Capstone Project

Machine Learning Engineer Nanodegree

Fujiki Nakamura

Definition

Project Overview

In this project, I solve a problem provided by Kaggle. Kaggle is a Data Science competition platform and provide their competitors with a lot of data in real world. Many companies are working with Kaggle and sharing their data on the platform of Kaggle to discover the methods that produce the best result and the talented competitors. Competitors at Kaggle make the most of their Data Analysis and Machine Learning skills and aim to win their competitions. Kaggle is also a platform of learning practical skills of Data Science and helps many junior Data Scientists advance their skills providing discussions on the forum and open scripts.

The problem I solve in this project is provided by AllState corporation, which is the second largest personal insurer in the United States. They provide their data including the information of how severe an insurance claim is and other information that can be useful to predict the severity. According to the Kaggle home page of this competition, they are "currently developing automated methods of predicting the cost, and hence severity, of claims" and looking for the talented competitors to recruit.

The objective for this project is not recruitment, but through this problem I provide a case study of developing automated methods to produce good predictions.

Problem Statement

This is a problem of regression because the target value (, which is the severity here) is numerical and a problem of supervised learning because the target value is explicitly provided in the training dataset and we have to predict the scores for the test dataset. In such a kind of problem, I need to build the model which predicts the cost of claims that Allstate's customers have to pay as correctly as possible. In order to achieve that goal, I follow the strategy outlined below:

1. Data Exploration:
   explore the data and grasp how they look like. In this process, I examine the meanings and distributions of the data.
2. Data Preprocessing:
   preprocess the data in a way the each model built later can handle them appropriately. Convert some type of values into another type of values to feed them into the models appropriately, transform some values to correct their skewness in distributions and remove some unnecessary values.
3. Building Machine Learning models:
   Develop several Machine Learning models to approach the solution. Build the baseline models and define their performance as the criteria of improvement later.
4. Tuning the models:
   enhance the performance of the models built in step 3. Tune the hyper parameters of the models, make the better architecture of the

models and so on.
5. Ensembling the models:
   To achieve the better performance than the single models above, ensemble them by introducing the method of stacking.

Metrics

The metric to measure the performances is Mean Absolute Error (MAE). In addition to the fact that the competition requires MAE as the metric, MAE is appropriate because it measures the differences between predicted values and ground truth values directly. Of course, other metrics like MSE or RMSE seem appropriate but they have a "side effect" which we don't want here. When they square the differences, they make the large differences even larger and the smaller differences even smaller (More precisely, when squared, the errors (i.e. the diffences) more than 1 have larger impact on the total error and the errors less than 1 have smaller impact on the total error). MAE is one of the appropriate metrics to measure how much the differences are between the monetary values and to handle all errors equally, and interpreting the total difference is intuitively easy (i.e. How much differs from the true cost as a whole).

Analysis

Data Exploration

The dataset consists of `188,318` training data and `125,546` test data. The training data has `132` features and the test data has `131` features. The difference is a feature of `loss`, which means the cost of the insurance claims. The training data has it but the test data does not. The feature `loss` is our target variable and our predictions have to be as close as possible to that value. The mean of the `loss` is `3037.34` and it turns out that the average cost is `$3037.34`. The standard deviation is `2904.09`, the minimum is `0.67`, the maximum is `121012.2`, and the median is `2115.57`.

Other feature, which the models don't need in their training process is `id`. As its name suggests, it identifies the each row of the each dataset. We have `188,318` `id`s in training data and `125,546` in test data.

The main features of our dataset are the continuous features and the categorical features. In our Machine Learning step, we feed these two types of features into our models. We have `116` categorical features and `14` continuous features. However, we don't know the meanings of these features. Allstate prepared the dataset really well and all the features we need to build our models are anonymized. This is the characteristic of this problem. We have to achieve the best performance without knowledge of the dataset and hence without the human intuitions.

