Machine Learning Engineer Nanodegree

Capstone Project

Patrick Park

December 6, 2017

I. Definition

Project Overview

In December of 2016, Allstate created a recruitment challenge on Kaggle: https://www.kaggle.com/c/allstate-claims-severity.

Kaggle is a website for people to learn machine learning by doing, by applying what they have learned about machine learning, and to learn new approaches from others. The goal is to use the data given in a competition to construct models that can be used to make predictions given new data from the problem domain. The data is provided by companies and organizations for various reasons such as solving an actual business problem or finding potential candidates for data science jobs.

In this competition Kaggle members were challenged to predict the severity of a claim – that is, predict the claim loss value – given certain information about that claim. Knowing the ultimate loss value for all open claims is important for an insurance company as they are required to set aside (reserve) a certain amount of the expected loss. The better they are at making this estimate, the more accurate they can be in setting these reserves.

Problem Statement

The challenge is to come up with a model that can be used to predict the severity of a claim based on a given set of categorical and numeric features of that claim. The severity of a claim is measured by the ultimate claim loss value in USD for that claim.

The challenge includes training and test data. The loss values are given for the claims in the training data, so this is a supervised learning scenario with the loss of the claim being the “target” value, the thing we want to be able to predict based on the other features of the claim. The goal is to create a model that can predict the loss value of a claim using the given claim features – the better the model the closer the predicted loss value will be to the actual loss of a claim.

The test data does not include the loss for the claim. The loss values for the claims in the test data are stored on the Kaggle website, but inaccessible to the competitors. The model generated from the training data is used to predict the loss values for the claims in the test data, which are uploaded to Kaggle for comparison with the actual values. These predictions are then scored based on how close they came to the actual loss values and the final score for the predictions is used to rank the submission against the efforts of all the other competitors.

Metrics

A Kaggle competition is scored in order to rank the contributions of each competitor. A metric for computing the score is chosen by the competition organizer. In this case, AllState chose MAE, or Mean Absolute Error, as the metric by which predictions would be scored.

Mean absolute error, is defined (https://en.wikipedia.org/wiki/Mean_absolute_error) as:

$$\mathrm{MAE} = \frac{\sum_{i=1}^{n} |y_i - x_i|}{n} = \frac{\sum_{i=1}^{n} |e_i|}{n}.$$

Where:

y_i is a prediction

x_i is the actual value

n is the number of predictions

e_i is the residual: the difference between the prediction and the actual value

In other words, MAE is the sum of the absolute value of the residuals, divided by the number of predictions.

Evaluation metrics need to be appropriate for the type of modeling being done, broadly speaking: classification models have one set of metrics to choose from, such as AUC (Area Under the ROC Curve) and Confusion Matrix, and regression models have their own metrics, such as MAE, Mean Average Error and MSE, Mean Squared Error (see examples of different types of eval metrics here: https://machinelearningmastery.com/metrics-evaluate-machine-learning-algorithms-python).

The MAE metric was chosen by the competition organizer, State Farm, so it's hard to say why they chose MAE over another metric that would also be appropriate for evaluating a regression, such as MSE. I did find some interesting points of view on the various advantages and disadvantages of MAE over other metrics for model evaluation.

The author of this Medium article https://medium.com/human-in-a-machine-world/mae-and-rmse-which-metric-is-better-e60ac3bde13d points out that "From an interpretation standpoint, MAE is clearly the winner. RMSE does not describe average error alone and has other implications that are more difficult to tease out and understand."

Phil Birnbaum of Sabermetric Research points out that the MAE is the correct way to calculate the impact of errors on the real world. In his example it is a business owner who runs a restaurant and who sometimes orders too few lobsters to meet demand and sometimes orders too many. She wants to know how much these errors are costing her http://blog.philbirnbaum.com/2012/08/the-standard-error-vs-mean-absolute.html:

> ***"The standard deviation is 10 ... so the typical error is 10 lobsters either way, or $100. That means, over 1,000 days, the total cost would be $100,000. …my guess of $100,000, based on***

the SD, is too high. To figure out what the errors cost, we need to know the mean [absolute] error. …for a normal distribution, the average [absolute] error is 20% smaller than the SD. Which means that the cost of the lobster errors isn't $100,000 -- it's only $80,000."

Without knowing how State Farm plans to use the information, it's difficult to say why they chose MAE over other regression metrics, but based on my research I would say it's because they want to know the actual impact (in USD) on the company of mistakes in estimating claim reserves.

II. Analysis

Data Exploration

Two data files, "train.csv" and "test.csv" are supplied with the competition. They are text files in Comma Separated Value (CSV) format. The data from these files were loaded into Pandas DataFrames in Jupyter Python notebook for exploration and analysis purposes. A "sample_submission.csv" file is also included to show what the submission file to be uploaded to Kaggle.com for evaluation should look like.

