Kaggle: Allstate Claim Severity Capstone Project

I. Definition

Project Overview

The domain background of this project is the casualty insurance field. Allstate is partnering with Kaggle to free up their customers' time by automating the car insurance claims process. They want to find ways to predict the cost of car insurance claims they receive using data science. This is a project I am especially interested in because Kaggle is a very reputable source for companies looking for data scientists. Furthermore, this competition is a recruitment opportunity and I will be potentially considered for hire if I do well in this competition.

The datasets to be used in this competition are "train.csv" and "test.csv". The model is to be trained off of the training set and then evaluated on Kaggle using the testing set.

Problem Statement

Given categorical and continuous variables, use a machine learning model to regress on the continuous variable "loss". The solution that we are looking for is the model that most reduces the evaluation metric for the predicted "loss" values. To do this, we are going to use regression supervised learning techniques as well as unsupervised learning techniques to create features.

Metrics

Results are to be evaluated off of the Mean Absolute Error (MAE) between the predicted loss and the actual loss. The formula for MAE is a follows:

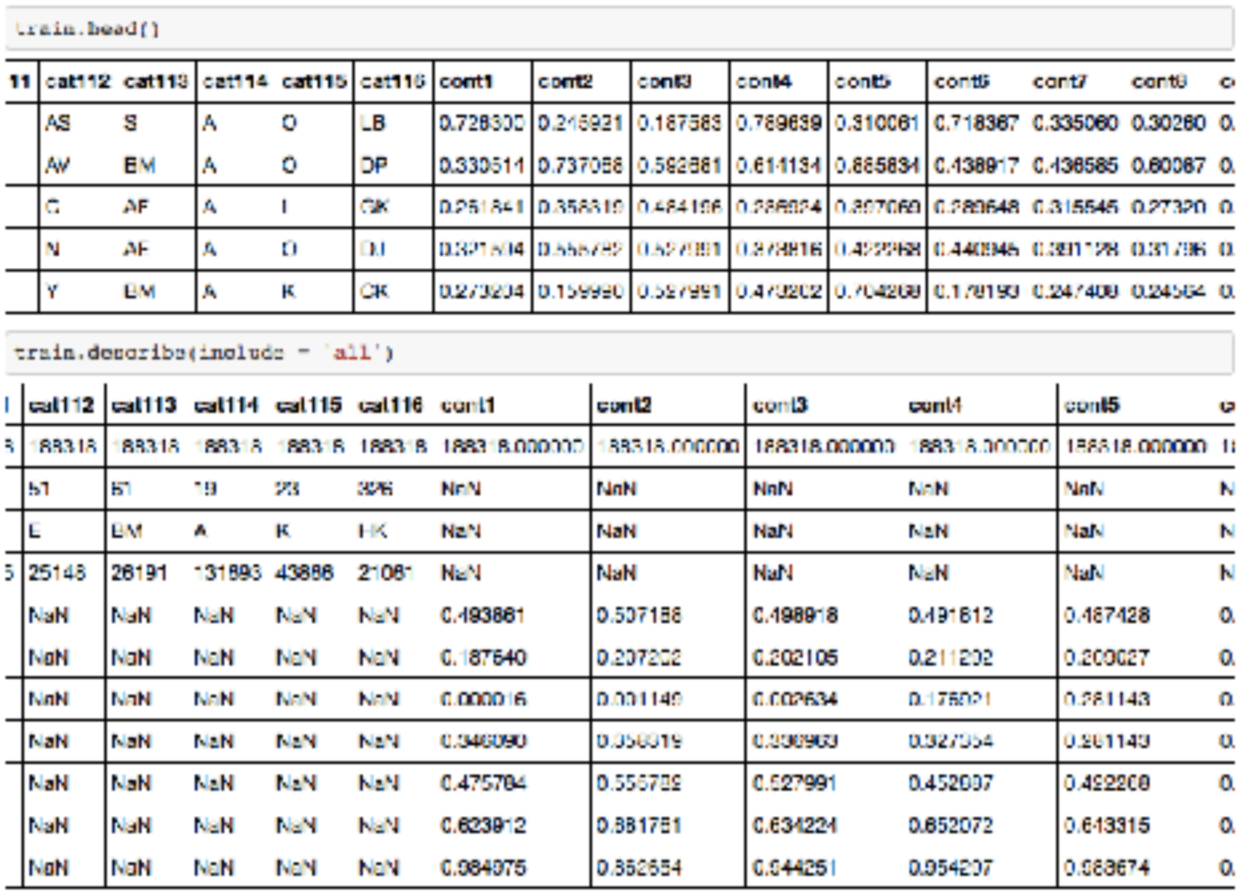
$$MAE = \frac{1}{n}\sum_{t=1}^{n}\left|x_t - \hat{x}_t\right|$$

for n = number of observations, x_t = actual loss and xhat_t = predicted loss. MAE makes sense for this problem because unlike its close counterpart root mean squared error (RMSE), larger errors from the predictions will not be penalized as harshly. In other words, MAE is looking for how far the predictions are from the median while RMSE is looking for how far the predictions are from the mean.