The categorical features are named from `cat1` to `cat116`, and they have at least two values represented in the uppercase alphabet. For example, `cat1` has two unique values, 'A' and 'B'. The largest part of the categorical features are binaries and in fact the categorical features from `cat1` to `cat72` have two unique values of 'A' and 'B'. Other categorical features from `cat73` to `cat76` have three unique values 'A', 'B' and 'C'. And as expected, there are also some categorical features that have more than three unique values: from `cat77` to `cat88` have 4 unique values and from `cat89` to `cat108`, `cat111`, `cat114` and `cat115` have 5 unique values or more. Almost all the categorical features consist of the single letter values as mentioned above, but some categorical features consist of 2-letter values. The features from `cat109` to `cat116` (except `cat111`, `cat114` and `cat115`) have not only single letter values but also 2-

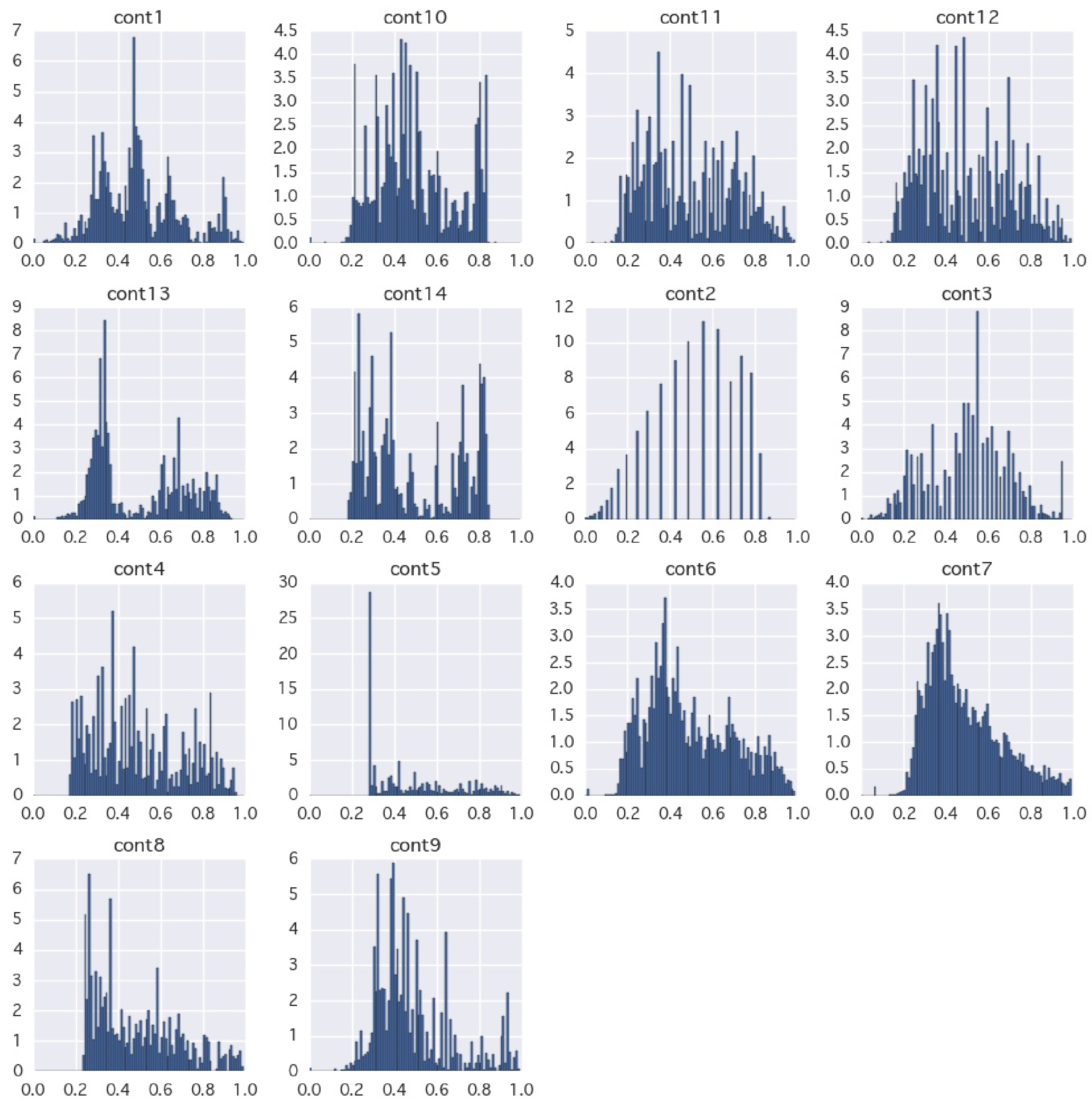
letter values like 'AA', 'AB' and 'BY'. It might be noted that 2-letter values in '*Z' form (e.g. `DZ`) rarely appear, and actually `cat109` barely has 'ZZ' in its values.