The train data set has 132 features (or "columns" in a Pandas DataFrame) and 188,318 items (or "rows" in a Pandas DataFrame), and the test data set has 131 features and 125,546 items. The train and test data have the same features except for the "loss" column, which contains the claim loss value as loss information only exists in the train data set. After building a model the loss values for the test data will be predicted, exported to a csv file in the specified format, then uploaded to Kaggle to be scored.

Kaggle competitions include a Leaderboard which shows the scores of competitors during the competition as well as the scores for any baseline models supplied. If it is after the competition new submission scores will not show on the Leaderboard, but you can refer to it to see how your predictions fared with respect to those who participated during the competition as well as baseline models.

The data in the train file comprises 116 categorical features and 16 "continuous" features. The continuous features are floating-point numeric values. Only 14 of the continuous features are used to build the model because one of these features, loss, is the target value, and the other is id, which should carry no information, so it is excluded. (However, in reading the discussions for the competition there appears to have been a "data leak" in that the id values could be exploited to improve one's score – however, I chose not to take advantage of this). None of the features had missing data (which would be stored in a Pandas DataFram as "NaN", or "Not a Number").

Sample of the data:

	id	cat1	cat2	cat3	cat4	…	cat111	cat112	cat113	cat114	cat115	cat116
0	1	A	B	A	B		C	AS	S	A	O	LB
1	2	A	B	A	A		A	AV	BM	A	O	DP
2	5	A	B	A	A		A	C	AF	A	I	GK
3	10	B	B	A	B		C	N	AE	A	O	DJ
4	11	A	B	A	B		C	Y	BM	A	K	CK

	cont1	cont2	cont3	cont4	…	cont11	cont12	cont13	cont14	loss
0	0.7263	0.24592	0.18758	0.78964		0.5697	0.59465	0.82249	0.71484	2213.18
1	0.33051	0.73707	0.59268	0.61413		0.3383	0.36631	0.61143	0.3045	1283.6
2	0.26184	0.35832	0.4842	0.23692		0.3814	0.37342	0.19571	0.77443	3005.09
3	0.32159	0.55578	0.52799	0.37382		0.3279	0.32157	0.60508	0.60264	939.85
4	0.2732	0.15999	0.52799	0.4732		0.2047	0.20221	0.24601	0.43261	2763.85

The categorical features comprise string values, such as "A", "B", and "BD". Many of the categorical features have only two values, "A" and "B", but several have more: the largest categorical feature has 326 unique values. These, of course, had to be encoded – translated into numbers – for the machine learning algorithm. In many of these categorical features, one value is used far more often than the other(s). For instance, for the "cat8" feature, 177274 out of 188318, or 94%, of the rows have the "A" value.

I checked all the categorical features for values that existed in only one of the data sets (train or validate). Some values were found in only one of the data sets.

Stats on the continuous features were as follows:

	cont1	cont2	cont3	cont4	cont5	cont6	cont7
count	188,318	188,318	188,318	188,318	188,318	188,318	188,318
mean	0.4939	0.5072	0.4989	0.4918	0.4874	0.4909	0.4850
std	0.1876	0.2072	0.2021	0.2113	0.2090	0.2053	0.1785
min	0.0000	0.0011	0.0026	0.1769	0.2811	0.0127	0.0695
25%	0.3461	0.3583	0.3370	0.3274	0.2811	0.3361	0.3502
50%	0.4758	0.5558	0.5280	0.4529	0.4223	0.4409	0.4383
75%	0.6239	0.6818	0.6342	0.6521	0.6433	0.6550	0.5910
max	0.9850	0.8627	0.9443	0.9543	0.9837	0.9972	1.0000

	cont8	cont9	cont10	cont11	cont12	cont13	cont14	loss
count	188,318	188,318	188,318	188,318	188,318	188,318	188,318	188,318
mean	0.4864	0.4855	0.4981	0.4935	0.4932	0.4931	0.4957	3037.3377
std	0.1994	0.1817	0.1859	0.2097	0.2094	0.2128	0.2225	2904.0862
min	0.2369	0.0001	0.0000	0.0353	0.0362	0.0002	0.1797	0.6700
25%	0.3128	0.3590	0.3646	0.3110	0.3117	0.3158	0.2946	1204.4600
50%	0.4411	0.4415	0.4612	0.4572	0.4623	0.3635	0.4074	2115.5700
75%	0.6236	0.5668	0.6146	0.6789	0.6758	0.6900	0.7246	3864.0450
max	0.9802	0.9954	0.9950	0.9987	0.9985	0.9885	0.8448	121012.2500

The continuous feature values appear to have already been standardized: their min/max/mean values, excluding the target feature, loss, are all very close to 0 / 1 / .5. However, some of the continuous features were skewed to such a degree that it made sense to apply a transformation of some kind in an attempt to normalize them since regression algorithms tend to perform better on normalized data.

