# CSCI 567 Group 50 Final Project Report

**Brian Nlong Zhao**
University of Southern California
briannlz@usc.edu

**Ozan Erdal**
University of Southern California
erdal@usc.edu

**Yicheng Li**
University of Southern California
yli63904@usc.edu

## 1 Introduction

Richter's Predictor: Modeling Earthquake Damage [1] is a Machine Learning competition hosted by DrivenData. The objective of the competition is to build a model to predict the damage level of a building given the features of the building such as age, floor, position, etc. The value we are trying to predict for the variable `damage_grade` has three grades of damage:

- `1` represents low damage
- `2` represents a medium amount of damage
- `3` represents almost complete destruction

**Dataset** The given dataset consists of a training set and a test set. In both splits, there are 38 columns of features and 1 column for `building_id`. The training set gives the corresponding `damage_grad` for the examples, and our model is supposed to learn the relationship between features and `damage_grad` from the training set, and do inference on the test set, where the `damage_grad` is not given to us.

**Metrics** To rank model performance, we as a team will need to submit the inferred `damage_grad` on the test set and submit it to the competition website for evaluation. The evaluation metric of the competition is F1 score, which is calculated as follows:

$$F_{micro} = \frac{2 \cdot P_{micro} \cdot R_{micro}}{P_{micro} + R_{micro}}$$

where

$$P_{micro} = \frac{\sum_{k=1}^{3} TP_k}{\sum_{k=1}^{3} (TP_k + FP_k)}$$

$$R_{micro} = \frac{\sum_{k=1}^{3} TP_k}{\sum_{k=1}^{3} (TP_k + FN_k)}$$

and $TP$ is True Positive, $FP$ is False Positive, $FN$ is False Negative, and $k$ represents each possible value of `damage_grad`.

[1] https://www.drivendata.org/competitions/57/nepal-earthquake/page/134/

## 2 Data Preprocessing

**Normalization** We have tried the standard scaling method, where the features are standardized by removing the mean and scaling to unit variance based on the following equation

$$x' = \frac{x - \mu}{s}$$

where $\mu$ is the mean of the feature among all samples, and $s$ is the standard deviation of the feature among all samples. This approach, however, does not have prominent effects on tree or ensemble-based methods as they are not sensitive to the variance of features. Our final solution is an ensemble-based method, therefore we omit the standard scaling approach. Instead of scaling, we apply an outlier removal method. Specifically, we calculate the 0.05 percentile and the 0.95 percentile for each feature among all samples, as well as an inter-percentile range by subtracting the 0.05 percentile from the 0.95 percentile. We set the lower and the upper threshold as:

$$th_{lower} = q_{0.05} - 1.5 \cdot (q_{0.95} - q_{0.05})$$
$$th_{upper} = q_{0.95} + 1.5 \cdot (q_{0.95} - q_{0.05})$$

where $q_{0.05}$ and $q_{0.95}$ are the values corresponding to the 0.05 and the 0.95 percentile respectively. With the upper and lower thresholds, we note the values that smaller than the lower threshold or larger than the upper threshold as outliers, and we cap the outliers with the thresholds.

**Encoding** For the non-numerical categorical features, since there is no ordinality among the letter representations, we encode them using one-hot encoding. When we were using models that have the functionality to take in categorical feature and automatically encode it, such as CatBoost, we naturally omit this encoding step. For the categorical data that are already integer-encoded, we leave them as is since they already encode the ordinality. For the labels, we mapped them to 0-based.

**Missing Data** The dataset has no missing entries, so no action is taken regarding missing data handling.

**Feature Selection** We have tried multiple feature selection methods to remove features that could potentially harm accuracy. This only applies to numerical data. We have tried selecting by variance threshold, where numerical features with variance smaller than a preset threshold will be removed. Another approach we have tried is to select features by univariate statistical tests. We set a preset value $k$ and the top $k$ features with the highest scores will be kept. However, based on several sets of experiments on different thresholds and $k$s on different models, we found that selecting a subset of features does not help or even harm accuracy in some cases. As a result, we don't use any feature selection strategy and use all the features in our final solution.

**Validation Set** To evaluate our model performance, we randomly split 1/3 of the training data as a validation set. The number 1/3 is chosen so that the size of the validation set matches the actual test size. After we evaluate the model performance on the validation set, we will retrain the selected model on all original training samples before making inferences on the test set.

**Upsampling** We observe that the training dataset is unbalanced on the labels. There are 98916 samples have `damage_grade` 2, 58063 samples have `damage_grade` 3, and 16754 samples have `damage_grade` 1. We have tried to upsample the minorities, i.e., resample the samples with `damage_grade` 1 and 3, to make it a nearly uniform label distribution. However, we found this approach does not help improve validation/test accuracy when the distribution of the training set is different from that of validation/test set.