Summary of the categorical features

name	Description	Example
cat1 to cat72	Binary categorical feature	A, B
cat73 to cat76	3 values	A, B, C
cat77 to cat87	4 values	A, B, C, D
cat88	4 values without C	A, B, D, E
cat89 to cat108, cat111, cat114, cat115	5 or more values	A, B, C, D, E, etc
cat109, cat110, cat112, cat113, cat116	2-letter values also appear	A, B, AA, BA, CI, etc

As I mentioned above, we have 14 continuous features. All the continuous features are scaled between 0 and 1. Checking the distributions of these continuous features, it is found that the distributions of the train data are similar to those of the test data. In other word, the corresponding continuous features in those two dataset have almost the same distributions respectively. Therefore, it seems that we don't have to concern about the problem resulting from the distribution differences between the train data and the test data. The distribution of the each continuous feature in the training data vary and is different from each other. Some of them have distributions like the gaussian (for example, `cont3`). Some of them have skewed distributions: the features like `cont6` and `cont7` have the right skewed distributions. In `cont5`, the skewness of the distribution is really high and there seem to be a lot of small values in `cont5` while there are few of the larger values. The distributions of the countinuous features are visualized below.

Exploratory Visualization



Above are the histograms of 14 continuous features from `cont1` to `cont14`. For each histogram, the x-axis are the values that each continuous feature can take and y-axis is the frequency. This visualization shows us the characteristics of the continuous features. They are all well preprocessed and anonymized, ranging between 0 and 1. Some features like `cont3` has the distribution like the gaussian. On the other hand, some features like `cont6` has the right skewed distribution.

The `cont2` seems to have an interesting distribution. It has some kind of beautiful distribution compared to the other continuous features, but the values it can take are discretely distributed. It is said that the `cont2` might be originally an categorical feature and converted to a continuous one. In another experiment, considering the special characteristics of the continuous features like this might be helpful. We don't consider such characteristics here.

Algorithms and Techniques

I chose two algorithms for this problem, Gradient Boosting Decision Tree and Neural Network.

Gradient Boosting Decision Tree

The package used is Extreme Gradient Boosting, XGBoost for short. XGBoost is a decision tree based algorithm and ensembles many trees to get better performance as a whole. So in terms of model (i.e. tree ensembles), it is the same as Random Forest. The difference is the way it learns. While Random Forest learns in a way that it makes trees that are different from each other grow in parallel and aggregates the result of each tree, XGBoost learns by making trees grow in sequence. When XGBoost learns, each tree learns what the previous trees didn't learn. In other word, the next tree learns and tries to minimize the error that the previous trees left. For example, the 3rd tree learns and minimizes the error that 1st and 2nd trees left. Sequentially ensembling trees, XGBoost gains better performance as a whole (For more detailed explanation of XGBoost, we can refer to Introduction to Boosted Trees.)

Recently, XGBoost is popular among the competitors. It became widely known through the competition of Higgs Boson Machine Learning Challenge, where the 1st place winner used XGBoost. Its popularity and competence made me choose the algorithm to solve the problem here in addition to my interest in solving problems with XGBoost. I admit that there is no best algorithms for solving any kind of Machine Learning problem, but its popularity shows that XGBoost is really competent and we can take the advantage of it at almost any time as one of the Machine Learning algorithms.

Neural Network

The other algorithm for this problem is Neural Network. Neural Network here is a kind of Multi-Layer Perceptron and it has multiple fully connected layers in its architecture. Neural Network is recently famous with the deep version of it (i.e. Deep Neural Network, or Deep Learning). Neural Network is said to work as any kind of functions approximately, so it is expected that it is also powerful to solve the problem here.

Although it seems powerful, Neural Network is prone to overfit and not to generalize well to unseen data. We have to pay attention to that fact. Fortunately, we have several techniques to avoid overfitting: L1 or L2 regularization, Dropout and Batch Normalization. Among them, Batch Normalization is the newest technique and used in this problem later. For more information about it, we can refer to the paper of Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.

The final step to solve the problem is ensembling model by stacking them. To have better performance by stacking, it is said to be good to have different kind of algorithms in the same level. Neural Network works in a very different way than XGBoost, so it is appropriate also in terms of stacking.

The package used for Neural Network models here are Tensorflow and Keras. "Tensorflow is an open source software library for numerical computation using data flow graphs" as mentioned in its documentation.