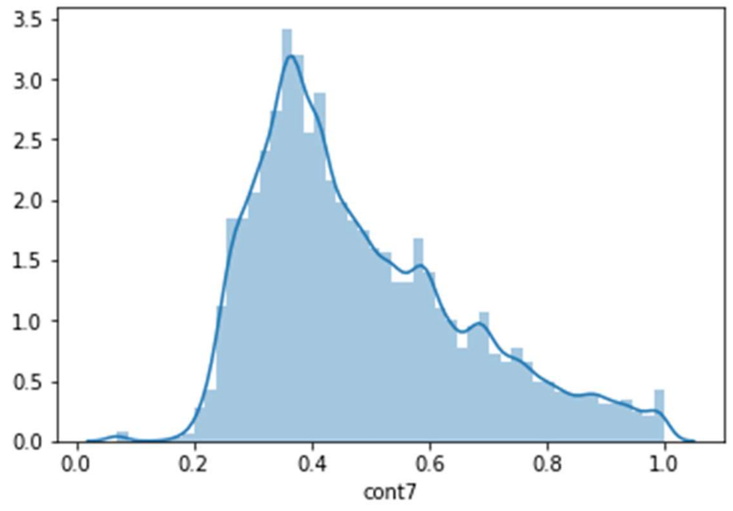
The top three skewed continuous features were:

```
cont7      0.826053
cont9      1.072429
loss       3.794958
```

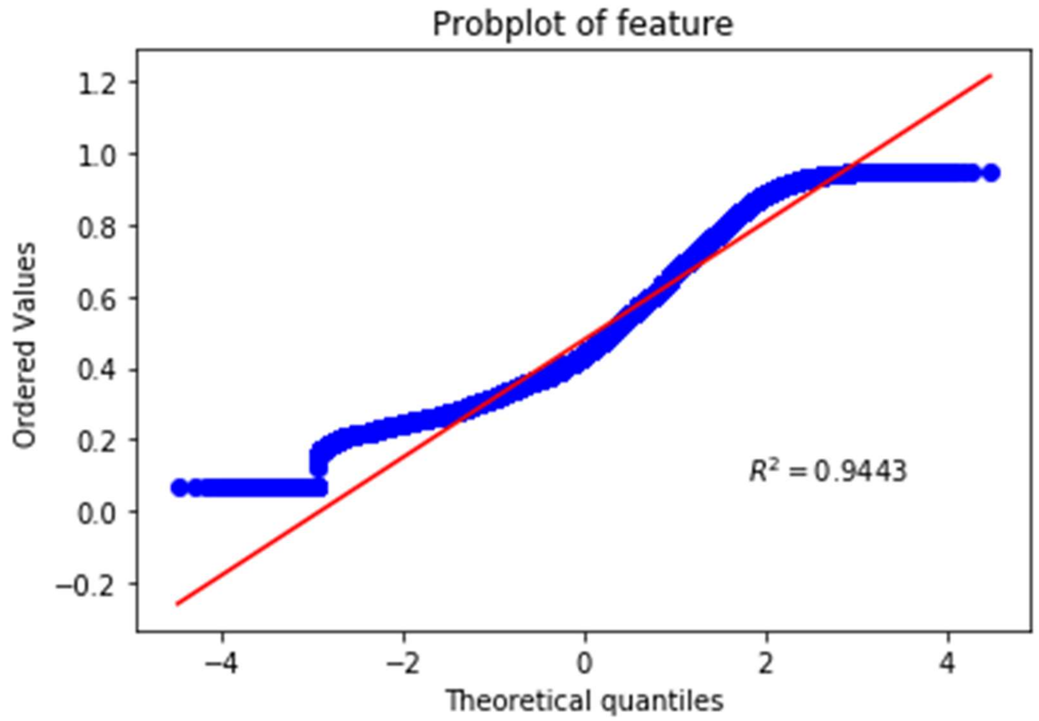
To check for outliers in all of the continuous features I used 1.5 times the IQR (Inter-Quartile Range: the difference between the third and the first quartiles) as the definition of an outlier. Cont7, 9 & 10 contained outliers.

Exploratory Visualization

The cont7 continuous feature column showed up as being very skewed. Here's how it looks (the y-axis is the % of the area under the probability distribution function curve for a given cont7 value):



This probability plot shows the deviation of cont7's values from a normal distribution:



The R^2 value, indicating how close the cont7 values are to a normal distribution, is actually really good. However, given the skewness of the data in this column, applying boxcox will certainly improve it.

Algorithms and Techniques

Since the target value, the thing we want to ultimately predict (the claim loss values), are included in the training data (and known for the test data, just not revealed), this is a supervised learning scenario. The target value is a float – a continuum of numerical values rather than a discreet set of categories – so a regression algorithm is called for in this case.

The approach I took is the supervised machine learning technique of Gradient Boosting Decision Tree (GBDT) to create a model that can be used to predict the loss values of new claims given the same features used to create the model. GBDTs such as xgboost are popular choices for Kaggle competitions, so I thought it would be a good approach to try. The implementation of gradient boosting I used is called LightGBM: https://github.com/Microsoft/LightGBM, which claims to be an improvement over xgboost. I will first discuss how GBDTs work in general, then focus on the distinctives of the LightGBM approach.