## 3 Learning Algorithms

**Model Selection** With the data preprocessing techniques applied, we have tried several basic learning algorithms including decision tree, random forest, SVM, and simple multilayer perceptron, however, none of them gives promising initial accuracy on the validation set, usually below 0.70 F1 score. We found that popular boosting algorithms, including AdaBoost, LightGBM, CatBoost, and XGBoost, generally give better initial results. The following table shows some results of different learning algorithms we have tried using default hyperparameters on a random validation set.

| Model | F1 Score |
|---|---|
| MLP | 0.6478 |
| Decision Tree | 0.6544 |
| Linear SVM | 0.6561 |
| AdaBoost | 0.6578 |
| Random Forest | 0.7132 |
| XGBoost | 0.7280 |
| CatBoost | 0.7305 |
| LightGBM | 0.7361 |

**Hyperparameter Tuning** With some manual hyperparameter tuning, we were able to achieve a final F1 score of about $0.74 \pm 0.003$ on the test set. The best F1 score we get at this point is 0.7431 using CatBoost. We later relied on automatic hyperparameter sweep pipeline. The XGBoost gives the best result in a series of experiments. Although the XGBoost classifier performs well in the F1 score, the F1 score is still not high enough, so we use the RandomizedSearchCV function in scikit-learn to perform a random search to optimize the hyperparameters of the XGBoost classifier. Hyperparameter tuning aims to find the best hyperparameter set that can maximize the classifier's performance on the test set. Based on the dataset, the problem to solve, previous experience in Machine Learning, and continuous practice with the selected parameters, we choose the following 6 hyperparameters to tune:

- `n_estimators`: Determines the number of trees in the model. Higher numbers can improve model performance, but may also increase the risk of overfitting and training time.
- `max_depth`: Determines the maximum depth of each tree in the model. A higher maximum depth allows the model to capture more complex patterns in the data, but may also increase the risk of overfitting and training time.
- `learning_rate`: Determines the step length of each iteration in the upgrading process. A lower learning rate can improve the generalization ability of the model, but more iterations may be needed to converge.
- `subsample`: Determines the proportion of training data set used to train each tree. A lower value of subsample can improve the generalization ability of the model, but may also increase the risk of overfitting and training time. Generally setting this parameter to a value less than 1 helps prevent overfitting by reducing correlations between individual trees.
- `colsample_bytree`: Determines the proportion of columns used to train each tree. A lower `colsample_bytree` improves the generalization of the model, but may also reduce its performance.
- `min_child_weight`: The sum of the minimum instance weights required among child nodes. Increasing this parameter can help reduce overfitting by requiring a minimum number of training instances per leaf node. Conversely, a higher `min_child_weight` may reduce the risk of overfitting, but may also increase model bias.

These hyperparameters were probably chosen because they have been shown to have a significant impact on the performance of XGBoost classifiers. The range of each hyperparameter is chosen based on the empirical results of a number of similar field applications and previous experiments. The specific range of values for each hyperparameter may be selected wide enough to include a wide range of values that are likely to be optimal for the model, while also narrow enough to reduce the number of iterations required during hyperparameter tuning. Therefore, the value range of the following hyperparameter pairs is selected through continuous experiment interval and parameter size:

- `n_estimators`: This refers to the number of trees to be built in the integration. In this case, the model will be trained using a range of 980 to 1020 trees, increasing by one at each step. The range of values for `n_estimators` is chosen based on previous experience and experiments in order to find a balance between model performance and overfitting risk.
- `max_depth`: This refers to the maximum depth of each decision tree in the integration. In this case, the model will be trained at a maximum depth of 8 or 9. The `max_depth` range is selected based on the complexity of the problem and the size of the data set.

- `learning_rate`: This refers to the step shrinkage used in each lifting iteration to prevent overfitting. In this case, the model will be trained using values in the range 0.11 to 0.15, with 31 equally spaced values. The value range of `learning_rate` is selected based on previous experiments to find the best learning rate that balances model performance and convergence speed.
- `subsample`: This refers to the proportion of samples used for each iteration of promotion. In this case, the model's training range is from 0.82 to 0.93, with 111 equally spaced values. The value range of the `subsample` is selected based on the size of the data set and the complexity of the problem.
- `colsample_bytree`: This refers to the proportion of features to be used in each iteration of promotion. In this case, the model's training range is 0.46 to 0.56, with 101 equally spaced values. The range of values for `colsample_bytree` was chosen based on previous experiments to find the best ratio of columns to balance model performance and generalization ability.
- `min_child_weight`: This refers to the minimum sum of instance weights required among the child nodes (hessian). In this case, the model's training ranges from 3.2 to 3.8, with 61 equally spaced values. The value range of `min_child_weight` was selected based on previous experiments to find the optimal value to balance model bias and variance.

Finally, by adjusting the hyperparameter, we get the optimal value:

```
{
    "subsample": 0.858,
    "n_estimators": 999,
    "min_child_weight": 3.5,
    "max_depth": 8,
    "learning_rate": 0.137999999999998,
    "colsample_bytree": 0.499
}
```

By putting the optimal hyperparameter into the XGBoost classifier, we get the train F1 score of 0.8275. The test F1 score is 0.7500.

## 4 Conclusion

In order to get a good estimator for the label, we first experiment with a variety of learning algorithms with default parameters. We select some of the algorithms that give good performance on a random validation set that has the same size as the test set. We further perform some manual hyperparameter tuning to see how much the model could be improved, and finally, we perform an automated hyperparameter sweep using a grid search scheme. Our final best F1 score is 0.7500 obtained by using XGBoost as the learning algorithm.

## 5 Running the Code

We provide the final camera-ready copy of our code in an Interactive Python notebook (.ipynb) file. To reproduce the result just run the cells sequentially. The whole pipeline of hyperparameter sweep takes about 2 hours to execute.