On the other hand, Keras "is a high-level neural networks library, written in Python and capable of running on top of either TensorFlow or Theano. It was developed with a focus on enabling fast experimentation. Being able to go from idea to result with the least possible delay is key to doing good research.", which is cited from its documentation. The Neural Network models were built using Keras with Tensorflow backend.

Stacking

Stacking is a technique to ensemble multiple Machine Learning algorithms. In stacking, we select some algorithms as 1st level models and make predictions with each of them. Then we use the 1st level predictions as the input for the 2nd level models. Thus we ensemble the previous level models by making the current level models learn with the result from the previous level models. The level can be more than two. So we can have 3rd or more level models using various algorithms. The method of stacking is also mentioned in this competition forum, and its concept and the way it works is well outlined in Stacking understanding. Python package for stacking. For more detailed explanation of the concept and the methodology, we can refer to the paper of Stacked Generalization.

Ensembling by stacking might not often be used the outside of competitions, but in terms of producing more generalized results (hence more better predictions), it seems appropriate in this problem. Stacking seems almost always useful under circumstances where we have to get as accurate results as possible.

Benchmark

We have three benchmarks here: Random Forest benchmark, XGBoost benchmark and Neural Network benchmark.

Random Forest benchmark

We have a benchmark from Random Forest which Allstate might have trained before the competition. The Random Forest Benchmark scored the `MAE = 1217.52141` on the Public LeaderBoard. This benchmark is not so good, and some XGBoost models which some participants had developed had already beaten the score when I joined the competition.

XGBoost benchmark

The XGBoost Benchmark is `MAE = 1114.16` on the Public LeaderBoard. The XGBoost model that produced the score is suggested by Vladimir Iglovikov on his kernel of xgb 1114. As shown in the kernel, the hyper parameters of the benchmark model of XGBoost are `min_child_weight = 1`, `max_depth = 12`, `eta = 0.01`, `colsample_bytree = 0.5`, `subsample = 0.8`, `alpha = 1` and `gamma = 1`. Also, this model trained with the target value `loss` transformed into `log(loss + 200)`. This transformation is mentioned later. While this model scored `MAE = 1114.16` on the Public LeaderBoard, the model scored `mean MAE = 1134.77` on 5-Fold Cross Validation with the train data. We start from this XGBoost model and are going to enhance the predictivity of XGBoost model.

Neural Network benchmark

The Neural Network benchmark is mean `MAE = 1184.39` on 5-Fold Cross Validation. The benchmark model is a simple 2-layer Neural Network. The First layer has 128 units followed by the ReLU activation and the second layer (, which is the output layer) has 1 unit. The detail of the model is shown in a script of `2_layer_v1/model.py`. The model trained with target value `loss` with no transformation.

Benchmark Scores (MAE)

Model	5-Fold Cross Validation	Public LeaderBoard	Description
Random			trained by Allstate

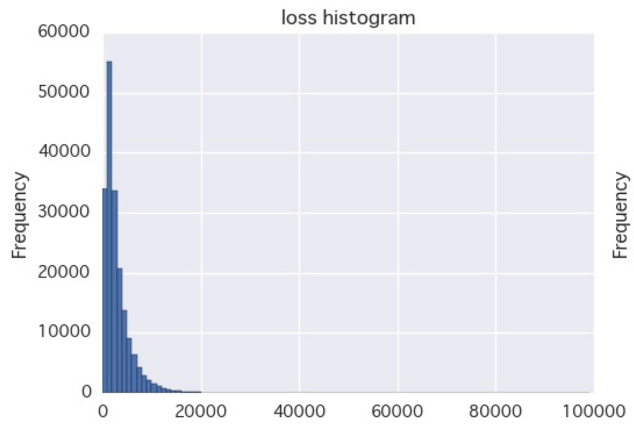
Forest	-	1217.52	
XGBoost	1134.77	1114.16	suggested by Vladimir Iglovikov
Neural Network	1184.39	not submitted	2-layer simple Neural Network