Algorithm

What are GBDTs, exactly? "GBDT is an ensemble model of decision trees, which are trained in sequence. In each iteration, GBDT learns the decision trees by fitting the negative gradients (also known as residual errors)." https://papers.nips.cc/paper/6907-lightgbm-a-highly-efficient-gradient-boosting-decision-tree.pdf I will unpack the meaning of this definition below.

Unlike the well-known Random Forest decision tree ensemble, which builds multiple decision trees in parallel, GBDTs build trees sequentially. In a GBDT algorithm, after the first tree is generated the second tree is fitted to "residual errors" based on an initial estimate and the first tree. Subsequent trees are generated on the residual errors from the tree just prior in the sequence (see "Motivation" section here: http://blog.kaggle.com/2017/01/23/a-kaggle-master-explains-gradient-boosting).

In GBDTs, the "residual errors" are calculated from the negative gradient of a loss function. From the xgboost documentation https://xgboost.readthedocs.io/en/latest/model.html#objective-function-training-loss-regularization:

> ***A very important fact about objective functions is they must always contain two parts: training loss and regularization.***
>
> $$obj(\theta)=L(\theta)+\Omega(\theta)$$
>
> ***where L is the training loss function, and Ω is the regularization term. The training loss measures how predictive our model is on training data. For example, a commonly used training loss is mean squared error.***
>
> $$L(\theta)=\sum_i(y_i-\hat{y}_i)^2$$
>
> ***The regularization term is what people usually forget to add. The regularization term controls the complexity of the model, which helps us to avoid overfitting.***

Briefly, the regularization term can be a typical "L_2 norm over the leaf scores" (from slide 25 here: https://homes.cs.washington.edu/~tqchen/pdf/BoostedTree.pdf). Another option is the L_1 norm (see the helpful descriptions and trade-offs of each here: http://www.chioka.in/differences-between-l1-and-l2-as-loss-function-and-regularization).

The "leaf scores" mentioned above for regularization are not just for regularization. They are also used to calculate the gradient for a given leaf. The scoring function used is specific to the loss function selected: for MSE, the leaf scores would be "the average of the target values of the cases within each leaf l" (page 63 here: http://www.dcc.fc.up.pt/~ltorgo/PhD/th3.pdf).

Gradients are used to calculate the "residual errors" we will use to fit the next tree in the sequence. The gradient is the partial derivative of the loss function, which for MSE gives (from slide 22 here: https://homes.cs.washington.edu/~tqchen/pdf/BoostedTree.pdf):

$$\partial_{\hat{y}^{(t-1)}}(\hat{y}^{(t-1)}-y_i)^2 = 2(\hat{y}^{(t-1)}-y_i)$$

Note: the MSE loss function typically includes a ½ factor at the front to make the math cleaner (from slide 38):

$$l(y_i,\hat{y}_i)=\tfrac{1}{2}a_i(\hat{y}_i-y_i)^2 \qquad g_i=a_i(\hat{y}_i-y_i)$$

Note that this is just a standard "residual" formula: the difference between the predicted values and the actual values. However, this is an example using MSE as the loss function, which simplifies the math. The same approach can be used with other loss functions, which will have different gradient functions.

Finally, in order to calculate our first "residual errors" we need a starting point to which we can compare our first tree's results. For a GBDT using the MSE as its loss function, this will simply be the average of the target values for all of the training data (see the "Gradient Boosting – Draft 2" section here: http://blog.kaggle.com/2017/01/23/a-kaggle-master-explains-gradient-boosting). This value will be the same for all rows in the training data.

Now we can begin (the following is based on the "Gradient Descent", "Leveraging Gradient Descent" and "Squared" error sections here: http://blog.kaggle.com/2017/01/23/a-kaggle-master-explains-gradient-boosting).

GBDT core steps (simplified)

From the "Leveraging Gradient Descent" section here: http://blog.kaggle.com/2017/01/23/a-kaggle-master-explains-gradient-boosting:

```
Initialize the model with a constant value:
$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma)$

For m = 1 to M:
    Compute pseudo residuals, $r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)}$ for $i = 1, \ldots, n.$
    Fit base learner, $h_m(x)$ to pseudo residuals
    Compute step magnitude multiplier $\gamma_m$. (In the case of tree models, compute a different $\gamma_m$ for every leaf.)
    Update $F_m(x) = F_{m-1}(x) + \gamma_m h_m(x)$
```

The sections "Square Error" and "Absolute Error" at http://blog.kaggle.com/2017/01/23/a-kaggle-master-explains-gradient-boosting show examples of how to do this for loss functions MSE and MAE. I have reproduced the calculations in a spreadsheet, gbdt.xlsx, which is included in my zip file for this project.