II. Analysis

Data Exploration

There are 188,318 observations in the training set and 125,546 observations in the testing set. There are 116 categorical variables (labeled "cat") and 14 continuous variables (labeled "cont"). Categorical variables range anything from 2 unique factors to 326. Continuous variables range from 0.48 to 1. No missing values are present in the datasets. It seems specific labels were avoided to remove any sort of domain knowledge advantage.



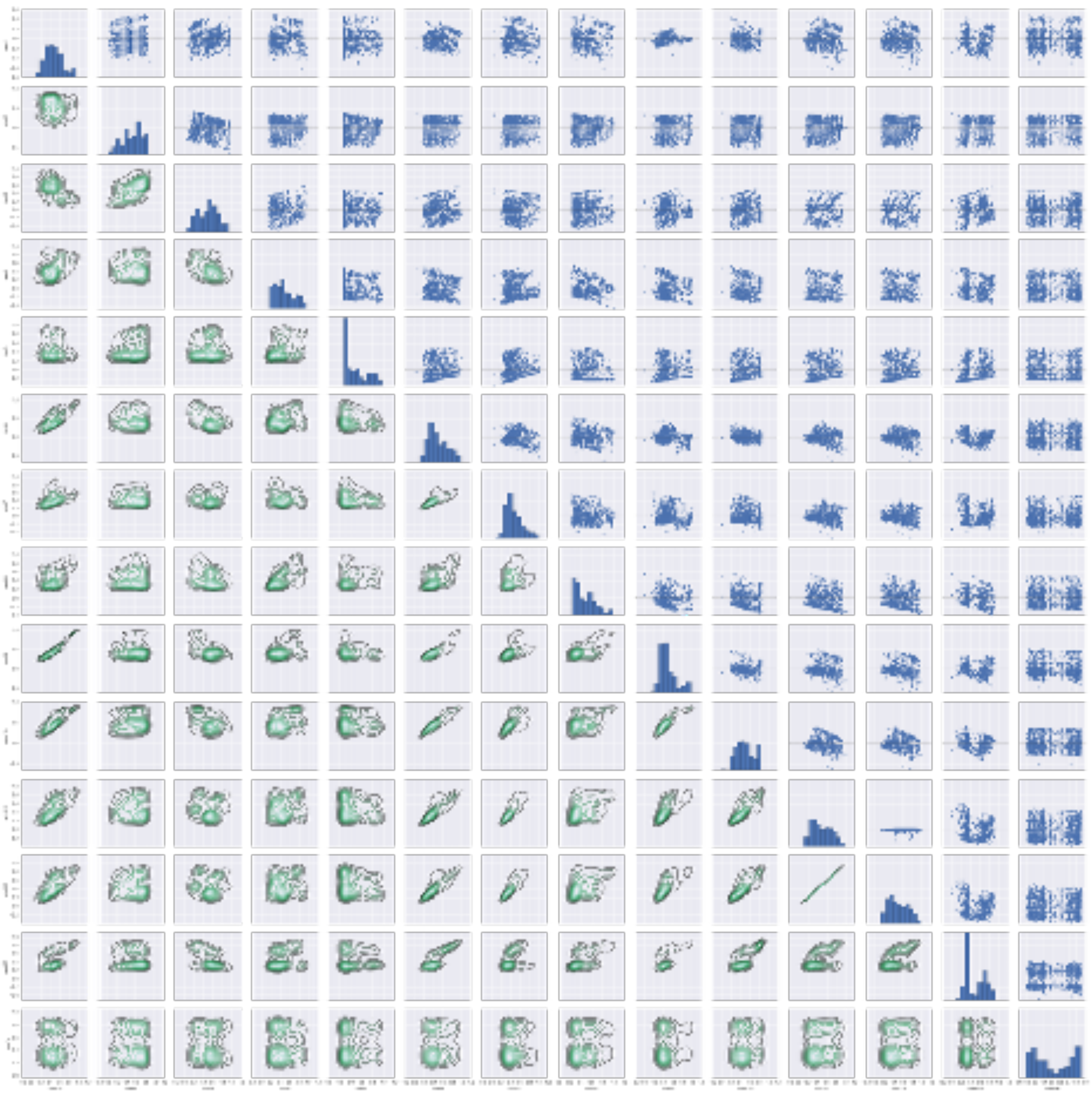
```
train.head()
```

11	cat112	cat113	cat114	cat115	cat116	cont1	cont2	cont3	cont4	cont5	cont6	cont7	cont8	c
	AS	S	A	O	LB	0.726300	0.245921	0.187583	0.789639	0.310061	0.718367	0.335060	0.30260	0.
	AV	BM	A	O	DP	0.330514	0.737068	0.592681	0.614134	0.885834	0.438917	0.436585	0.60087	0.
	C	AF	A	I	GK	0.261841	0.358319	0.484196	0.236924	0.397069	0.289648	0.315545	0.27320	0.
	N	AF	A	O	DJ	0.321594	0.555782	0.527991	0.373816	0.422268	0.440945	0.391128	0.31796	0.
	Y	BM	A	K	CK	0.273204	0.159990	0.527991	0.473202	0.704268	0.178193	0.247408	0.24564	0.

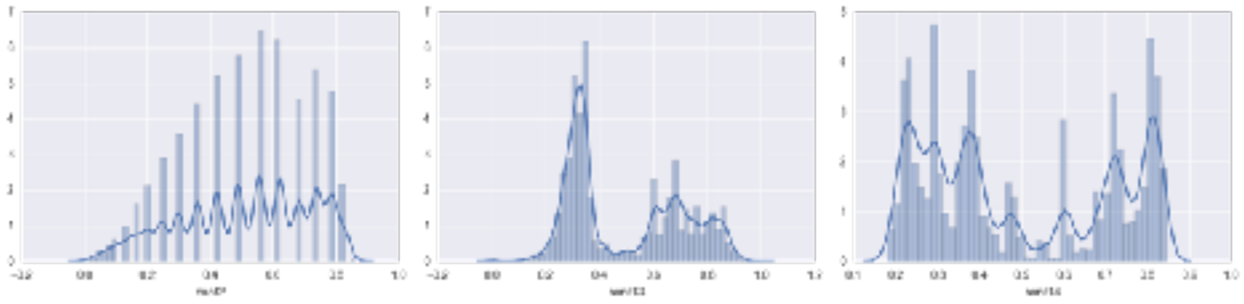
```
train.describe(include = 'all')
```

I	cat112	cat113	cat114	cat115	cat116	cont1	cont2	cont3	cont4	cont5	c
8	188318	188318	188318	188318	188318	188318.000000	188318.000000	188318.000000	188318.000000	188318.000000	1
	51	61	19	23	326	NaN	NaN	NaN	NaN	NaN	N
	E	BM	A	K	HK	NaN	NaN	NaN	NaN	NaN	N
5	25148	26191	131693	43866	21061	NaN	NaN	NaN	NaN	NaN	N
	NaN	NaN	NaN	NaN	NaN	0.493861	0.507188	0.498918	0.491812	0.487428	0.
	NaN	NaN	NaN	NaN	NaN	0.187640	0.207202	0.202105	0.211202	0.209027	0.
