having the same outcome is no longer sufficient. But if they don’t have the same outcome, then they definitetly aren’t considered similar, just as was the case before. Also note that two rows which were similar in the first iteration, may not be similar in the second iteration, and vice versa. And now redo the calculation, with the updated weights.



Again we’d round up to 1, and say P4 → yes, at the moment. And for the other:



which is much closer to the actual value. So we’d update our table to:

|  |  |  |  |
| --- | --- | --- | --- |
| **Loves Popcorn** | **Loves Soda** | **Age** | **Loves Cool as Ice** |
| Yes | Yes | 7 | No |
| Yes | No | 12 | No |
| No | Yes | 18 | Yes |
| Yes | Yes | 35 | Yes |
| Yes | Yes | 38 | Yes |
| Yes | No | 58 | No |
| No | No | 83 | No |

**3rd Iteration, etc.**

And then we’d just keep repeating, until the values seem to be converging.

Parenthetically, the similarity matrix can be thought of as saying how close each pair of rows is. We could convert this into a ***dissimilarity* matrix**. Let’s say we take our similarity matrix and divide by the number of decision trees we’ve run it through. Then in our case we’d have:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | row 1 | row 2 | row 3 | row 4 | row 5 | row 6 | row 7 |
| row 1 |  | 0.5 | 0.5 |  |  |  |  |
| row 2 | 0.5 |  | 0.5 |  |  | 0.5 | 0.5 |
| row 3 | 0.5 | 0.5 |  | 0.5 | 0.5 |  |  |
| row 4 |  |  | 0.5 |  | 0.5 |  |  |
| row 5 |  |  | 0.5 | 0.5 |  |  |  |
| row 6 |  | 0.5 |  |  |  |  | 1 |
| row 7 |  | 0.5 |  |  |  | 1 |  |

And if we then subracting every entry from 1, we’d have:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | row 1 | row 2 | row 3 | row 4 | row 5 | row 6 | row 7 |
| row 1 |  | 0.5 | 0.5 |  |  |  |  |
| row 2 | 0.5 |  | 0.5 |  |  | 0.5 | 0.5 |
| row 3 | 0.5 | 0.5 |  | 0.5 | 0.5 |  |  |
| row 4 |  |  | 0.5 |  | 0.5 |  |  |
| row 5 |  |  | 0.5 | 0.5 |  |  |  |
| row 6 |  | 0.5 |  |  |  |  | 0 |
| row 7 |  | 0.5 |  |  |  | 0 |  |

We could call this a distance matrix, or a dissimilarity matrix. Can see that, as far as the outcome is concerned, rows 6 and 7 are identical (at this level of analysis – if we include more trees, then they’ll likely still be very close, but not identical)

**Outliers**

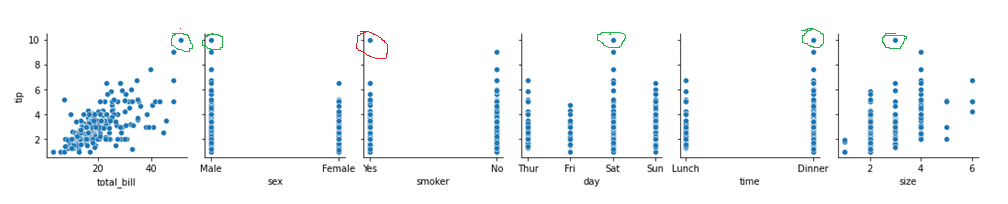
Now let’s consider how we might handle outlier data. First, what is an outlier? I guess I’d say an outlier is a point whose value lies well outside the range of its local neighborhood of all independent variables. So for instance, the red dot on the left and the blue dots on the right would be considered outliers.

A diagram of a graph

Description automatically generated A graph of a graph with a red line

Description automatically generated

It’s important that we consider the entire local neighborhood when determining whether a point is an outlier or not. Consider the blue dots in the plane-graph. If we just plotted z vs. x, or z vs. y, the blue dots wouldn’t stand out, because their z values are not atypical for a given x and all y, or a given y and all x. They are only atypical for a given x *and* y (or neighborhood about it). Conversely, consider the graph on the left. Even though the red dot seems to be an outlier here, if we had in addition to x, another variable, say, y, to plot the z values against, it might turn out that the red dot has a y value way further out than any of the other points, and its extreme y value might be why it has such a large z value. Or maybe y would be a categorical value, and the red dot might have a different y value than all the other blue dots. For instance z could be home prices, x square feet, and y location. Maybe the red dot is at the beach, while all the other points are in the suburbs. Just to drive that point a little further. Consider the tips dataset, where we want to predict the tip from the other variables, like total\_bill, sex, smoker, day, time, (party) size. If we plot tip vs. each of these variables, as below, we might see the red circled point in the smoker = yes column and suspect it to be an outlier.