The steps I took to reproduce the author's results were as follows:

1. **Age**: Target values given by the example data.
2. **F_x**: Calculate the starting point prediction. This is our initial estimate of the target values, which in this case will be the average of all the target values in the training data. For a MSE loss function, it is the mean of the target values. For MAE it is the median.
3. **PseudoResidual$_x$**: Calculate the "pseudo-residuals" using the gradient function, the first partial derivative of the loss function. For MSE this is just a normal residual: Age – F_x. For MAE it is the Sign function applied to the same residual: Sign(Age – F_x).
4. **Assigned Leaf**: Leaf assignments taken from the example data. Note: The process of fitting a GBDT to training data is described below.
5. **H_x**: For each leaf in the tree, calculate the mean of the target values in each leaf.

6. **Target-F_x/h_x**: This is an intermediate result for the gamma calculation: (Age-F_x)/h_x.
7. **gamma$_x$**: For MSE this the average of the Target-F_x/h_x intermediate value calculated in the previous step for each leaf. MAE it is the mean of the same value.
8. **F_{x+1}**: Calculate a new predicted target value: F_x+h_x*gamma$_x$

Repeat steps 3 – 8. The number of iterations is a hyperparameter supplied to the GBDT algorithm.

Fitting a tree with GBDTs

To fit a tree, you have to know two things: how to create the splits and when to stop building the tree.

The splitting algorithm is as follows (from slide 32 here: https://homes.cs.washington.edu/~tqchen/pdf/BoostedTree.pdf)

"For each node, enumerate over all features

- For each feature, sorted the instances by feature value
- Use a linear scan to decide the best split along that feature
- Take the best split solution along all the features"

For GBDTs, the gain for a given split is defined as (https://xgboost.readthedocs.io/en/latest/model.html#learn-the-tree-structure):

Split a leaf into two leaves, and the score it gains is

$$Gain = \frac{1}{2}\left[\frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{(G_L+G_R)^2}{H_L+H_R+\lambda}\right] - \gamma$$

This formula can be decomposed as 1) the score on the new left leaf 2) the score on the new right leaf 3) The score on the original leaf 4) regularization on the additional leaf. We can see an important fact here: if the gain is smaller than γ, we would do better not to add that branch. This is exactly the pruning techniques in tree based models!

Note that in the above description we are also given the condition to stop tree building, "if the gain is smaller than [gamma], we would do better not to add that branch."

The definitions of G_j and H_j are (from here: https://xgboost.readthedocs.io/en/latest/model.html#the-structure-score):

$$G_j = \sum_{i\in I_j} g_i \text{ and } H_j = \sum_{i\in I_j} h_i$$

I_j "is the set of indices of data points assigned to the j-th leaf" (from the same reference as above).

g_i and h_i are defined as the first and second partial derivatives of the loss function (here: https://xgboost.readthedocs.io/en/latest/model.html#additive-training)

$$g_i = \partial_{\hat{y}_i^{(t-1)}} l(y_i, \hat{y}_i^{(t-1)})$$
$$h_i = \partial^2_{\hat{y}_i^{(t-1)}} l(y_i, \hat{y}_i^{(t-1)})$$

For the MSE loss function, g_i and h_i are defined on slide 38 here: (https://homes.cs.washington.edu/~tqchen/pdf/BoostedTree.pdf)

$$l(y_i, \hat{y}_i) = \frac{1}{2}a_i(\hat{y}_i - y_i)^2 \qquad g_i = a_i(\hat{y}_i - y_i) \quad h_i = a_i$$

The above describes the process of gradient descent on decision trees, which is what differentiates GBDTs from other decision tree algorithms. However most decision tree algorithms, including GBDTs, also include other techniques such as selecting random rows (bagging) and random columns from the training data (see the section “Gradient Boosting – Draft 5” here: http://blog.kaggle.com/2017/01/23/a-kaggle-master-explains-gradient-boosting).

Also, like many other learning models, not just decision trees, the calculation of the final estimates for each tree (F_x) will include a “learning factor”, a value from 0 – 1, that is typically set to “shrink” over time. This reduces the magnitude of the changes to the predictions from one tree to the next as it gets (presumably) closer and closer to the actual target values (as mentioned in the section “Gradient Boosting – Draft 4” here: http://blog.kaggle.com/2017/01/23/a-kaggle-master-explains-gradient-boosting).

Some of the features which distinguish LightGBM from other GBDTs include (https://lightgbm.readthedocs.io/en/latest/Features.html):

1. It “uses the histogram based algorithms, which bucketing continuous feature(attribute) values into discrete bins, to speed up training procedure and reduce memory usage.”
   Note: this is also an option that can be selected in xgboost, it’s just not the default.
2. Optimization for sparse features.
3. Leaf-wise (best-first) tree growth.
4. Optimal split for categorical features.

LightGBM also includes other optimizations such as DART, early stopping, etc.

Feature processing techniques

I did minimal feature pre-processing at first – just the necessary basics such as creating dummy features from categorical features – and ran LightGBM with its default parameters – just specifying random_state to get consistent results over multiple runs – to see how it did in comparison with the benchmark models.

