Worker Compensation Kaggle - A Reflection (STA303/1002 version)

Eric Zhu

03/04/2021

Introduction

For one of my statistics courses (STA303/1002) on applied linear regression at the University of Toronto, we were tasked with some professional development. As I have a strong interest in machine learning and predictive modelling, I decided to do a Kaggle competition. Our department regularly sends us information about professional development and along with that comes links to various competitions, and in sometime in February of this year, we were notified of **this** actuarial competition on simulated data. The competition was originally intended to end in the beginning of March 2021, but has so far been extended to the beginning of April 2021. I personally only worked on this over the Spring University of Toronto reading week, and this competition was really a chance for me to explore various methodologies since my first ML focused internship after first year (summer of 2019).

The goal of the competition is to predict worker insurance claims using various features (predictors). In particular, there are 11 categorical/continuous features that we're given access to, while there are 4 date or text features in the dataset. Personally, I haven't worked a lot with time dependent data or text data in order to best know how to proceed with that data, so I decided to consider this subset of features:

`Age, Gender, Matrital Status, Dependent Children, Dependent Others, Weekly Wages, Part-time status, Hours worked/week, Days worked/week, Initial Incurred Claims Cost`

Since the goal was prediction rather than interpretation of our findings, I decided to approach this from a solely prediction mindset, i.e., I did not care if I used so-called black box models. Although, certain black box models (namely neural networks) are generally time consuming to tune, so my overall strategy was to pursue those models with less overall hyper-parameters, such as boosted trees. Additionally, one big drawback to doing a competition like this from a sort of "generic" prediction point of view is that we have very limited domain information that we can use to reason about the features in our dataset, and we wouldn't know what to do with things like influential points, missing points etc...

While there were no `NA` values in the dataset, we did have a rather "small" dataset by ML standards, i.e., only 54000 samples (rows). So it'd make sense to either use bagging (bootstrapping) with cross validation or some sort of imputation method to help train the models. So in the next section, I'll go into depth about the ensembling methods I used to greatly improve the predictive capabilities of my models.

A Bayesian Foray

I did originally try to use a Bayesian method to impute more data values using a multivariate normal distribution with a identity covariance matrix. The idea was to just produce more samples like the ones we currently have, which of course would introduce selection bias into the dataset if the sample we got for the test set were in fact a biased sample. So in a sense, this was more so of a weird "bootstrap". Unfortunately, this didn't quite work. Having taken a Bayesian statistics course this semester (Spring 2021), it's pretty clear that in order to set justified priors, you must know something about the variables in your model. When I went to attempt this in February, at the beginning of the course, my approach was that the MLE of each parameter would account for our prior "beliefs". In a sense they would since, theoretically, the MLE is an unbiased estimator and thus would give a somewhat sensible estimate of the parameter values. I hoped that

this would somewhat replicate the process an actuary would go through by looking at more and more data over time, forming their "expert" prior beliefs.

In the end, there just wasn't enough information about the features to do this. The values from the posterior predictive distribution were just on the wrong scales for every feature. I asked my Bayesian professor about this, and he said that it was likely due to my priors being too constrained. But to recap, I tried using this model:

$$\boldsymbol{y_i} \sim N(\boldsymbol{\mu}^{(\boldsymbol{i})}_{\boldsymbol{MLE}}, \boldsymbol{\Sigma})$$

Note that $\boldsymbol{\mu}^{(\boldsymbol{i})}_{\boldsymbol{MLE}}$ is a vector of the MLE estimates for each feature for the i^{th} sample. While $\boldsymbol{\Sigma}$ is the identity covariance matrix with a constant scaling factor, i.e., $\boldsymbol{\Sigma} = c \cdot I, c \in \mathbb{R}$.

With this failing to produce viable results, I went back to traditional machine learning methodsm, which I'll talk about in the next section. My Bayesian coursework will be uploaded on my **Github** at the end of the semester with PDF reports available on the "university" section of my **website**.

