CS 145 Project 2C
Liberty Mutual Group: Property Inspection Prediction

Project Objective
Predict the hazard scores using a dataset of property information

Team Members

203588671 Aaron Tang
304743326 Andrew Lin
303913013 Erick Ruiz
904063131 Ryan Trihernawan
004464072 Stacy Miller

(i) Instructions

1. Go to Kaggle Kernel: https://www.kaggle.com/c/liberty-mutual-group-property-inspection-prediction/kernels/scripts/new
2. Provide a title for the script on Kaggle Kernel before you can run it
3. Make sure to select Python under the drop-down list above the code editor
4. Remove the default code in the code editor
5. Open code.py in the same directory as the report
6. Copy and paste the code to the code editor
7. Click the "**Run**" button to the right of the drop-down menu
8. After the script finishes running, go to the right of the code editor and click the "**Submit to Competition**" button next to the title of the csv file
9. You should obtain the result of **0.384358**

(ii) Explanation of Applied Techniques
We used Python for our project. Looking at the training data, we observed that the problem is a regression problem. The label i.e. hazard score that we had to predict is not binary and instead it ranges from 1 to 69. The variables are anonymized, hence we could not perform feature engineering to omit less relevant variables or merging less relevant variables when teaching the model. Moreover, we noticed that half of the variables or 16 of them are categorical. Since the Python libraries for the machine learning models we used cannot learn from categorical variables, we had to convert the categorical variables to numerical variables. We used Scikit-learn's DictVectorizer to do so. The 16 categorical variables were converted to 95 vectorized numerical variables. Then we merged the normalized numerical data (16 variables) with the vectorized numerical data (95 variables) to obtain our final data with which we teach

our machine learning models. Through our research on Google, we discovered several Python libraries to help us do our experiments. They are as follows:

- Pandas: library for executing operations on the csv data
- Scikit-learn: library for various kinds of machine learning models
- XGBoost: high-performance gradient boosting library

(iii) & (iv) Experimental Results and Analysis

We experimented with four different regression algorithms to find the one that gives us the highest score on Kaggle. For each of the algorithm, we tuned its parameters to fit our training data while ensuring to minimize underfitting and overfitting. Below are the results and analysis for each of the algorithm and also a feature engineering we did in between.

Algorithm I: Linear Regression

Linear regression is one of the most commonly used regression algorithms, therefore it made sense to running our data through this algorithm first. The algorithm tries to find a linear relationship between multiple independent variables and one dependent variable, so that future values can be predicted for the dependent variable. To test the algorithm's performance on our data, we used the LinearRegression function provided by scikit-learn. With default parameters, the model gave a score of 0.328440. LinearRegression from scikit-learn has 2 main tunable parameters: fit_intercept and normalize. Toggling these parameters had little effect on the score, only lowering the score by < 0.0001. This means that for linear regression, our best score is the one given by using the default parameters, which was a relatively low score.

The low score returned by the LinearRegression model can be attributed to several weaknesses of the algorithm. Linear regression is limited in that it assumes that the dependent is linearly related to the independent variables, even though the data may in fact have a non-linear relationship. The algorithm weighs all independent variables equally in predicting the output, and is sensitive to outliers which may skew the data. Furthermore, it assumes all input variables to be independent of one another. For these reasons, we needed to try running the data through other regression algorithms.

Algorithm II: K-Nearest Neighbors

K-nearest neighbors is an algorithm used for classification based on predicting a point's label by finding the closest distance to the training samples. To test using k-nearest neighbors we used KNeighborsClassifier from scikit-learn. The default variables used for this algorithm was 5 neighbors and 30 leaves for the generated tree. After running k-nearest neighbors, the score we received was 0.054337.

In an attempt to improve the score, some of the variables were further tweaked.

n_neighbors: Number of neighbors to analyze
leaf_size: Leaf size of the generated BallTree or KDTree

The leaf size was increased to 40 and the neighbors was increased to 7. This boosted the score to 0.059569. Attempts at increasing neighbors and leaf size even further resulted in the same/very similar score (never exceeding 0.05). Unfortunately, K-nearest neighbors did not produce an acceptable score. One possible explanation is that the training data could not be clearly categorized and had a lot of overlapping parameters. Due to the low score and fairly extensive runtime for this algorithm, further tweaking was not justified.

Feature Engineering: PCA reduction

We decided at this point that perhaps there were too many variables in play and perhaps doing a reduction in variables will help. The categorical data could not be easily reduced with sci-kit's PCA. Even then, the PCA analysis would give you the covariance of the variables to see if any were more valuable than others. We found, when running PCA analysis on the numerical variables that most of them were around the <0.01 range. This method quickly showed use that this data would be not be able to go without all the variables. We didn't have to continue from here, but we decided to reduce the very negligible variables and try out gradient boosting which yielded worse results. It was clear that we had to tune either Random Forest or Gradient Boosting to receive better performance.