I used sklearn’s GridSearchCV to search over a subset of the algorithm’s hyperparameters.

learning_rate (default=0.1)
n_estimators (default=10)
num_leaves (default=31)
min_data_in_leaf (default=20)

I chose these parameters to tune based on the parameters that came up in articles about tuning decision tree boosting algorithms, such as LightGBM and xgboost, and the documentation. The following provided helpful insights for tuning the hyperparameters:

Which algorithm takes the crown: Light GBM vs XGBOOST?

Light GBM Docs: Parameters Tuning

I looked at several “kernels” for this competition – code posted by competitors during and after the competition – and read the associated discussions. Both the code and the discussions were very helpful in deciding what feature processing would be appropriate/helpful.

Based on my explorations in the various kernels and discussion for this competition, I chose those feature transformations that made sense to me and seemed to offer the biggest “bang for the buck”. I applied those transformations and re-ran the regression multiple times, trying various combinations of LightGBM parameter values after some of the feature transformations to see what combination improved the model the most.

Benchmark
Allstate provided two benchmark models for the Kaggle competition: one based on a Random Forest model, the other consisting of zero loss values for all claims in the test data set. The Random Forest model scored 1227.74973 and ranked just above the 2302th competitor. The Zero model scored 3031.71274 and ranked just above the 2975th competitor.

III. Methodology

Data Preprocessing
Note: in my Jupyter notebook, I called the DataFrame with the test.csv data “validate” in order not to confuse it with the train/test data created with train_test_split for model training/evaluation, so I will be referring to the test.csv data as “validate” from here on out.

The following transformations were applied to the data at various points during the training process – some before training on the default parameters, but most were applied after that.

1. There were values in categorical columns that existed in only the train or “validate” (train.csv) data set. These values were set to NaN in the data set where they occurred.

2. Categorical features in both train and validate were one-hot-encoded using Pandas get_dummies, which created additional feature columns in the data set for each feature value.
3. The target value (loss) was highly skewed. This was transformed with a combination of log and shift functions based on the best transformations found by one of the Kaggle competitors.
4. I removed several highly correlated features (above .75 Pearson's correlation coefficient): cont1, cont9, cont10, cont11, cont12.
5. Of the three columns found to contain outliers (cont7, 9 & 10), only cont7 was left at this point.
    a. I Tried setting the outlier values in cont7 to the median value for cont7, but this didn't improve the model's score, so I didn't include this transformation in the final model.
    b. I then dropped the rows with cont7 outliers which did improve the model's results.
6. Of the three highly skewed features found (loss, cont9 and cont7), cont9 had been dropped and loss had already been transformed, so I applied the BoxCox function to cont7, which improved the model's score.
7. I tried PCA to see if reducing the number of remaining continuous features would help improve the model's score, but it didn't help so I left this out of the data processing for the final model.

Implementation

Before running LightGBM with Default Parameters I performed the following data pre-processing steps:

- Checked for columns in train but not in validate.
- Checked continuous features for missing values.
- Created a Series with just the target value, "loss", from the train data set.
- Created new train and "validate" DataFrames from the train and test data sets that excluded the id and loss columns.
- Checked all the categorical features for values that existed in only one of the data sets (train or validate). Those categorical values found only in one set of data were set to NaN in the data set where they occurred.
- One-hot-encoded the categorical features in both train and validate using the Pandas get_dummies function.
- Verified that no columns existed in one DataFrame but not the other.

I used train_test_split to section off 20% of the data for model testing (to avoid overfitting). I then fit the LightGBM regressor to the training data with the default parameters. I used GridSearchCV to try a variety of hyperparameters in an attempt to improve the mae score and stopped optimizing when the mae score of the resulting model stopped improving substantially.

I then tried additional data processing, checking the results on the model score periodically by re-splitting the data using train_test_split and tuning the LightGBM model's parameters with GridSearchCV:

- Transformed highly skewed target values (loss).
- Removed highly correlated continuous features (cont1, cont9, cont10, cont11, cont12).
- Tried setting outliers to the median value for the feature.
- Removed rows with outliers.

At this point I tried a couple more transformations to see if I could achieve additional improvement in my model's mae score.

- Transformed the one remaining highly skewed continuous feature, cont7, using BoxCox.
- Tried PCA to reduce the number of continuous features.

After transforming cont7 and performing some additional hyperparameter optimizations I was able to eke out some small improvements to the mae score (results for the final model are presented in the Results section, below).

I then applied the same pre-processing steps – the ones that helped improve the training model, minus the removal of outliers – to the "validate" data:

- Dropped highly correlated features.
- Transformed cont7 using BoxCox.

Refinement

Fitting the LightGBM regressor with default parameters and minimal feature processing achieved a mae score of 1601.8762.

I then tuned the following parameters using sklearn's GridSearchCV:

learning_rate
n_estimators
num_leaves
min_data_in_leaf

After correcting the target value's extreme skew and removing some highly correlated "continuous" features, LightGBM was able to achieve a mae score of 1139.1960. The best parameters at this point were found to be:

learning_rate: .1
n_estimators: 500
num_leaves: 16
min_data_in_leaf: 125

When outliers were set to the mean for that feature, the model score degraded a bit to 1140.2383. However, after outright removing outlier rows, the model score improved to 1128.0170. This was achieved with existing hyperparameters.