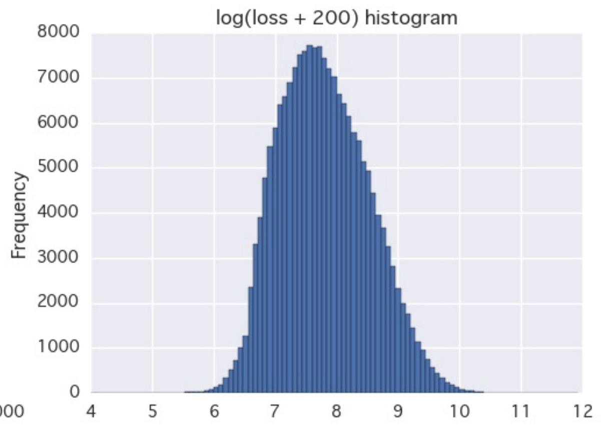
Methodology

Data Preprocessing

Allstate has already preprocessed the data well, so there is almost no room for us to preprocess more. However, we have 4 feature preprocesses done: Log(`loss` + 200) transformation, removing some values of categorical features, Label encoding of the categorical features for XGBoost models and One hot encoding of the categorical features for Neural Network models. All these preprocesses are done in a script `preprocess.py`.

Target value `loss` transformation
Originally, the target value `loss` has a right skewed distribution. Generally speaking, we have to have features normally distributed in regression tasks. So, we transform the `loss` into `log(loss + 200)`, which gives us a normally distributed `loss` values. This transformation is shown in the figure below: left is the original `loss` distribution and right is the `log(loss + 200)` distribution. However, the Neural Network models were not trained with this transformation and the models got the better scores without the transformation.





Removing some values of categorical features

Some categorical features have some values which appear only in training data or only in test data. We remove such values and consider them as missing values. It doesn't seem helpful for our models to train with the categorical values only in the training data because they can't take advantage of the knowledge of the values only in the training data when they predict with the test data. Also, it doesn't seem helpful that our models predict with the values that they don't see at all in their training process. Therefore, we remove the categorical values that exist only in the training data or the test data.

Label encoding of the categorical features

Generally speaking, Machine Learning algorithms don't work with categorical features without any preprocess on them. So, we have to preprocess categorical features. We have two ways to preprocess: Label encoding and one-hot encoding. Label encoding gives numerical labels to all classes in a categorical feature. Label encoded, `n` classes in a categorical feature is converted into the value between `0` and `n - 1`. We can keep the order of the classes in label encoding. To consider the order of the values in the categorical features, we preprocess the categorical features with label encoding for XGBoost. The order is a lexicographical one (For example, A, AA, AB, ..., B, BA, BB, ..., ZZ and not like A, B, ..., AA, AB, ..., ZZ).

One-hot encoding of the categorical features

On the other hand, for Neural Network models, we preprocess the categorical features with one-hot encoding. With this encoding, we get `n` dummy variables with a categorical feature which has `n` classes. In an experiment, it was found that our Neural Network models don't work well with the categorical features label encoded, but work better with one-hot encoding in this problem. Hence we preprocess the categorical features with one-hot encoding for Neural Network models.

Implementation

Algorithms

As we take advantage of the existing packages and libraries, we don't have complicate implementations as to the learning algorithms. We use XGBoost packages for Boosted Trees and Keras for Neural Network. Keras is really helpful and makes it easy to implement Neural Network models. The Neural Network models implemented with Keras are shown in `create_model()` functions in scripts named `model.py`. For example, the implementation of the 2-layer Neural Network model for the benchmark is shown in the `create_model()` function of `2_layer_v1/model.py`.

Metrics

As I mentioned earlier, the metrics for this problem is MAE. However, as we transform `loss` into `log(loss + 200)` for XGBoost models, we can't compute MAE directly with the predictions from the models. We need a reverse transformation. So, we have to exponentiate the values of predictions, and then we can compute MAE. The implementation of the metrics with the reverse transformation looks like below and is shown in `xgb/utils.py`.

```
def evalerror(preds, dtrain):
    labels = dtrain.get_label()
    return 'mae',
            mean_absolute_error(
                np.exp(preds)
              , np.exp(labels))
```

Other techniques

We implement a custom objective function for XGBoost models. It is what is called Fair objective function. `Fair objective function` plots a graph which looks like a graph by absolute value function. So, the functionality of the custom objective function is similar to MAE. However, it is different from MAE in that it is differentiable at any point (We can refer to this page about Fair objective function). The default objective function of XGBoost is `linear regression` and it minimizes MSE not MAE. As we don't have an appropriately objective function for minimizing MAE in the XGBoost package, we need to use `Fair objective function` as a custom objective function (and with this custom objective function, XGBoost models performed better, which is mentioned later). The implementation of `Fair objective function` looks like below and is also shown in `xgb/utils.py`.

```
def fair_objective(preds, dtrain):
    labels = dtrain.get_label()
    con = fair_obj_constant
    x = preds - labels
    grad = con * x / (np.abs(x) + con)
    hess = con**2 / (np.abs(x) + con)**2
    return grad, hess
```

The implementation of the custom function was some kind of complicate thing. The default objective function for XGBoost is `reg:linear` and it tries to optimize `MSE` not `MAE`. So, we needed other objective functions to train XGBoost models appropriately and got the `Fair objective function`. Applying this function to our XGBoost models might be complicate. We have to know how XGBoost optimizes its given loss function. It is documented in the introducing document. Also, we have to know how to implement the function appropriately in XGBoost. We have to get the first and second order derivative of the function, not the function itself, as shown above.

