Machine Learning Engineer Nanodegree

Capstone Project

Roshan Ram

November 17th, 2019

I. Definition

Project Overview

I chose to work on a topic in which machine learning plays an undeniably large and immensely crucial role: insurance claims. The challenge in question actually comes from Kaggle, a popular online community of data scientists and machine learners, who allow many companies to host competitions, often for monetary prizes. The competition I chose to work towards is Allstate Severity Claims, which is described by Kaggle as such:

"When you've been devastated by a serious car accident, your focus is on the things that matter the most: family, friends, and other loved ones. Pushing paper with your insurance agent is the last place you want your time or mental energy spent. This is why Allstate, a personal insurer in the United States, is continually seeking fresh ideas to improve their claims service for the over 16 million households they protect.

Allstate is currently developing automated methods of predicting the cost, and hence severity, of claims. In this recruitment challenge, Kagglers are invited to show off their creativity and flex their technical chops by creating an algorithm which accurately predicts claims severity. Aspiring competitors will demonstrate insight into better ways to predict claims severity for the chance to be part of Allstate's efforts to ensure a worry-free customer experience."

In short, Allstate, at the time this competition was posted, had been aiming to improve their insurance claims system, and distributed a dataset of accidents happened to households (each is represented as a row of anonymized features belonging to this household) with a numerical metric of the cost of this claim. Our task is to predict how severe the claim will (or might) be for a new household.

Problem Statement

- Our goal: to produce a model that correctly predicts the target, loss based on the given features.
- This is a regression problem, since our target variable is numerical.
- This is a supervised learning task, since our target variable is in clear terms, defined in the training set.

Allstate already preprocessed a good portion of the data, so not a lot could be done in the way of data cleansing itself. The following steps were taken to go about this project:

1. Perform EDA on the given dataset, including understanding of different relationships/correlations, features, and target variables
2. Do necessary data preprocessing and train several different ML algorithms (XGBoost), and get a baseline score: seen in the XGBoost notebook.
3. Tune each model to obtain marked score changes.
4. Summarize the results aptly and evaluate the final score.
5. Brainstorm methods of improvement

Some complications that occured in the coding process included:

- Runtime of XGBoost
    - In an attempt to reduce runtime, I made use of Amazon's AWS EC2 instances to increase computing power, which didn't quite work out particularly well due to some setup issues, but I was able to obtain results *eventually*
- Number of parameters in GridsearchCV
    - The sheer volume of the parameters used in the Gridsearch implementation of XGBoost to find the optimal subset incurred quite some power from my machine and thus might not have been all too efficient
    - A possible alternative to this might have been to utilize a bayesian optimization technique such as hyperopt, employing a combination of both random search and grid search until the computational budget has been exceeded, as suggested by one of my project reviewers

Metrics

Mean absolute error is a straightforward metric that directly compares the predicted value with the truth. MAE metric is immutable--it is the Kaggle standard, in this case. This is, however, a non-problem since MAE is easy to interpret, among a few other pros: mean absolute error does not too heavily punish outliers, which is good in our case. Generally speaking, MAE is a good metric for data science novices. It is easily calculated, simple to understand and hard to be misinterpreted.