But we cannot judge without more context. We have to see what tip points with similar total\_bill, sex, day, time, and size values have. And if we consider the red circled point in the context of the other variables (now point circled green), we see that it isn’t really much of an outlier at all. So we need to consider the local neighborhood.

But in this respect, the feature space matters too. For instance, if we put every point in its own category, then there would be no local context from which to judge outlier-ness. This kind of happens with that housing dataset with 1200 different locations. If we treat every location as being an important identifier, then there will likely be so few points per location that determining outliers from that set will be sketchy. And in reality, it is unlikely every location matters. Maybe some do, or maybe they self segregate into two varieties, like ‘rich’, and ‘poor’ locations, but all the rest of the variation is noise. And maybe then from segregating into just these locations, we can detect outliers within those two sets.

Another point. As a practical matter, it might be hard to judge an outlier as such if we’re looking at the local context. Numerically, we could calculate the distance between points in featuer space, say. Then for any given point, we could find its, say, 25 nearest neighbors. And then we could remove any outlier from this group. But what if for one point, the neighbors are are within some distance d, whereas for another point, the neighbors are spread out over a distance 100d? We would expect much greater variation possible in the latter case. Or what if 9 of the neighbors is within a distance d, but the 10th neighbor is 100d out? Then almost certainly we’d end up removing the 10th guy.

Maybe a better way is to run a regression through the data, and remove all points whose absolute value residuals are greater than some cutoff point. But then this is regression-dependent. Can average a bunch of regressions, but results are kind of messy here too. Could just treat the outlier removal as part of any regression, and just test to see which regression program is best at outlier-removing *and* fitting.

Next question is, ‘what should we do with outliers?’. I guess we’d drop them. For three reasons. One would be that the outlier might actually be an error, and so of course it should be dropped. The other is that, even if it is a true value, it probably means that our model doesn’t have the independent variables necessary to fully explain its presence. And so keeping it would only skew our model’s performance for no good reason. For instance, consider the following example (dependent variable now y, instead of z).

A diagram of a graph

Description automatically generated

Using the mean square error loss function, the blue line is the best fit curve for the non-outlier data points. The red curve is the best fit line for all the points, say. But if we include the outlier, then our model will lose its predictive power. A third possibility for the presence of the outlier is that it is still only a random fluctuation, but since it is, in this case, the only random fluctuation so far below the line, it will affect its slope and intercept. If on the other hand, we had more such extreme fluctuations, then perhaps they wouldn’t have such a drastic effect on the shape. We’d have something like this:

A diagram of a graph

Description automatically generated

But extreme fluctuations occur, by their nature, only randomly, and so it is perhaps unlikely that we will have enough of them to balance each other out. And so we should probably remove the point.

So I guess in general, we’d need to consider all independent variables together, isolate a cluster of data within some some region of feature space, Δ**X**, and determine the standard deviation, say, of all of these points, σΔ**X**, and eliminate the outliers. I think keeping points within (μ-3σ, μ+3σ) is the standard. Another standard is keeping points within (Q1 – 1.5IQR, Q3 + 1.5IQR), where IQR = Q3 – Q1. I feel like the IQR method would be better for detecting outliers, as σ would itself be affected by the outlier, whereas the quartiles/quantiles would not be, so much. On the other hand, if you have a lot of data within (Q1, Q3), but also a decent amount of data outside the range, you might classify the decent amount of data as outliers, because we put too much weight on the really nicely behaving data. Anyway, the Δ**X** method will work in principle. But the easiest way to do this is based off of the regression plane/hypersurface itself. One way to implement this is:

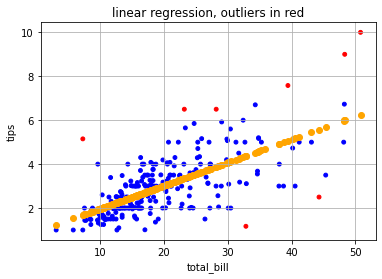
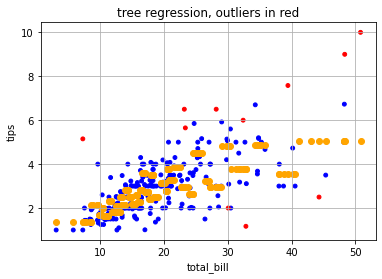
1. Draw a regression plane through all points, R0.
2. Identify outliers based off of R0. Take these out, and put them in a set S1. Using the quartile identification method, (Q1 – 1.5×IQR, Q3 + 1.5×IQR), seems best.
3. Redraw a regression plane, R1, through remaining points.
4. Put outliers, S1, back in, and identify new outliers based off of R1. We’ll call these S2. And take these points S2 out.
5. Redraw a regression plane, R2, through remaining points.
6. Put outliers, S2, back in, and identify new outliers based off of R2. We’ll call these S3. And take these points S3 out.
7. Redraw a regression plane, R3, through remaining points.
8. Put outliers, S3, back in, and identify new outliers based off of R3. We’ll call these S4, etc.
9. And keep doing this until the outlier sets converge to a final result, i.e., Sn = Sn-1. And we’d say our final regression plane is Rn. We can’t just go with S1 because if outliers are really large, then it can skew R0 and cause it to misclassify acceptable points as outliers.

This does seem to converge in practice. FYI, I implemented this and made a histogram the residuals/residues,

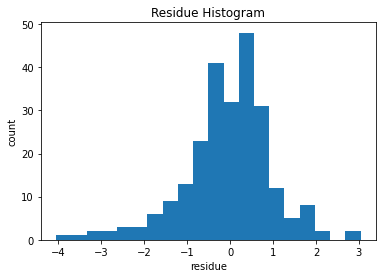
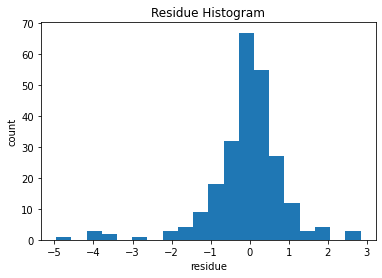
A graph of a graph

Description automatically generated with medium confidence

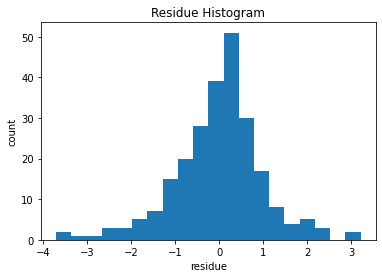
The red markers are where the program considered outliers to begin. Can see that the histogram rises slightly at these points, roughly. So I guess you could say the outliers are where we get a break in the normal distribution? Anyway. But before we get too confident, consider these graphs,

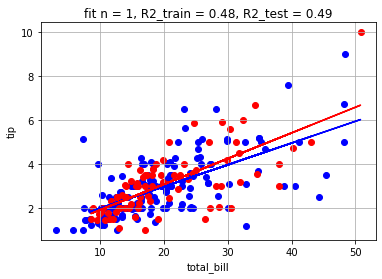
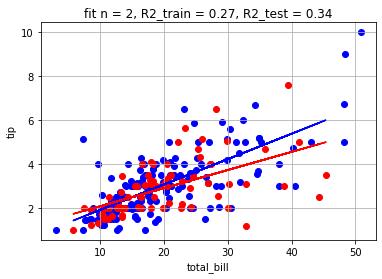
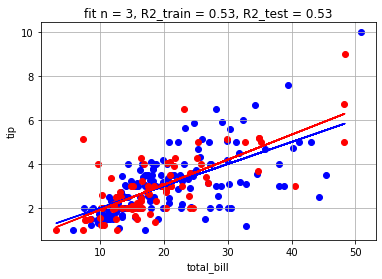
I did the linear regression and outlier removal program on the left, and the decision tree regression and outlier removal program on the right. The regression predictions are the orange points, the data are the blue and red points. The reds are classified as outliers. Now arguably, none of the data should be an outlier, except maybe that red dot at total\_bill = $8 or so. And I set the outlier threshold at beyond Q1 – 2×IQR, and Q3 + 2×IQR. So that’s generous. You’d think the decision tree regressor would be better at capturing outlies, because it responds more locally, than linear regression. But can see it doesn’t necessarily do a better job, because a point can be an outlier for a given Δx range, but not an outlier for the Δx range right next to it. So decision tree would classify outlier status based on where it draws these Δx intervals, and shifting them over just a bit to the left or right could have large impact on the classification. This shouldn’t be so. On the left and right are the linear and tree histograms for the errors, i.e., residuals, or residues, whatever.