Refinement

XGBoost

Hyper parameter tuning

I tried several combinations of the hyper parameters, and found that the hyper parameters of the XGBoost benchmark are already tuned well. However, more improvement was done by tuning `min_child_weight`. The `min_child_weight` work as some kind of regularization on tree building. The document says "If the tree partition step results in a leaf node with the sum of instance weight less than min_child_weight, then the building process will give up further partitioning." The Large `min_child_weight` might cause under fitting because the building process ends earlier than the small one. In contrast, the small `min_child_weight` might cause overfitting because the building process continue more than the large one. In this problem, tuning `min_child_weight` to `100` improved the benchmark score on the cross validation. Perhaps `min_child_weight = 1` might be small and might cause overfitting. With `min_child_weight = 100`, the model might avoid overfitting and scored `MAE = 1133.56` on the cross validation. The score is slightly better (`-1.21`) than the benchmark (`MAE = 1134.77`). The hyper parameter tuning and cross validation can be done by changing parameters in `parameter.py` and executing `xgb_cv.py` in `xgb/xgb_cv/` directory.

One parameters combination that produced the better result is mentioned above, but more trials and errors were done in fact. The list of the parameters combinations in the trials and errors is documented in `xgb/xgb_cv/param_table.md`.

Custom objective function

As I mentioned earlier, we use a custom objective function called `Fair objective function`. With this objective function, the XGBoost model scored better MAE than the benchmark. The model's hyper parameter is `No.1` and mean `MAE = 1132.18` on the 5-Fold Cross Validation. This refined model improved the benchmark score by `-2.59`. As the model improved the score on the cross validation, I submitted this model's predictions for the test data and got `MAE = 1113.43` on the Public LeaderBoard, which slightly improved (`-0.73`) the benchmark score on the Public LeaderBoard.

Refinement history of the XGBoost models

Description	MAE on 5-fold Cross Validation	MAE on Public Leaderboard
XGBoost Benchmark	1134.77	1114.16
param No.1	1133.56	not submitted
Fair objective, param No.1	1132.18	1113.43

Neural Network

In order to create more predictive Neural Network models, we take a simple strategy of increasing the layers of the Network.

3-layer architecture

I added one more layer to the 2-layer Neural Network and create a 3-

layer Neural Network. The first layer of the Network has 128 units followed by ReLU activation, the second layer has 64 units followed by ReLU activation and the last layer is the output layer of single unit. The detail is shown in `3_layer_v1/model.py`.

The 3-layer Neural Network model performed better than the benchmark. It scored `MAE = 1163.72` on the 5-Fold Cross Validation, which improved the benchmark score by `-20.67`.

4-layer architecture

To enhance the predictive ability of the 3-layer Neural Network model, I added one more layer to it and created a 4-layer Neural Network model. The first layer has 128 units followed by ReLU activation, the second layer has 64 units followed by ReLU activation, the third layer has 32 units followed by ReLU activation and the last layer is single unit output layer. The detail is shown in `4_layer_v1/model.py`.

The 4-layer Neural Network model performed better than the 3-layer Neural Network model (hence performed better than the benchmark of course). It scored `MAE = 1161.83` on the cross validation, which means it improved the 3-layer model score by `-1.89`.

Refinement history of the Neural Network models

Description	MAE on 5-fold Cross Validation
2-layer, benchmark	1184.39
3-layer	1163.72
4-layer	1161.83

Stacking

In order to get even better results, we ensemble the models refined above by the method of stacking. 1st level models for stacking are the best XGBoost model (Fair objective function, the hyper parameter No.1), the 3-layer Neural Network and the 4-layer Neural Network. In addition to those models, I prepared one more XGBoost model for stacking. The new XGBoost model is almost the same as the best XGBoost model but it has a different random seed. The preparation for the stacking with this XGBoost model is done in `xgb/xgb_v2/stacking.py`. By having multiple XGBoost models, I expected to make the stacking process stronger.