II. Analysis

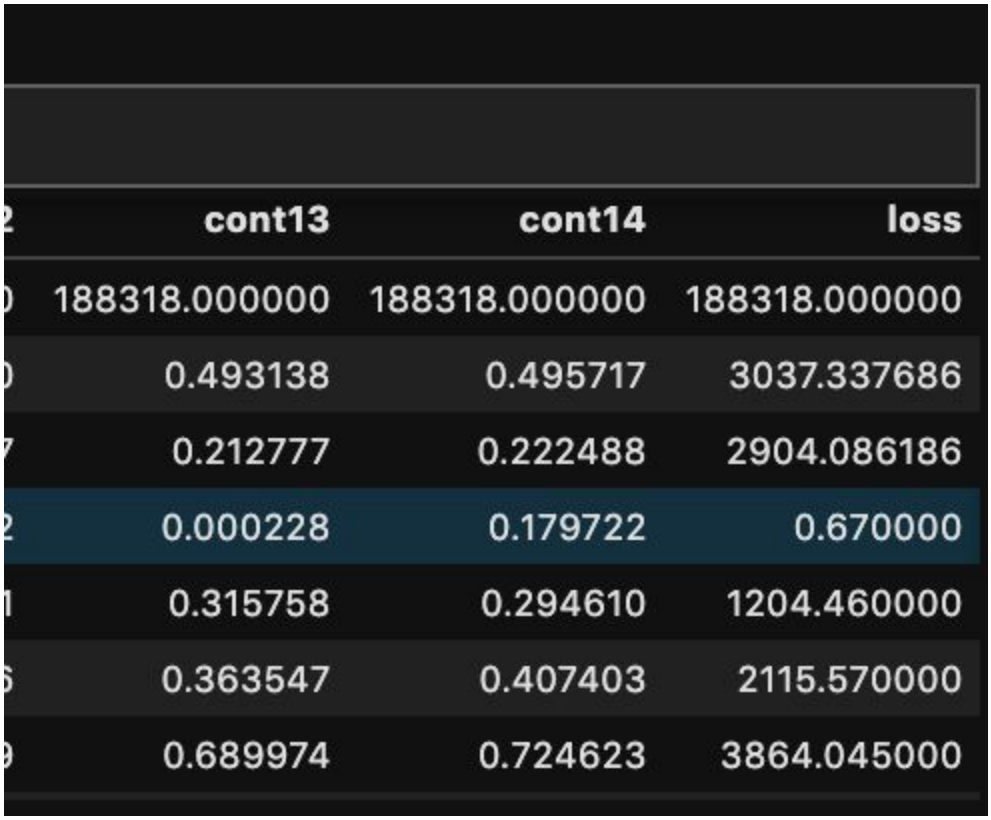
Data Exploration/Visualization

The entirety of the dataset consists of 188318 samples, each index denoting their *ID*.