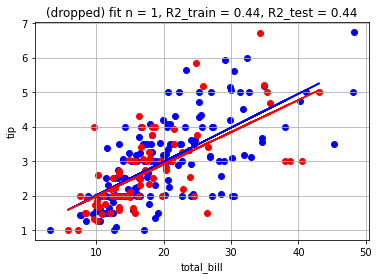
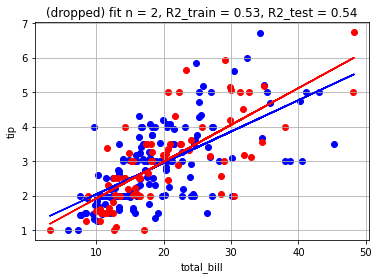
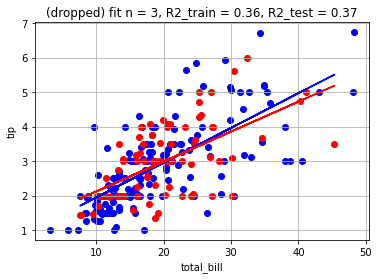
It’d be hard to tell what should be an outlier from the linear histogram. The decision tree histogram makes a decent case for some of the extremely negative points (large tips, as residues are calculated as: regression point – data point). But as we see from the scatter plots, classifying those large tip points as outliers is dubious. Below I changed the criterion for the linear regression outlier removal program to ‘outlier is beyond μres ± 3σ’.



As it turns out, σ = 1 just about, so points beyond ±3 are outliers. This looks reasonable from the histogram. But we aren’t compelled to this conclusion. This all being said, eliminating the few points, doesn’t really change the regression curve. For instance, below, I did Kfold cross validation with 3 splits on the tips dataset. The blue points are training data, red points test data, blue line the regression on training data, and red line the regression on testing data. R2\_train, R2\_test are the R2 scores for training regression line applied to testing data, and testing regression line applied to training data. The testing regression line and R2\_test are just there to compare to see what the best fit line and R2 values are for the testing data alone.

And I did the same thing with the data, having removed the lin\_regress predicted outliers first. Can see there’s not much difference honestly. But the match between R2\_train and R2\_test is a little better with the outliers removed.

So even if a case can be made for keeping the outliers, I don’t think it hurts to remove them either. Well, we can occasionally get worse results for small datasets like this one (I’ve found better and worse results depending on the number of splits when doing cross-validation). But I think this is just a random fluctuation kind of thing. But it *certainly* helps to remove them when they’re extreme outliers. I got *much* better results on the housing price dataset when removed outliers. So I guess we should just go ahead and do it.

Should probably do cross-validation on this procedure. So we’d do, say, 5-fold cross validation, training on 80% and testing on 20%. Leaving the testing aside for a second, for each of the training folds, we’d perform the above steps. If we’re correctly identifying outliers, then each time a prospective outlier appears in one of the training folds, it will be identified as such. So I guess I’d say the true outliers are those rows which are always identified as an outlier every time it appears in the training set. Once the outliers are identified, we’d remove them from the data set? And then we could perform the usual ML algorithms. This procedure can be readily generalized to Neural Net Regression. And Decision Tree Regression: can just replace regression plane, Rk, with decision tree, Dk.

Well, is there another way to do this? Is there a way to quantify in the loss function the idea that we’d rather fit 95% of the data really well, and 5% horribly, than fit 100% of it just mediocrely? We could use a different loss function. If we used mean absolute error, rather than mean square error, then we’d be penalizing outliers less, and so there would be less incentive to skew the regression curve close to them. Or maybe if we had a loss function that was more concerned with approximated the most frequent value of the points within a neighborhood, rather than the mean value? But generally, I think we should just remove the outliers.

Can also adapt this procedure to remove outliers of Classification Problems?

Having removed outliers, it’d be nice to be able to quantify their location and spread, so we know where our predictions are most likely to be off, and by how much.

Another point should mention. What if we *want* to identify outliers, their value, and when/where they occur? There might be a pattern to their appearance – for instance in a regression model, they might show up every Δx or something. Then we’d want our ML model to predict their occurance, and their magnitude. So a true outlier is something your data can’t predict, because it doesn’t have the right kind of data. And so you’d want to either remove it yourself, or train your model to ignore it. But in this case, we’d be presuming that we do have the data to predict it. We’d just need to consider a more complex/comprehensive model.