After additional data processing transformation of the cont7 column to correct its extreme skewness using boxcox, and optimizing the regressor's hyperparameters, the resulting model produced further gains in mae score (results for the final model are reported below in the Results section).

Implementation Issues

One unexpected issue that required more coding than I anticipated was the problem of different categorical values between the test.csv and train.csv (validate) files. Several of the columns' had unique sets of values that differed between test and validate. For instance, in train, a column's unique set of values might include "A" and "B" but in validate it might include "A", "B", and "C". This causes issues when one-hot-encoding categorical values using dummy columns since test will have two dummy columns, one for "A" values and one for "B" values, and validate will have three, one for "A", one for "B" and one for "C" values.

I found a kernel on Kaggle that addressed this issue – but it turned out the code had a bug! In fixing the bug I improved the code using sets of column names and set operations to get the intersection of unique categorical values not in both sets.

```
remove = set(train[column].unique()) ^ set(validate[column].unique())
```

I learned a lot about the Pandas library in this project. For example, it was really cool to be able to get a list of correlated features in a couple of lines of code:

```
train_corr = train_orig.iloc[:,1:-1].corr(method='pearson').stack()
train_corr[(train_corr < 1) & (train_corr >= .75)].sort_values(
             ascending=False).iloc[1::2]
```

Calculating and removing outlier values was challenging and another great opportunity to learn more about Pandas. It was great learning how to apply my IQR test to an entire column with one line of code:

```
series.apply(lambda x: not lower_bound <= x <= upper_bound)
```

Replacing outliers with the median of the feature (which was not a transformation that did not help improve my score so the results of this transformation were not included in the data processing pipeline) was challenging. I examined the data's pre and post transform values to be sure it was correct.

IV. Results

Model Evaluation and Validation

After applying various types of data processing to the training data's features, keeping only those that actually improved the model's score, and tuning a subset of the model's hyperparameters after each transformation was applied, the final mae score achieved on the train data was 1126.7992.

The final hyperparameters of the best-performing LightGBM model were:

learning_rate: .2
n_estimators: 500
num_leaves: 12
min_data_in_leaf: 375

Trying values near the model's final hyperparameter values did not appreciably change the result, suggesting that this is model is robust since small changes to hyperparameters did not result in radically different results.

After applying the appropriate transformations to the “validation” (test.csv) data – as described in the Methodology section above – and fitting the final model created from the training data to the validation data, the resulting predictions were exported to a CSV file in the format specified by the All State Claims Severity Kaggle competition and uploaded to Kaggle for evaluation against the stored target (loss values) for the validation data set.

The mae score achieved on the validation (test.csv) data set was 1131.90796.

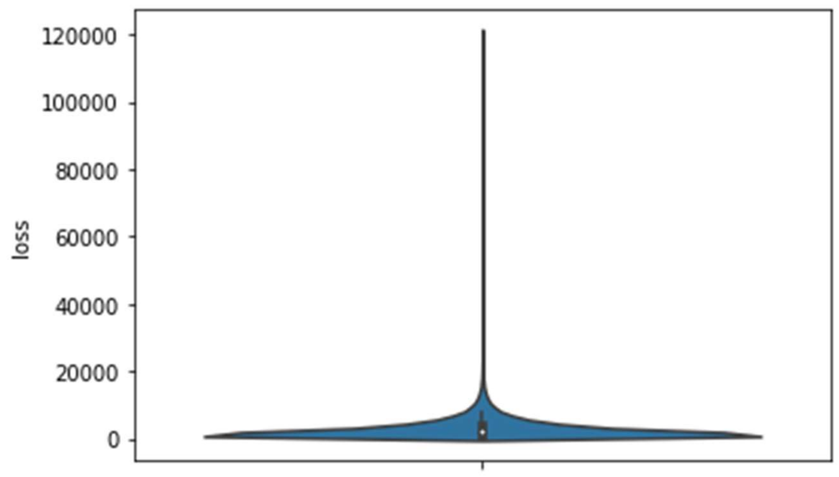
Justification

My model’s best mae score was quite a bit better than the benchmarks: 1131.90796 vs 1227.74973 for the Random Forest model and 3031.71274 for the Zero model. The Random Forest model benchmark ranked 2300 out of 3055 (on the final Leaderboard for the competition). My model’s final mae score would place me at rank 1351, an improvement of 949 places in the ranking, which is 31% better than the RF benchmark.