Methods

I started with some quick EDA to examine the correlations between the features we wanted to use:

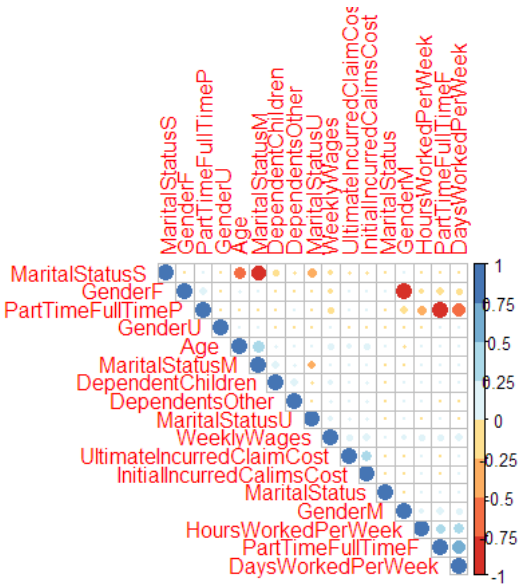

Figure 1: correlation plot

Clearly, we'll have an issue with multicollinearity given that certain features are clearly correlated with each other, and this is logical since by conventional knowledge can assume something like `MartialStatus` is correlated with `Age`. While multicollinearity isn't really a problem with prediction (in contrast to casual inference), we would still want to ideally have independent features because those would be advantageous in "saying" something about our target rather than many correlated features. This motivates us to find the most important features out of our subset to start our model building given our limited time (and adding more features to our model makes it harder to tune).

I then constructed a linear model with the cube root of `UltimateIncurredClaimCost` (our target variable) with previously mentioned features as predictors, i.e., we consider the model:

```
UltimateIncurredClaimCostˆ(1/3) ~ Age + Gender + MaritalStatus + DependentChildren +
DependentsOther + WeeklyWages + PartTimeFullTime +  HoursWorkedPerWeek + DaysWorkedPerWeek
+ InitialIncurredCalimsCost
```

Note that I used the cube root transformation as it's a fairly "soft" variance stabilizing transformation.

Then, I used R's stepwise AIC and BIC variable selection process, which resulted in this model:

```
UltimateIncurredClaimCostˆ(1/3)} ~ Age + Gender + MaritalStatus + WeeklyWages + PartTimeFullTime
+ DaysWorkedPerWeek + InitialIncurredCalimsCost
```

So using these features, I started by building out various boosted tree models, which don't run into issues with multicollinearity, are fast, and most importantly generally perform well with little tuning. Also since boosted trees are an ensemble of trees, it's easy to implement bagging later and other ensembling methods. In particular, we'll want to bag our boosted trees to help reduce variance, while keeping bias approximately stable. Recall that the bias-variance decomposition is as follows (for MSE, taken from my CSC311 notes):

$$\begin{aligned}\mathbb{E}(y-t)^2 &= \mathbb{E}[(y-\hat{y})^2] + Var(t) \\ &= (\hat{y} - \mathbb{E}[y])^2 + Var(y) + Var(t)\end{aligned}$$

Note that above, t are the targets (the actual data), y is our sample, and $\hat{y}$ is our prediction.

In the interest of time, I decided on `CatBoost` (found **here**) as the boosting tree. It is an extremely powerful library with great features like built in grid-search for hyperparameter tuning, which was extremely useful. It is also has robust categorical feature handling which probably helped it perform the best out of all of our tree algorithms, and many of our 7 features were categorical. At first, I used a 22% validation set, which gave us 12000 rows for validation. The model (with RMSE as the metric) performed approximately 23,800 dollars on the validation set - quite good! So then, I explored other boosted tree libraries such as `XGBoost` and `LightGBM`. `XGBoost` just wasn't quite as good at `CatBoost` even when used bagging, so we ended up only using `LightGBM` and `CatBoost`, and it turned out `LightGBM's` random forest option was better than it's default option. So then, after having these base regressors, we fit a bagged regressor to each boosted tree model, that is:

$$\hat{y_i} = \sum_{j=1}^{m} \frac{f_j(\mathcal{D}_j)}{m}$$

Where in the above equation, we take $\hat{y}_i$ to be the prediction for the i^{th} sample, we take $\mathcal{D}$ to be a bootstrap dataset, and we take f_j to be the j^{th} base model/regressor.