One more thing. Whatever outlier removal program you fit to the training data should be applied to the testing data as is. So if you’re using quartiles or std’s calculated from the training data to estimate outlier thresholds, you must use those same quartiles and std’s on the testing data – don’t redo on the testing data. **Should go back and check that I did this in my outlier removal program.**

**Scaling Data**

It’s often a good idea to scale data (i.e. columns) before running ML models, so that each column carries the same weight. This is probably necessary for algorithms that use some sort of distance metric, so KNN for sure. But even logistic regression, and neural networks seem to need it. Because at least, they use gradient descent to optimize themselves, and this implicitly uses a distance metric. So first, I’d say if data in a column varies over several orders of magnitude, then I’d use a log-scale it: Xi -> ln(Xi) or something. Once all the columns are within a couple orders of magnitude, then I’d use scaling that puts each column within range (0, 1): Xi -> (Xi – Xmin)/(Xmax – Xmin). Or could normal scaling, that Xi -> (Xi – μ)/σ.

One more thing. Whatever scaling program you fit to the training data should be applied to the testing data as is. So if you’re calculating the min/max or average/std’s of the training data to apply a scaling method on it, you must use the same training data min/max or average/std’s when transforming your test data. So for instance, if you use a min/max scaling on the training data, it is possible that when this is applied to the testing data you’ll get a point above 1 or below 0. That’s okay.

**Sparse Data**

Another problem we can run into is sparse data. Say we want to run a regression model on the following data:

A line graph with dots

Description automatically generated

If we do train, test, split, we could get something like:

A graph with red and blue dots

Description automatically generated

I guess there’s nothing to be done about it per se´? From the variability of the fit, one might suspect, w/o looking at the training data, that we’re overfitting the data. Although a linear fit seems reasonable, maybe just a horizontal fit is better, i.e., just approximating with the mean. On the other hand, if we tried a quadratic fit, I suspect the fit with the test data would be even worse. So how do we tell which model is best: horizontal, linear, quadratic? I guess we’d just have to do cross-validation, and take the average of the scores. Whichever model has the best average would win. Of course this is hat we’d do in any event. So I guess with sparse data we’ll just expect more variability in the fit parameters, and the R2 (or whatever) value.

Another question…I’m a little concerned that in the first graph, I’d classify the two blue dots as outliers. Would this be so? Say the red dots are close to the line, with residuals [0, 1, -1]. and blue dots have residuals [-5, -4]. Then the residuals are [-5, -4, -1, 0, 1]. The quartiles are Q0,1,2,3,4 = [-5, -4.5, -1, 0.5, 1]. The IQR = 5. Then outliers would be outside the range (Q1 – 1.5IQR, Q3 + 1.5IQR) = (-4 – 1.5(5), 0.5 + 1.5(5)) = (-11, 8). Okay, so they wouldn’t be.

**Imbalanced Data**

Another problem is imbalanced data sets. For instance, if you’re trying to discover fraudulent transactions, you will have a data set with perhaps a 100/1 ratio of allowable vs. fradulent transactions. And if you do a random sample of this data set to get an X\_train, y\_train, etc., we probably won’t have enough fraudulent samples to adequately train on. There are a few remedies. Let case A be the allowable transactions, and case B be the fraudulent ones.

A green rectangular object with black text

Description automatically generated

**Undersample case A**. So we could randomly sample 1000 case A rows, and combine with the 1000 case B rows, and then use the 2000 row data set as *the* data set. I think this is called **downsampling**. Hopefully we wouldn’t be missing anything special from case A by taking just 1000/99000 of its rows. One way to address this is to use an **ensemble method**. Basically we’d pair the 1000 case B’s with the 99 different 1000 case A’s. And we’d do an ML algorithm for each of these 99 pairings, and then use for our test data, we’d run it through all 99 ML algorithms and take the majority vote prediction.

**Oversample case B**. Or we could duplicate case B 99 times so that we have effectively 99000 B cases, same as A. Then we can randomly sample as usual. This seems a better approach? But then this might place undo emphasis on the case B, and so the algorithm might not learn case A as well, and/or might overfit case B. One way to address this is the **SMOTE** (Synthetic Minority Oversampling Technique). This uses the KNN algorithm to generate synthetic minority cases that we can include in the dataset.

**Focal Loss**. This uses a special weight function that weighs case B more heavily than case A.

By the way, we should apply these methods only to the training data sets. Don’t want to introduce anything artificial into the testing data sets.