For 2nd level model for stacking, I selected a simple model, that is Linear Regression. Using out-of-fold predictions of the 1st level models as features and `log(loss + 200)` as a target value, the 2nd level model scored `MAE = 1130.37` on the 5-Fold Cross Validation. It beat the best single model score on the cross validation (`MAE = 1132.18`), which resulted from the XGBoost model with Fair objective function and the hyper parameter No.1, and improved the score by `-1.81`.

The whole model ensembled by stacking is the final model for solving the problem. The stacking process above was done in `level_2_model/linear_regression.ipynb`.

Results

Model Evaluation and Validation

As we examine the parameters in the final model, we find that the intercept is `0.065` and the coefficient is `0.064` for the 3-layer Neural Network model, `0.052` for the 4-layer Neural Network model, `0.757` for the best XGBoost single model and `0.118` for the different random seed XGBoost model. The coefficients for the Neural Network models are relatively small and the coefficients for the XGBoost models are relatively large (especially for the best XGBoost single model). This seems reasonable in that the best XGBoost model scored the smallest MAE on the cross validations as a single model. The impact from the best single model on the final scores should be large in order to make the final scores closer to the ground truth scores. However, this doesn't mean that the Neural Network models are unnecessary in that they scored worse MAE than the XGBoost models. They helped the 2nd level model in stacking predict more accurately. In fact, the 2nd level Linear Regression model scored the smaller MAE than the best XGBoost single model.

Justification

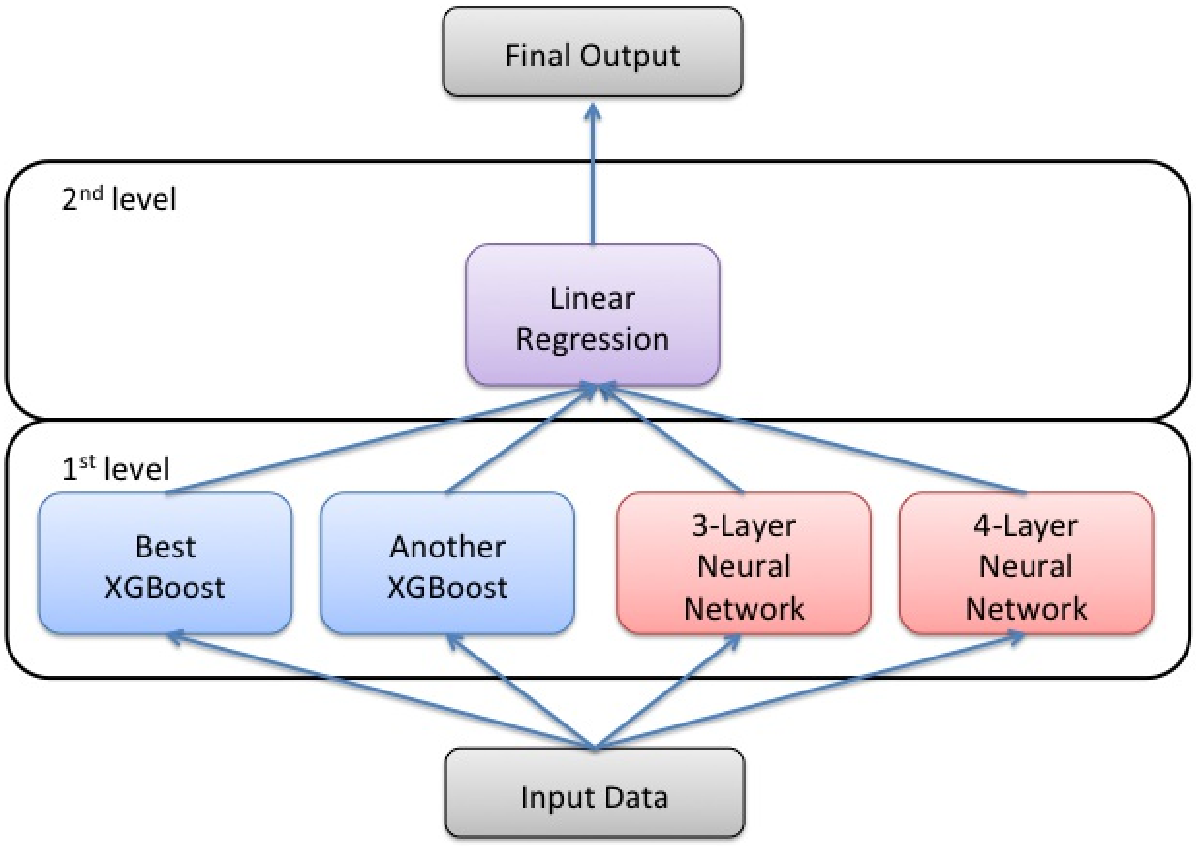
The result is that the final model performed better than any benchmark. It seems difficult to say that the final model performed significantly better in rigorous sense. However, the performance of the final model can be justified as we see the result history on the LeaderBoard. In addition to the fact that the final model performed better than the best benchmark model on the cross validation, it performed better on the Public LeaderBoard. Moreover, it got the better score on the Private LeaderBoard. In that the final model outperformed the benchmark model in three different testing environment, it can be said that the final model performed significantly better than the benchmark. And it seems that the final model is significant enough to solve this problem because the model performed well in all of the evaluating environment above. The final model got the better performance in the cross validation, and (as expected) also in the Public LeaderBoard. That is, the performance of the model doesn't seem a coincidence. In addition to that, finally, it got the better performance in the Private LeaderBoard. So, those facts convince us of the models' unchanged ability to solve the problem. So, I think the model is robust enough and it is likely to generalize well to unseen data.