Data exploration summary:

```
train.describe()
```

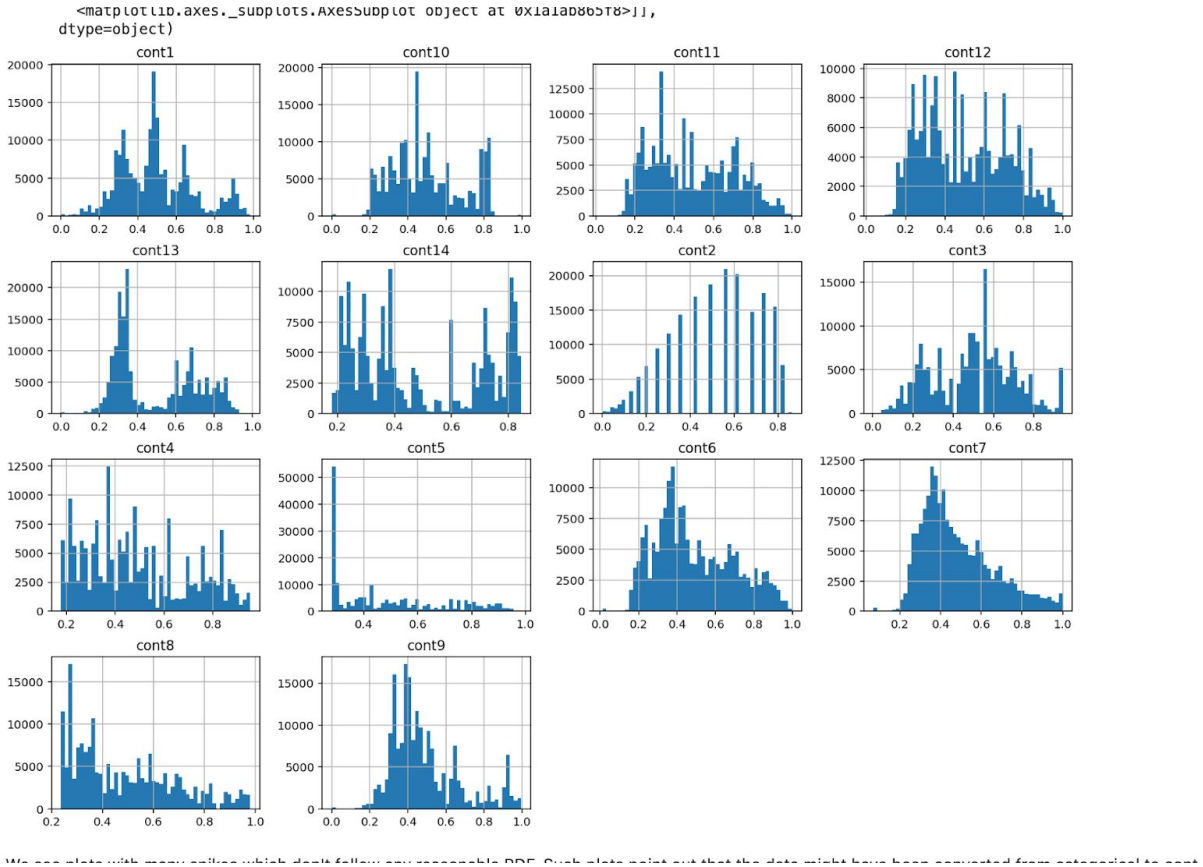
	id	cont1	cont2	cont3	cont4	cont5	cont6	cont7	cont8	cont9	cont10	cont11	cont12	cont13
count	188318.000000	188318.000000	188318.000000	188318.000000	188318.000000	188318.000000	188318.000000	188318.000000	188318.000000	188318.000000	188318.000000	188318.000000	188318.000000	188318.000000
mean	294135.982561	0.493861	0.507188	0.498918	0.491812	0.487428	0.490945	0.484970	0.486437	0.485506	0.498066	0.493511	0.493150	0.493138
std	169336.084867	0.187640	0.207202	0.202105	0.211292	0.209027	0.205273	0.178450	0.199370	0.181660	0.185877	0.209737	0.209427	0.212777
min	1.000000	0.000016	0.001149	0.002634	0.176921	0.281143	0.012683	0.069503	0.236880	0.000080	0.000000	0.035321	0.036232	0.000228
25%	147748.250000	0.346090	0.358319	0.336963	0.327354	0.281143	0.336105	0.350175	0.312800	0.358970	0.364580	0.310961	0.311661	0.315758
50%	294539.500000	0.475784	0.555782	0.527991	0.452887	0.422268	0.440945	0.438285	0.441060	0.441450	0.461190	0.457203	0.462286	0.363547
75%	440680.500000	0.623912	0.681761	0.634224	0.652072	0.643315	0.655021	0.591045	0.623580	0.566820	0.614590	0.678924	0.675759	0.689974



2	cont13	cont14	loss
0	188318.000000	188318.000000	188318.000000
0	0.493138	0.495717	3037.337686
7	0.212777	0.222488	2904.086186
2	0.000228	0.179722	0.670000
1	0.315758	0.294610	1204.460000
6	0.363547	0.407403	2115.570000
9	0.689974	0.724623	3864.045000

- There are ~130 distinct features (“~” because we are not counting id and loss), so dimensionality is not much of a problem. 116 of these are categorical, while 14 are numerical.
- We may have to encode these 116 features since most ML algorithms cannot correctly process categorical features. The ways of encoding and the difference among them will be discussed later on.
- There are **no** missing values in the whole dataset, which makes the data very user-friendly.
- Target feature is not normally distributed either, though it can be easily log-transformed to achieve a distribution close to Gaussian.
- Target feature contains a few extreme outliers, which must be carefully worked around, since they are important but risky if not dealt with properly
- Train and test set have a similar data distribution. This is an ideal characteric of train-test split which greatly simplifies cross-validation and allows us make informed decisions about the quality of models using cross-validation on training set. It greatly simplified my Kaggle submissions, but it won't be useful in Capstone.