	NaN	NaN	NaN	NaN	NaN	0.000016	0.001149	0.002634	0.176921	0.281143	0.
	NaN	NaN	NaN	NaN	NaN	0.346090	0.358319	0.336963	0.327354	0.281143	0.
	NaN	NaN	NaN	NaN	NaN	0.475784	0.555782	0.527991	0.452887	0.422268	0.
	NaN	NaN	NaN	NaN	NaN	0.623912	0.681761	0.634224	0.652072	0.643315	0.
	NaN	NaN	NaN	NaN	NaN	0.984975	0.862654	0.944251	0.954297	0.983674	0.

The first 5 rows and summary statistics of the training set.



A pairwise plot of all 14 continuous variables in the dataset.



Distribution plots of ‘cont2’, ‘cont13’ and ‘cont14’ .

Exploratory Visualization

For the visual exploration of the dataset, I split it between the continuous variables and the categorical variables. It was necessary for me to view the distribution as well as relationships between the continuous variables to identify any patterns if possible.

Some observations o the variables include:

- 'cont2' seems to be a stepwise variable. May be something like “income bracket"?
- Some rather linear relationships seems to appear between several variables
- 'cont11' and 'cont12' seem to have a 1x1 relationship with one another: as 'cont11' increases, so does 'cont12'. Perhaps we can just toss one out or use PCA to extract the outliers from the 1x1 trend?
- 'cont13' definitely seems to split the data into 2 parts and is bimodal. All the other data falls pretty well in line with the peaks of 'cont13'. We can perform some KMeans clustering on this data later.
- 'cont14' also has appears to have a bimodal distribution

Algorithms and techniques

Some steps I will take Use Python to perform exploratory data analysis (EDA) on the training set including:

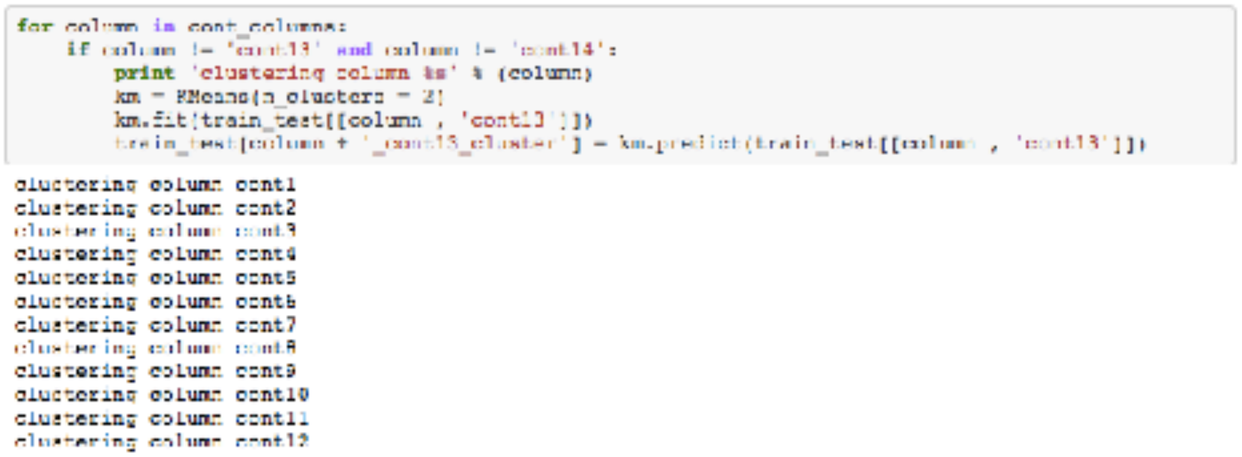
- Create features that will provide more predictive power than just the variables alone including:
    - Transformations of the data such as BoxCox and Logarithmic/Exponential transforms
    - Interactions between variables, especially the categorical variables
    - Clusters created by unsupervised learning methods such as KMeans
- Begin testing different models like:
    - Random Forest
    - AdaBoost
    - XGBoost
- Enhance the model by:
    - GridSearchCV / Random Parameter Search + CV

Benchmark

The current high-scoring public kernel from Misfyre has a cross-validation MAE of 1131.63 (and a testing score of 1108.72665). I will be basing my models off of Misfyre’s script and improve upon it.