Refinement history on LeaderBoard

Model	5-Fold Cross Validation	Public Leader Board	Private Leader Board
XGBoost benchmark	1134.77	1114.16	1128.57
Best XGBoost model	1132.18	1113.43	1128.26
Final model	1130.37	1111.96	1124.68

Conclusion

Free-Form Visualization



One of the important parts of this project is mainly about stacking. After we had several single models, we ensembled them by stacking. The 1st level models in stacking were those single models. We trained them on the dataset and obtained the 4 predictions. Then we trained the 2nd level model, which was Linear Regression, on top of them. The 2nd level model learned by taking the predictions from the previous level as its inputs. The prediction from 2nd level model was the final output. And as a whole, we obtained the more generalized result and hence got the more accurate predictions.

Reflection

Summarizing the entire end-to-end problem solution, we built some predictive single models and then ensembled them by stacking. Ensembling single models by stacking might not be so tough because the theory ensures more generalized result in higher levels. The difficult part of this problem for me might be to make powerful single models. I tried a lot of experiment on the hyper parameter tuning of XGBoost models, but hardly got good results. To be able to tune the hyper parameters well manually, we have to have good knowledge of the algorithms of the models. Or we could take advantage of some searching method like random search instead.

The final model fit my expectations for this problem and showed the performance better than the single models. When we definitely want to make powerful predictive models, solutions like the final model (i.e. model ensembling by stacking) must be helpful. However, I don't think that the final solution is necessarily perfect in all aspects for problems similar to this one. Sometimes we might not be able to have large and complicate models because of the limitations the on computational resources, the cost of the maintenance of the models and so on. Under such circumstances, we might want models less complicate and slightly less predictive but still predictive enough to solve our problems. This is a kind of trade off problems. The method of stacking seems powerful to solve problems on competitions (, in which we aim to make our models more predictive than other participants), but in general settings, especially in business settings, we might have to consider some trade off problems and make modifications to fit models practically in the settings.

Improvement

One aspect that could be improved is about feature engineering. Although there is almost no room for feature engineering, we might get better performance by handling some features more appropriately. One example is about `cont2`, which is a numerical feature but seems to be converted from a categorical feature (It is said to represent something like age groups). Because it seems to be originally a categorical feature, we could re-convert and consider it as a categorical one, which might lead to the better performances of the models.

Another aspect is about stacking. Although we have two different Machine Learning models above, we could have had more models in our 1st level. As we see in #1st Place Solution, competent kagglers have a lot of models in their 1st level and ensemble them. One algorithm which seems interesting for solving this problem is `LightGBM`. One of the competition forum post mentions the competence of LightGBM and it seems that LightGBM also works well on this problem (LightGBM LB 1112.XX). So, we could have used `LightGBM` and made our stacking process more powerful. Also about stacking, we could have had more levels. We have 2 levels in the stacking process, but by using the same method as the 1st level on the 2nd level we can create the 3rd level. If we have more levels in stacking, we might get more generalized models and the better predictions.