- Several continuous features are highly correlated (the correlation matrix is plotted on Figure 1 below). This leads to data-based multicollinearity in dataset which can drastically reduce the predictive power of linear regression models. L1 or L2 regularization can help reduce this problem.
- We can note some key values such as the mean, standard deviation, minimum, and interquartile breakdowns (25%, 50%, 75%) from the above pictures.



We see plots with many spikes which don't follow any reasonable PDF. Such plots point out that the data might have been converted from categorical to cont

Algorithms and Techniques

XGBoost. One of my motivations was to test boosted trees approach and specifically XGBoost. This algorithm became a de-facto standard swiss knife for many Kaggle competitions due to its scalability, flexibility and an amazing predictive power.

Generally speaking, XGBoost is a variation of boosting, a machine learning ensemble meta-algorithm for reducing bias and variance in supervised learning, and a family of machine learning algorithms which convert weak learners to strong ones.

System-wise, the library's portability and flexibility allow the use of a wide variety of computing environments like parallelization for tree construction across several CPU cores; distributed computing for large models; Out-of-Core computing; and Cache Optimization to improve hardware usage and efficiency.

The algorithm was developed to efficiently reduce computing time and allocate an optimal usage of memory resources. Important features of implementation include handling of missing values (Sparse Aware), Block Structure to support parallelization in tree construction and the ability to fit and boost on new data added to a trained model (Continued Training).
["https://www.kdnuggets.com/2017/10/xgboost-top-machine-learning-method-kaggle-explained.html"]

The idea behind boosting is building weak trees, or "learners" in a nested sequence. Each subsequent weak tree tries to reduce the bias of the whole combined estimator, thus combining weak learners into a powerful ensemble model. There are various examples of boosting algorithms and techniques, including AdaBoost (adaptive boosting which adapts to the weak learners), LPBoost and gradient boosting.

Specifically, XGBoost is a library which provides the gradient boosting framework. Gradient boosting model is built in a stage-wise fashion like other boosting methods do. This boosting technique generalizes weak learners by allowing optimization of an arbitrary differentiable loss function (the loss function with a calculable gradient).

Benchmark

- The first benchmark is the one that was set by Allstate: they trained a Random Forest ensemble model and got the score MAE = 1217.52, which is not too hard to beat.
- There are benchmarks I set when I trained models. My benchmark is a performance of a simple model of its class. For XGBoost, a benchmark is set to MAE=1230.34--the predicted loss from a 50-tree model with no hyper-parameter tuning.

III. Methodology

Data Preprocessing

Our target feature is exponentially distributed which can lower the score of our regression models. As we know, regression models work best when the target feature is normally distributed.

To solve this problem, we can simply log-transform loss: using *np.log(train['loss'])*

This can be improved even more: there are several outliers to the left of the bell curve. To remove them, we can shift all values from loss+200 and take the logarithm afterwards.

One-hot encoding

we have to convert our categorical features to numbers for XGBoost, so I used one-hot encoding. One-hot encoding is a default way to deal with categorical features. It outputs a sparse matrix where each new column represents one possible value of one feature. This results in a longer training time, with more memory requirements.