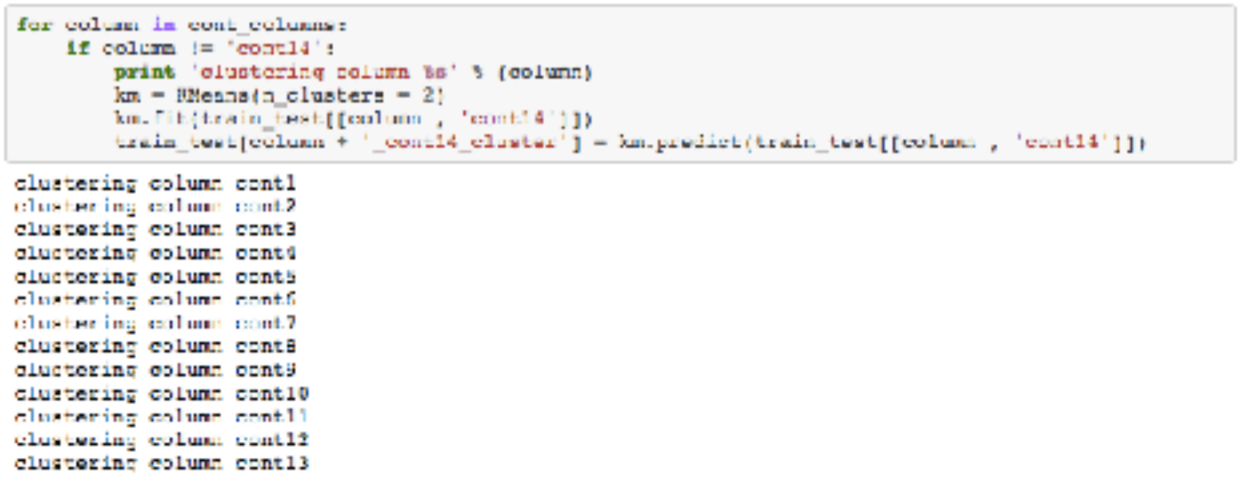
III. Methodology

Data Preprocessing



```
for column in cont_columns:
    if column != 'cont13' and column != 'cont14':
        print 'clustering column %s' % (column)
        km = KMeans(n_clusters = 2)
        km.fit(train_test[[column , 'cont13']])
        train_test[column + '_cont13_cluster'] = km.predict(train_test[[column , 'cont13']])
```

```
clustering column cont1
clustering column cont2
clustering column cont3
clustering column cont4
clustering column cont5
clustering column cont6
clustering column cont7
clustering column cont8
clustering column cont9
clustering column cont10
clustering column cont11
clustering column cont12
```



```
for column in cont_columns:
    if column != 'cont14':
        print 'clustering column %s' % (column)
        km = KMeans(n_clusters = 2)
        km.fit(train_test[[column , 'cont14']])
        train_test[column + '_cont14_cluster'] = km.predict(train_test[[column , 'cont14']])
```

```
clustering column cont1
clustering column cont2
clustering column cont3
clustering column cont4
clustering column cont5
clustering column cont6
clustering column cont7
clustering column cont8
clustering column cont9
clustering column cont10
clustering column cont11
clustering column cont12
clustering column cont13
```

Creating cluster features from cont13 and cont14.

The steps I took to preprocess the data are as follow:

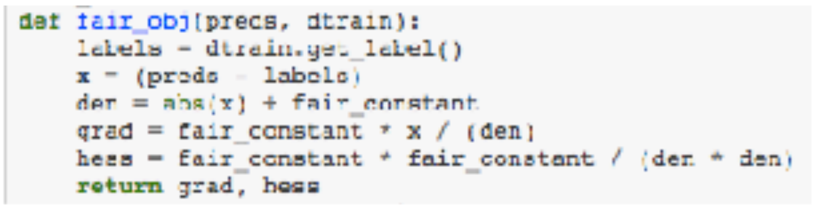
1. I combined train and test sets so we can apply the same preprocessing methods on both sets.
2. For the continuous features although the features were already from 0 to 1, we still needed to check for skew. I applied a boxcox transformation on continuous features that had skew whose absolute value was greater than 0.25.
3. After checking for we wanted to make sure than the continuous variables were in the same min-max scale and scaled using StandardScaler.
4. Creating clustering features for cont13 and 14 on all the other continuous features.

5. Creating a PCA feature from cont11 and cont12.
6. Setting factors of categorical features found in the test set but not raining set to be NaN so the model will not mistakenly act on those.
7. Creating interaction features from many of the categorical features.
8. Factorizing categorical features using pandas factorize. The reason for using this is to give a sense of lexical encoding/turning categorical features into continuous features and that creating dummy variables makes the model train much slower.

Implementation

I decided on using XGBoost as my supervised learning algorithm because it seemed to be the algorithm of choice on the Kaggle Kernels and because it had the option of adding a custom objective function which determines how the algorithm will be trained on new data. This was especially powerful as most regression machine learning algorithms default on minimizing RMSE, instead of MAE as this problem requires. One of the ways to minimize for MAE instead of RMSE is to use an M-estimator, which is a type of estimation function that seeks to minimize the MAE. I was able to find a 'fair objective function' to use which other users have found to converge faster as a result of it's non-continuous second derivative.