Since I treated this as a way to test out various models on actuarial data, I also wanted to test out the (in)famous neural network. So I used tensorflow and trained a custom neural network using an autoencoder architecture. I believed that despite only have 7 features, this was still rather high-dimensional, so an autoencoder would help by only using the most important features from the data by projecting onto the 100 dimensional bottleneck layer. Here's the architecture diagram:

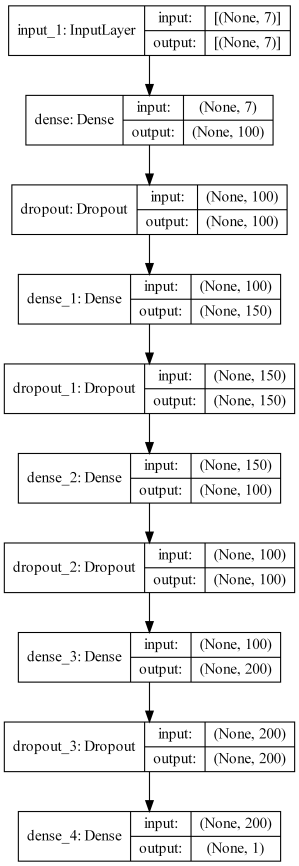

Figure 2: neural network architecture

Finally, then to combine the power of the 3 models, I used them in a stacking ensemble; using the predictions from the 3 individual regressors to make final predictions. I used `sklearn's` RidgeCV stacked regressor to do this, which includes $L2$ regularization along with k-fold cross validation, which is fantastic. This process got me a result of approximately 70 out of 130 teams on the public leader board. Although, it is worth nothing that the best team to date has a score of 29725 and this process led me to a score of approximately 30200. The leaderboard is **here**.

Learning from others

A user named `aldparis` (profile **here**) submitted a notebook with a tutorial for using text data, date data along with better data transformation for neural networks. This was a great resource because neural network activation functions are fairly dependent on data being scaled properly and consistently (else you "say" certain features are more important than others).

In particular this seemed to remedy issues I had with the small feature subset I used. He had a fairly robust pipeline for dealing with text data, involving a lemmatizer, a count vectorizer, and a regressor. A lemmatizer breaks down words into their core component, e.g., "caring" becomes "care", and a count vectorizer just counts the occurrences. So from this process he was able to generate a score for the textual data, which I was pretty amazed by.

Additionally, he used the date data to adjust for inflation between the start date and the claim date. So essentially he was able to take into account all of the information we were given.

I ended up incorporating his process and that score was approximately 30000, which was just far better than what I could do with my previous process. This suggests that my old process did about as best as I could,

and the models had a lot of bias due to leaving out so much information. I'm not surprised because adjusting for inflation is definitely a no-brainer while I knew leaving the text data out was leaving some predictive performance on the table!

Takeaways

The process I had was fairly robust and "algorithmic". It helped me figure out very easily where to go next; this process was essentially the one I used at my first year internship. Boosted trees are extremely good for how much time and effort they take to train tune, and creating ensembles of these trees, regardless of algorithm are quite good at improving model performance. But, they are quite bad at performing with data in out of sample ranges.

While neural networks work, tuning them can drastically improve performance, and indeed deep neural networks improve performance, even though by the universal approximation theorem, NNs can approximate any function with only one hidden layer of dimension $n + 1$, where n is the input layer dimension. The autoencoder architecture worked quite well even though our data wasn't particularly high dimensional, and new activation functions like `swish` are quite a bit better than even `leaky-relu`.

Finally, actuarial descriptions of claims do make a difference and handling them in a manner that `aldparis` did is sufficient to make large leaps in performance. Also, adjusting for inflation is low hanging fruit and should be definitely done. This is definitely a case of needed domain knowledge and having experience with data types that would occur in that domain.

Previously, I've had professional experience with photographic/auditory data. But if I were to want to do something with actuarial data, I'll want to read up on some papers on actuarial predictions and the like.

Limitations for STA303 professional development

I had hoped to put the code and PDF on my Github. Due to school commitments and because I use my Github as part of my recruitment strategy, I don't quite feel ready to put it up there. I would want to comment my code and clean up some of it to bring it up to my standards.

I do feel comfortable putting this write up on my Github, but I'll more likely just put it up on the "university" section of my website under STA303 for the time being until I clean up my code and upload it together as a repo on Github.

Regardless I think this was certainly a good professional growth experience that I can use as a talking piece with recruiters.