Implementation

- Train, tune and cross-validate our XGBoost model:

Steps taken:

1. Train a shallow model, obtain a baseline score.

2. To facilitate hyper-parameters tuning, we utilize GridSearchCV from the sklearn.model_selection module to find the optimal XGBoostRegressor algorithm parameters. This seems to take a very large chunk of time.
3. We define and fix the learning rate and the number of estimators to be used in the next grid search. Our priority is to keep the training time and the running time of grid search at minimum. Eta of 0.1, with 50 trees.
4. Tuning max_depth and min_child_weight. We get the following best results: max_depth of 8 and min_child_weight of 6. We have improved our score.
5. Tune gamma, one of the regularization parameters
6. Tuning the ratio of features and training examples to be used for each tree: `colsample_bytree`, `subsample`. We find an optimum: `subsample=0.9`, `colsample_bytree=0.6` and our score is improved to `MAE=1183.7`.
7. Finally, we add more estimators by increasing the number of trees and decrease the learning rate, or eta. Our final model uses 200 trees, has eta=0.07 and a final score of 11

IV. Results

Model Evaluation and Validation

In the final step, to assist in the evaluation of our results, I trained and validated our final XGBoost model on different subsets of the data. This helps us validate how stable the model is and whether it is reliant on the basis of the primary training set.

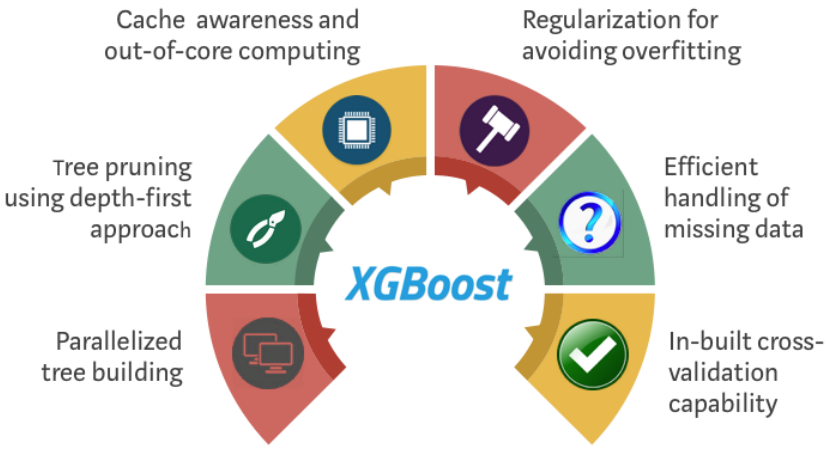
Measurement metric	**XGBoost values**
mean	1143.589538
std	2.325297
min	1140.778078

25%	1141.327482
50%	1145.197159
75%	1145.232551
max	1145.412418

We observe that the MAE values are indeed stable enough.