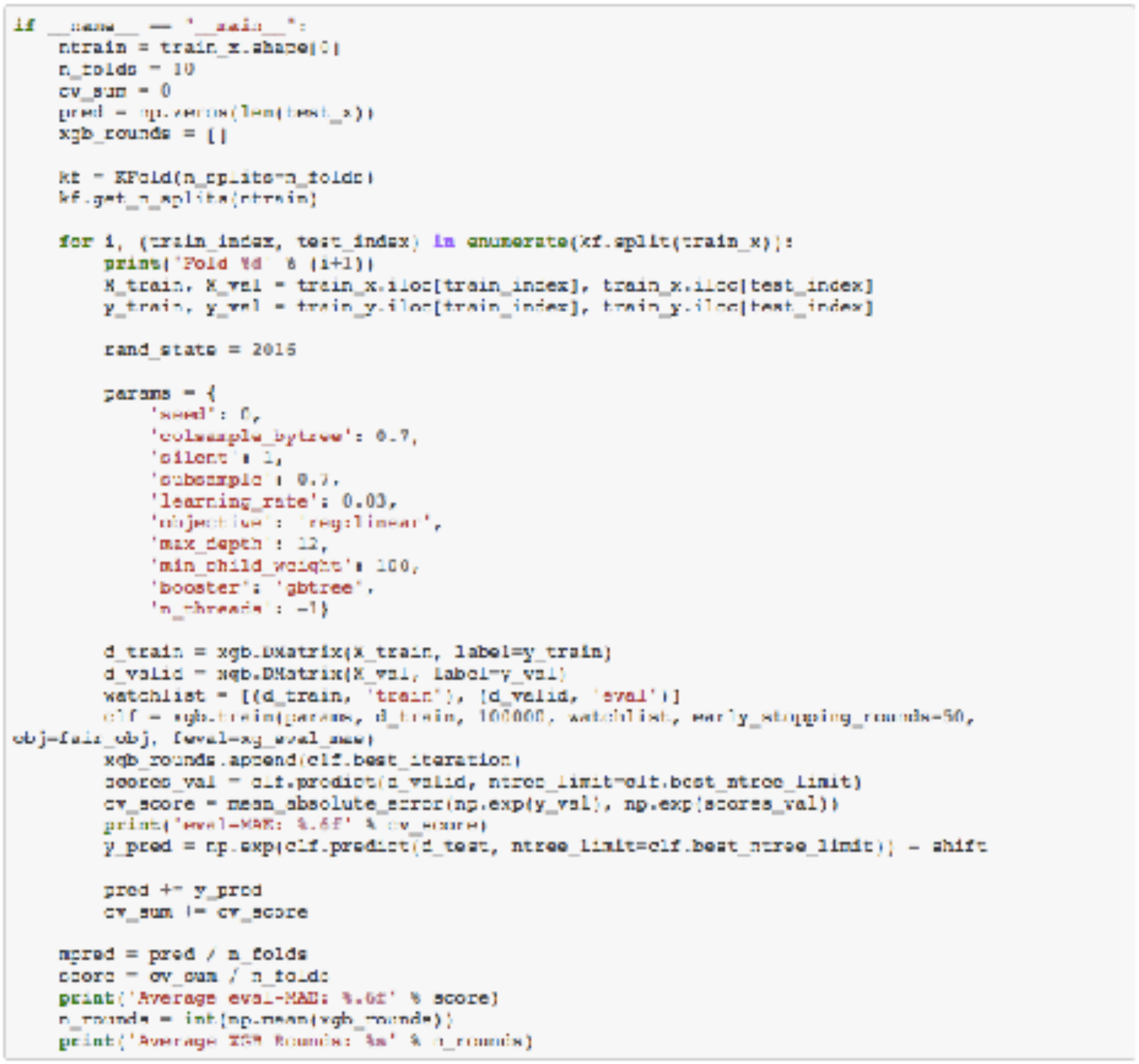
To be used by the XGBoost algorithm, I needed to calculate the first and second derivative of the fair objective function so that XGBoost can make intelligent splitting decisions in its decision trees.



```
def fair_obj(preds, dtrain):
    labels = dtrain.get_label()
    x = (preds - labels)
    den = abs(x) + fair_constant
    grad = fair_constant * x / (den)
    hess = fair_constant * fair_constant / (den * den)
    return grad, hess
```

The custom objective function to be inserted in XGBoost

After that I split the data into 10-folds of training and test sets in order to perform 10-fold cross-validation on the data.



```
if __name__ == '__main__':
    ntrain = train_x.shape[0]
    n_folds = 10
    cv_sum = 0
    pred = np.zeros(len(test_x))
    xgb_rounds = []

    kf = KFold(n_splits=n_folds)
    kf.get_n_splits(ntrain)

    for i, (train_index, test_index) in enumerate(kf.split(train_x)):
        print('Fold %d' % (i+1))
        X_train, X_val = train_x.iloc[train_index], train_x.iloc[test_index]
        y_train, y_val = train_y.iloc[train_index], train_y.iloc[test_index]

        rand_state = 2016

        params = {
            'seed': 0,
            'colsample_bytree': 0.7,
            'silent': 1,
            'subsample': 0.7,
            'learning_rate': 0.03,
            'objective': 'reg:linear',
            'max_depth': 12,
            'min_child_weight': 100,
            'booster': 'gbtree',
            'n_threads': -1}

        d_train = xgb.DMatrix(X_train, label=y_train)
        d_valid = xgb.DMatrix(X_val, label=y_val)
        watchlist = [(d_train, 'train'), (d_valid, 'eval')]
        clf = xgb.train(params, d_train, 100000, watchlist, early_stopping_rounds=50,
obj=fair_obj, feval=xg_eval_mae)
        xgb_rounds.append(clf.best_iteration)
        scores_val = clf.predict(d_valid, ntree_limit=clf.best_ntree_limit)
        cv_score = mean_absolute_error(np.exp(y_val), np.exp(scores_val))
        print('eval-MAE: %.6f' % cv_score)
        y_pred = np.exp(clf.predict(d_test, ntree_limit=clf.best_ntree_limit)) - shift

        pred += y_pred
        cv_sum += cv_score

    mpred = pred / n_folds
    score = cv_sum / n_folds
    print('Average eval-MAE: %.6f' % score)
    n_rounds = int(np.mean(xgb_rounds))
    print('Average XGB Rounds: %s' % n_rounds)
```