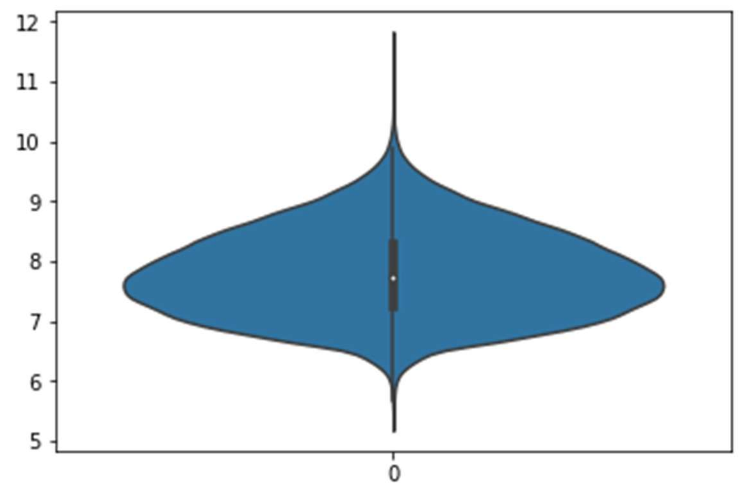
V. Conclusion

Free-Form Visualization

The target feature, claim loss value, was highly skewed, as shown by this Seaborn violin plot:



I used a feature transformation on it suggested by another Kaggle competitor: shift by 200 then take the log.



The Seaborn violin plot created after the transformation shows the resulting feature values for claim loss to be much closer to a normal distribution. With this change I was able to achieve a mae score of 1146.133. Prior to applying this feature transformation to the data, my model's best score was 1211.805.

Reflection

After loading and exploring the data, I then had to deal with the tricky issue of the train and "validation" (test.csv) data sets having different unique values for the same categorical features. This had to be addressed because when dummy columns were created for the values in these features, the training dataset would differ from the validation set. This became an issue when I got errors trying to apply the model created from the training data to the validation data. I found some code that another Kaggle competitor had created to address this, which, after fixing a bug in the code and improving it, fixed the issue by setting the troublesome values (those not in both data sets) to NaNs.

With this housekeeping chore complete, I was able to proceed with the first of several data processing steps: creating the one-hot-encoded dummy columns from the categorical features. This was the minimum pre-processing necessary to begin training the LightGBM model, so I immediately jumped into splitting the data into train/test splits and seeing what I could get from LightGBM with the default values.

I then chose a set of hyperparameters to focus my tuning efforts on. I picked them based on the examples in the documentation for LightGBM and what I saw being used in the articles I read about the algorithm. After doing some tuning of these parameters using GridSearchCV, I proceeded with more feature engineering starting with one recommended by one of my Udacity ML mentors: normalizing the highly skewed continuous target value, claim loss. After making this change to the data set I performed additional hyperparameter tuning and tried other types of feature engineering: checking collinearity and removing some highly correlated columns (features with a Person's correlation greater than .75), removing outliers (values outside the IQR * 1.5 range), and normalizing a skewed continuous feature using BoxCox.

After each data processing step I checked that my mae score was improving. I tried and abandoned a couple of approaches that didn't improve my model's mae score: setting outlier values to the feature mean value and applying PCA to the remaining continuous features.

I was amazed at how much improvement I got from some of the feature transformations, especially the first one: normalizing the continuous target values. Also, I didn't expect to get as much of a boost as I did from just removing outliers. I was very happy with the speed of the LightGBM algorithm as I was able to try many different feature transformations and combinations of hyperparameter. I also really enjoyed working with the Seaborn data visualization library – it has some great plots that were very easy to generate.

Some things were difficult or tripped me up for a while, such as the errors I got when trying to apply my trained model to the validation data before I realized why I had to fix the issue of disparate unique values in the categorical columns. I really learned a lot about the Pandas library – the kernels uploaded by the competition participants and the accompanying discussions were very helpful.

It would have been nice to score higher than I did with my final model (I was in the middle of the pack), but from my reading of the competition discussions, I would have to make use of a data leak (in the claim IDs) and add sophisticated algorithm stacking, tuning the additional algorithms in the stack, to really make more than trivial progress from my final spot in the rankings. However, the differences between the top quartile or so of competitors become extremely minor (except for the surprising gap between the #1 and #2 spot), so if the advantage of the data leak were ignored, with some tweaking, I believe my model would most likely be "good enough" for application in the real world.

Improvement

I picked a sub-set of LightGBM hyperparameters to tune, but (as shown above in the Algorithms and Techniques section of this report), there are many more that could have been chosen (this https://lightgbm.readthedocs.io/en/latest/Parameters-Tuning.html is a good guide to tuning parameters for this boosting algorithm). In particular it would be interesting to try different boosting algorithms, such as DART.

However, adding additional LightGBM hyperparameters to this GridSearchCV would be very time consuming as it does an exhaustive search of all the possible combinations of hyperparameter values. To optimize over a larger set of hyperparameters in a reasonable amount of time I would instead use RandomizedSearchCV: http://scikit-learn.org/stable/auto_examples/model_selection/plot_randomized_search.html.

Given the number and variety of hyperparameters available for this algorithm, I think it is highly likely that additional hyperparameter tuning could find a significantly improved model.

I would also be interested in moving the outlier removal step after normalizing the cont7 feature to see if that improves the mae score.