CSCC11 Kaggle Bonus - 1000831237

Chun Ho, Ho

Tasks

One hard “Job” competition: “Allstate Claims Severity”

Build your own baseline solution.

Experiment with a more advanced solution from someone else.

Try to improve on the advanced baseline by exploring.

Write a report on things you tried, your analysis and findings.

Share your code and report on github.

Version control and submission

https://github.com/Numbingbird/Kaggle-Allstates

Report

Problem description:

Every accident is unique, and insurance claim prices must be built to match. Every case is complicated and personal, making them hard to estimate. However, perhaps it is not an impossible task through machine learning techniques.

Given data: 188310 cases with 130 dimensions (features) each, and their corresponding price, presumably in dollars.

The last 14 of these features are continuous values in (0,1). However, the rest are letters, and some dimensions (ex. 116) often have multiple letters, though their length is not fixed. For example, entry (113,1) is ‘S’ but entry (113,2) is ‘BM’. This indicates that some feature engineering may be required.

For this task, Kaggle evaluates your model using a hidden answer set, having provided a validation set. However, for reasons described below, this dataset was not used. Instead, a random validation set is partitioned from the given, known set.

Personal goal:

Unfortunately, I did not have much time to work on this project. I decided on making the most of my results by focusing on programming generic concepts that will be undoubtedly useful in the

future. This way, any future projects should be much easier, as opposed to having a specific algorithm that may not be applicable.

Your baseline method:

First was importing and cleaning up assignment 1 of CSCC11, which employed least square basis function regression as described in class and class notes. By generating a graph comparing K values and corresponding error, one can manually choose K that best models the data. However, this model can only be used on one dimension(feature) at a time.

This is the core model of the final solution, though the project was made with the intention of swapping out models as desired. At first, this could only be applied to the last 14 given features, which were already numerical.

To address this, some feature engineering on the remaining 116 features is necessary. They hold almost ten times the data, which is certainly significant.

This step was also done with future projects in mind. A plug for a translation function was created, and a simple mapping of ‘A’=1, ‘B’=2, .. ‘Z’=26 was plugged in. This is clearly a poor representation, as these were classes, not discrete values in the first place. Thus regression should not be applied, but considering my personal goal of the project, I am fine with it.

Another problem appeared when I got a late start on this project. Unfortunately, the competition ended alongside the exam period. Consequently I could not submit any results for the competition’s assessment, and had to create my own test set. This was done by randomly partitioning out a tenth of the given data for validation. This partitioning is redone every time the programs are run.

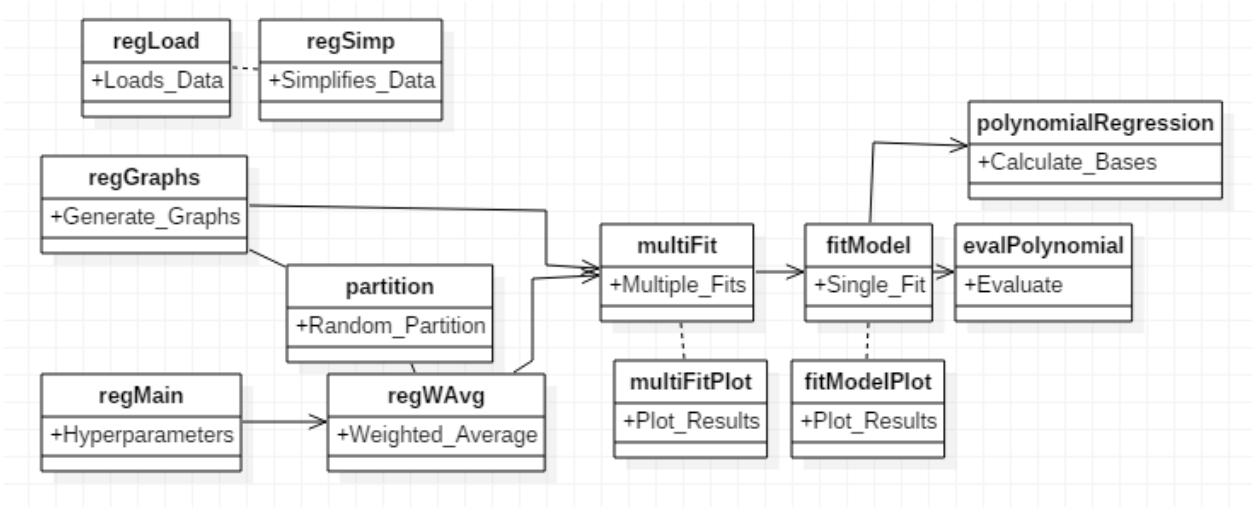
Advanced baseline from the Kaggle community:

The generic method being explored is boosting for discrete values. Although boosting was discussed to be an ensemble method for classification, there is nothing about the idea preventing it from being applied to discrete values. Many Kaggle solutions also employ methods of both boosting and bagging. I chose to employ the simplest concept, weighted averaging. Given models, hyperparameters and weights, recalculate and return a weighted average.