Algorithm III: Random Forest

Random forest is an ensemble algorithm which builds multiple decision trees with random samples of observations and attributes, and makes a final prediction base on each result. The idea behind random forest is to obtain a more precise and stable prediction by suitably combining a number of base-learners instead of inducing one strong learner. We used the RandomForestRegressor from scikit-learn library. The default set of parameters gave a very low accuracy of 0.267059. So we performed cross validation on 4 main parameters within ranges as follows:

1. n_estimators (number of decision trees): 100, 200, 300, 400, 500
2. max_depth (maximal depth of each decision tree): 20, 30, 40, 50
3. max_features (maximal percentage of features to consider): 0.2, 0.4, 0.6, 0.8, 1
4. min_samples_leaf (minimal number of samples in a leaf node): 20, 30, 40, 50

To avoid getting different outcomes with the same set of parameters, we set the random seed parameter random_state to 50.

As a result, the set of parameters below gave us the highest accuracy of 0.378159 over other combinations:

1. n_estimators: 400
2. max_depth: 40
3. max_features: 0.8
4. min_samples_leaf: 30

Algorithm IV: Gradient Boosting

Initially we used scikit-learn's gradient boosting model (GradientBoostingRegressor) and we discovered that it outperformed other algorithms even with its default parameters. The model gave a score of 0.367691. One of the YouTube videos on machine learning that we watched suggested XGBoost library for a better performing gradient boosting model. However, the model gave a slightly lower score of 0.366956. Thus we tuned the parameters for both of the models and compared them. For each of the model, we used scikit-learn's GridSearchCV, which exhaustively considers all user-provided parameter combinations. Since scikit-learn's gradient boosting model is implemented differently than the XGBoost's model, they do not completely share their parameters. For the former, we performed five parameter tests as follows:

1. n_estimators: 50, 60, 70, 80, 90, 100, 110, 120, 130, 140, 150
2. max_depth: 5, 7, 9, 11, 13, 15
   min_samples_split: 200, 400, 600, 800, 1000
3. min_samples_leaf: 30, 40, 50, 60, 70
4. max_features: 7, 9, 11, 13, 15, 17, 19
5. subsample: 0.6, 0.7, 0.75, 0.8, 0.85, 0.9

Our tests provided us the most optimal parameters for the model in learning the training data as follows:

1. n_estimators: 100
2. max_depth: 7
   min_samples_split: 400
3. min_samples_leaf: 40
4. max_features: 15
5. subsample: 0.8

We tried different learning rate and the default one i.e. 0.1 turned out to be the most optimal for our model and training data. For the XGBoost's model, we performed ten parameter tests as follows:

1. max_depth: 3, 5, 7, 9
   min_child_weight: 1, 3, 5

2. max_depth: 4, 5, 6
   min_child_weight: 0, 1, 2
3. min_child_weight: 2, 4, 6, 8, 10, 12
4. gamma: 0, 0.1, 0.2, 0.3, 0.4
5. subsample: 0.6, 0.7, 0.8, 0.9
   colsample_bytree: 0.6, 0.7, 0.8, 0.9
6. subsample: 0.85, 0.90, 0.95
   colsample_bytree: 0.65, 0.70, 0.75
7. reg_alpha: 1e-5, 1e-2, 0.1, 1, 100
8. reg_alpha: 80, 90, 100, 110, 120
9. reg_alpha: 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95
10. reg_alpha: 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90

Our tests provided us the most optimal parameters for the model in learning the training data as follows:

1. max_depth: 5
2. min_child_weight: 2
3. gamma: 0.2
4. colsample_bytree: 0.65
5. subsample: 0.85
6. reg_alpha: 84

We tried different combinations of learning rate and estimators, and we obtained 0.1 and 310 to be the most optimal learning rate and estimators, respectively. Running both models with their most optimal parameters yielded 0.376872 and 0.384358 for scikit-learn's GradientBoostingRegressor model and XGBoost's model, respectively. Therefore we picked XGboost's model as our final model.

Conclusion

Based on our experiments, we discovered that Gradient Boosting outperforms other algorithms especially Random Forest due to some reasons explained below. First of all, Random Forest is more likely to overfit than Gradient Boosting. Secondly, Gradient Boosting uses "weak-learners" by default, while Random Forest does not. However, it is possible to tune the parameters of Random Forest so that it uses "weak learners". Essentially, each tree uses the previous result to adjust weights on the data for the next tree. Since Random Forest grows its trees in parallel, its final result is simply the weighted average of all the estimators or trees. On the other hand, since Gradient Boosting grows its trees sequentially, its final result is the result of its final tree.