The code for model training over 10-folds

Refinement

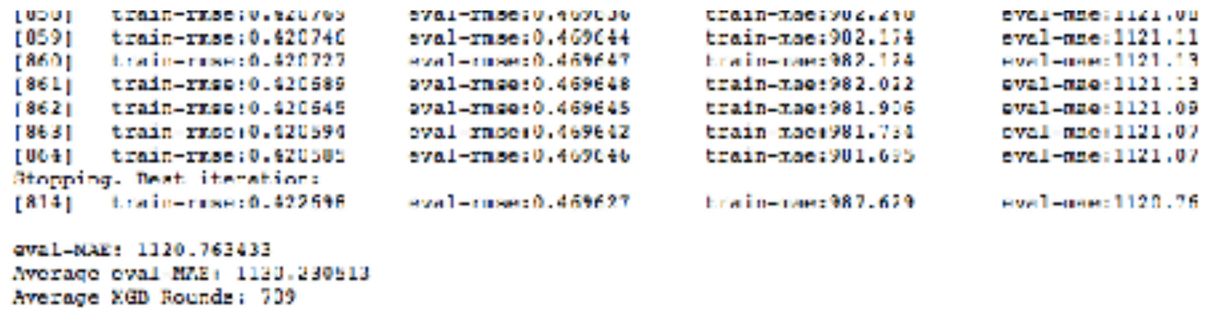
While I made a few refactoring fixes to Msfyre's script, because of the the entire training process sometimes took over 24 hours to complete, I didn't have time to do GridSearchCV or Random Parameter Search and ended up using the same parameters for XGBoost Misfyre had in his script. However, these parameters do indeed make sense:

- seed is the random number seed as isn't an essential param.
- colsample_bytree is the ratio of columns used in training each tree. 0.7 indicates we are using 70% of the columns, which prevents overfitting.

- silent is the parameter set to print messages from the model. 1 means silent mode.
- subsample is the ratio of data we use. 0.7 indicates we are using 70% of the data in each training fold, which also prevents overfitting.
- learning_rate is the incremental step size each tree takes to reduce the loss function. I tried setting this to 0.06, increasing MAE in the cross-validation stage, and 0.015, making it very difficult for any run of XGBoost to converge. Indeed 0.03 seems to be the value that performs best and most efficiently.
- objective is the objective function we are running under the hood. We override it with our "fair_obj" function.
- max_depth is the max number of splits each tree before it stops splitting. 12 splits, while it seems high, does make sense as there are 751 variables and there is a good amount of regularization as detailed by min_child_weight.
- min_child_weight is the minimum hessian value (see "hess" in the fair_obj function) needed in a split to keep splitting down that leaf. 100 is a very high value and should control greatly for overfitting.
- booster is the type of model we are using each XGBoost run. "gbtree" is the default.
- n_threads is the number of cores the algorithm uses. I used "-1" to use as many cores as possible to train the model.

IV. Results

Model Evaluation and Validation



```
[858]   train-rmse:0.420765     eval-rmse:0.469650      train-mae:982.210       eval-mae:1121.08
[859]   train-rmse:0.420740     eval-rmse:0.469644      train-mae:982.174       eval-mae:1121.11
[860]   train-rmse:0.420727     eval-rmse:0.469647      train-mae:982.124       eval-mae:1121.13
[861]   train-rmse:0.420589     eval-rmse:0.469648      train-mae:982.022       eval-mae:1121.13
[862]   train-rmse:0.420545     eval-rmse:0.469645      train-mae:981.936       eval-mae:1121.09
[863]   train-rmse:0.420594     eval-rmse:0.469642      train-mae:981.734       eval-mae:1121.07
[864]   train-rmse:0.420505     eval-rmse:0.469646      train-mae:981.695       eval-mae:1121.07
Stopping. Best iteration:
[814]   train-rmse:0.422598     eval-rmse:0.469627      train-mae:987.679       eval-mae:1120.76

eval-MAE: 1120.763433
Average eval MAE: 1123.230513
Average XGB Rounds: 739
```