Normally, blindly averaging the Y results would be a poor idea, as some features may not be as indicative or accurate as another. This can be resolved by smartly choosing the weights. There were ways of minimizing error by manipulating weights on Kaggle. Again, these were not incorporated due to lack of time and my decision to focus on the framework for future plugin.

By using a basic weighted average, I established a connection between features, albeit crude. Still, it is an improvement to the baseline that was only able to take one feature at a time. Now the model is capable of receiving input from many features.

Your improvements:



The remaining simple tasks are now to choose K for every feature, as well as the weight that feature has for weighted average.

K was chosen through generating graphs for all 130 features, and manually selecting them. This is because having the computer choose the lowest error would result in an overfitted K.

The weights were chosen based on how much error those K have. For example, for some choices of K we have errors [3.6, 3.3, 3.9]e+07. Although there is a way to choose the best weights on Kaggle, I do not have the time to explore it. I chose a simple method of taking abs(E-4), as all the errors were within [3,4]e+07. Simply put, it is a crude way of giving the features with less error, more weight. This is to say, those features which can more accurately describe the end result are given more weight.

Of course, the weights must be standardized (sum to 1).

Unfortunately, manually choosing weights is clearly evidenced my own tests to be suboptimal. Even choosing random weights gave a lower error. Further details provided in below Testing logs section.

Testing logs

Though not well recorded, there were some interesting results from the testing that occurred in between steps. The following is mostly my notes during testing.

Low K values, K = [4;2;3;3;6;4;2;2;6;4;3;3;3;2];
Individual errors are around 3.8e+07, though some approach 4e+07
In this round of testing, these K values were chosen based on graph results from 14 dimensions, randomly partitioned on Dec-15.
Equal weights (just averaging every dimension) gives no improvement to error.
Manually setting weights based on the errors from which K was chosen gave 3.64, a -7e+05 improvement to error.

```
W =
   0.0768   0.0792   0.0722   0.0698   0.0698   0.0698   0.0698
   0.0698   0.0698   0.0698   0.0698   0.0698   0.0698   0.0733
ans =
  3.6374e+07
```

Random weights gives a noticeable, but not significant increase in accuracy, around -8e+05 improvement to error.

```
W =
   0.1251   0.0364   0.0168   0.0706   0.0445   0.0446   0.1875
   0.0265   0.0651   0.0131   0.0936   0.0328   0.2344   0.0089
ans =
  3.6298e+07
```

Interestingly, random weights have noticeably better performance. There should be a way to minimize error through manipulating weights, left for a future project. (addendum: Kaggle has many methods)

The result using weighted boosting over 130 features gave an error of 3.5739e+07. While this is a significant improvement from the previous attempts, a test on Dec-17 revealed that regression on dimension 80 alone gave ~3.0e+07 error.
This event, where the weighted average performed worse than a single dimension, did not occur on the 14 continuous features. Of particular importance is that the boosting included this dimension 80 and gave additional weight to it. This indicates that weight choice was indeed far too arbitrary, and has a significant effect on the end result.

Anything else that you learned in the process:

A single error in any step can result in catastrophic error.

Not just limited to poor programming. Feature engineering, selection of model, selection of hyperparameters, etc. In this project, I intentionally ignored selecting a better model, opting for boosting instead. However, even the selection of boosting method is critical to the end result. Perhaps a useful approach to machine learning problems is to create a plug-and-play framework where one can mix and match their choices at will, like a custom car.

Feature engineering can be troublesome in various ways.

In both the sense of choosing how to do feature engineering, and the actual processing itself. The decisions of feature engineering can negatively impact the rest of the project. The processing of data takes a very long time - in my case, it took around 10 minutes.

Running time is a very real concern.

On this dataset of 188310x130, runtimes become very noticeable, mostly when generating graphs. The number of operations comes out to 188310 elements * 130 features * MAX_K, which was generously set to 15. Generating graphs proceeded at a trackable, steady pace and took around 30 minutes.

Given that code for what I had interpreted to be a side project became large enough for wait times, one shadows to imagine the amount of time required for real world big data.