Justification

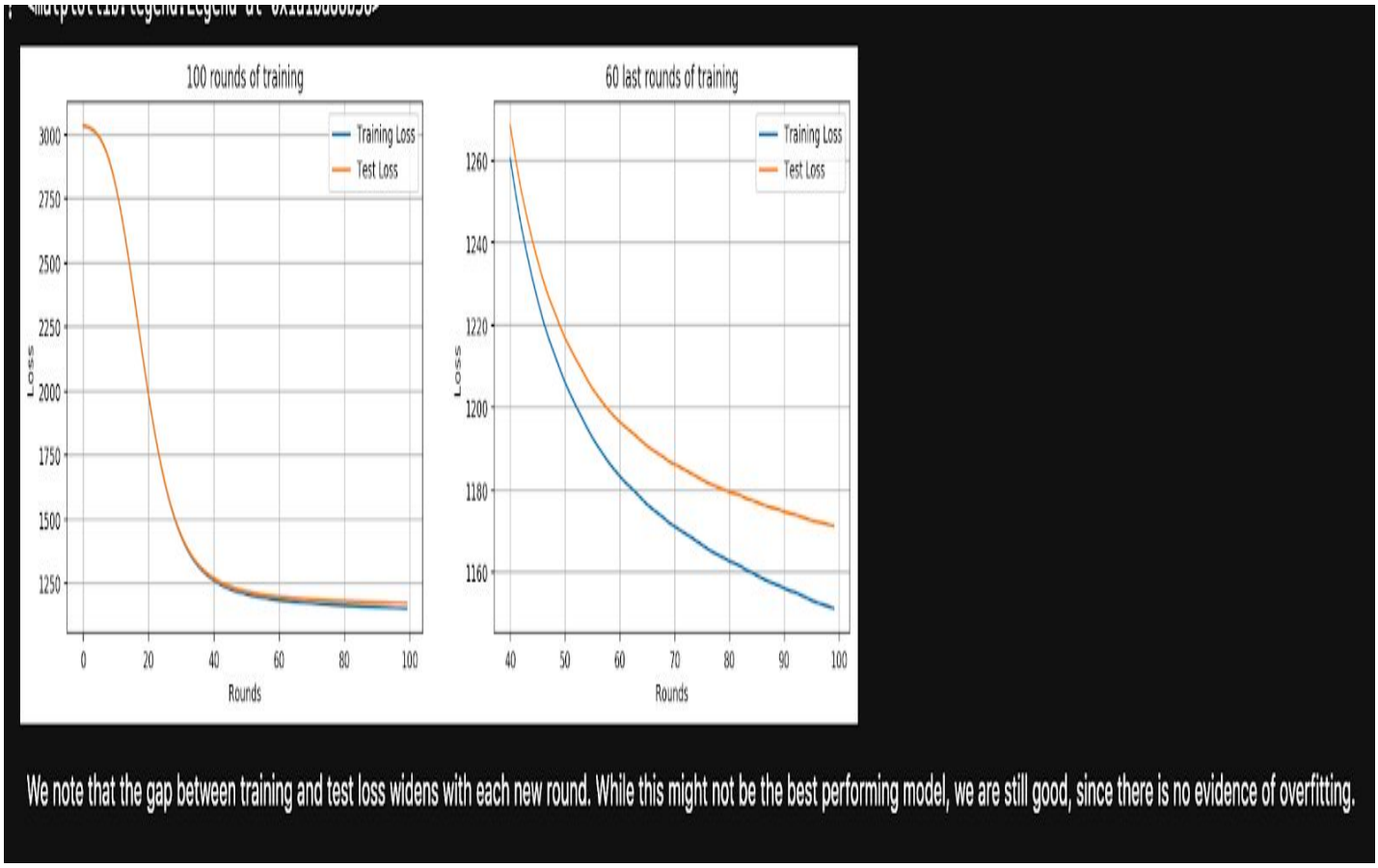
This solution is sufficient enough to consider the problem “solved” since our MAE is around ~1000. Comparing my MAE to the benchmark MAE, we see that we have made a notable improvement with regard to Kaggle’s baseline Random Forest model--our MAE is higher by about 5.8%. Thus, we can be sure that our XGBoost model has “succeeded” is some way.



Success!
[https://towardsdatascience.com/https-medium-com-vishalmorde-xgboost-algorithm-long-she-may-rein-edd9f99be63d]

V. Conclusion

Free-Form Visualization



We note the relationship between epochs/rounds of training and loss. This is an important visualization because it allows us to visualize and decide whether we must stop the training here itself or proceed with no fear. This is pivotal since valuable machine CPU/GPU computing power hours could be wasted on a model that ends up memorizing the data, or overfitting. Overfitted models can work on testing sets, but they are dangerously unstable and unreliable.

Reflection

The idea of this project was to work with a dataset where feature engineering is not exactly pivotal. This allows us to focus more on optimization of essential algorithms rather than get bogged down in lower-level data cleaning--which, at the risk of sounding ignorant, is still important, but not as difficult to algorithmic construction itself.

Improvement

1. Could have fit a more complex XGBoost model by adding even MORE trees and playing around with the learning rate (eta). In the current implementation, I simply used GridSearchCV to
2. Could also use early stopping to prevent the model from overfitting and also save some time. However, this is a tradeoff, since, as a result, the model will receive less data to train on may thus suffer from a marginally decreased performance.
3. Finally, instead of using 3-5 fold “k-fold” cross validation, we could experiment with the idea of increasing the number of folds to try to train on “more” data.