Final results of model training

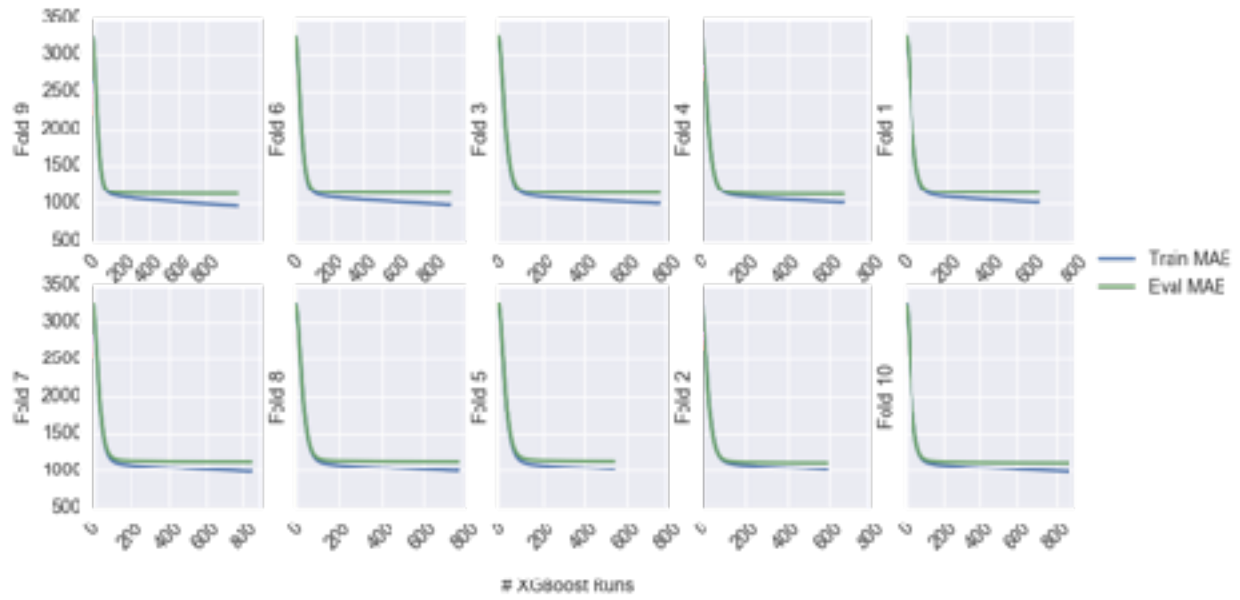
The final results of the model had an average MAE of 1130.230513 on the test sets of the 10 folds used in 10-fold cross-validation. Each fold had about 709 trees trained and the resulting models from each fold were used to predict on the test set. The results of the 10 predictions were averaged and submitted to Kaggle. As a test for robustness (to ensure that this isn't overfitting on the training set), the test set predictions achieved a score of 1107.63216 on the public leaderboard of Kaggle, compared to 1108.72665 on Misfyre's script. Misfyre's script also have an average MAE of 1131.63 on its cross validation sets.

Justification

While a 1.4 point decrease in average MAE may not seem very exciting, it led to a a 1.09 decrease in MAE on the test set which was led to a 62 point increase in my ranking (666 to 604). The highest ranked individual has a test MAE of 1096.92533 on the public leaderboard. Indeed, in machine learning, much of the work is focused on improving the model by a fraction more. Therefore, I would say that my model, which increased performance by 0.1% was successful.

V. Conclusion

Free-Form Visualization



The above are graphs of MAE in training and cross-validation sets for each of the 10 folds. As we can see, train MAE and eval MAE both decrease rapidly and then plateaued, with train MAE still decreasing and eval MAE decreasing ever so slightly.

Reflection

In this project, I loaded both training and test sets, preprocessed the continuous and categorical variables, created clustering and PCA features, implemented a custom objective function in my XGBoost model, which was trained over 10 folds of the training data.

I thought the most interesting part of this project was using a custom objective function in XGBoost to minimize for MAE. I had never used a custom objective function and learning the theory behind it and appreciating XGBoost's capability to utilize custom objective functions was a pleasurable experience.

The most difficult part was training the model itself because of the technological limits of my computer (2013 MacBook Air) and the size of the dataset (1.8 GB training). It was frustrating having finished training all 10 folds and then having to close my computer or having a bug appear at the end of the code.

Overall, I am pleased with the results of my model which improved upon the benchmark model using the supervised and unsupervised learning methods I learned during the course of this Nanodegree.

Improvement

Some improvements I could have made to model are to use ensembling and categorical embedding. Ensembling has been a crucial key to all the top three results by Bishwarup B, Alexey Noskov and Faron and I attempted to use it but with limited success, mainly because training and several models took far too long. In the future, I will consider using a service like AWS that allows for cloud computing or getting a fast computer. Furthermore, categorical embedding is a technique that has greatly improved the performance of many of these algorithms. Had I a better understanding of these algorithms, I would have utilized them, or even used PCA to create new features from all the categorical features. One of the issues of the latter approach, however, was that the Jupiter Notebook crashed each time I attempted to perform PCA on all the categorical variables in the model.