CSCI567 Final Project Report

Bumsik Kim
USC ID: 5162632967
bumsikki@usc.edu

Dongwon Jung
USC ID: 3118988459
dongwonj@usc.edu

Abstract

The final project focuses on predicting the level of damage caused to buildings during the 2015 Gorkha earthquake in Nepal based on building location and construction aspects. Specifically, it aims to develop a predictive model that can predict the ordinal variable damage grade, which indicates the severity of damage to the building, categorized into three grades: 1, 2, and 3. The experimental results show that the ensemble stacking method on XGBoost, LightGBM, and CatBoost models outperforms all the other attempts in the project period.

1 Problem Definition

In April 2015, a 7.8-magnitude earthquake hit the Gorkha district of Gandaki Pradesh, Nepal. The Nepalese government generated the post-disaster dataset with the help of the National Planning Commission. Our goal is to predict the ordinal variable *damage_grade*, which represents the level of damage to the buildings that was hit by the 2015 Gorkha earthquake in Nepal, given the post-disaster dataset.

2 Dataset

The dataset contains primarily details about buildings' structure and legal ownership. There exist three grades of damage, 1, 2, and 3. Each represents low damage, medium damage, and almost complete destruction, respectively. Each row corresponds to a particular building in the area impacted by the Gorkha earthquake, with their unique identifier, *building_id*.

2.1 Categorical and Numerical Features

There exist 38 columns in this dataset, each column representing the feature of the data. Among 38 features, there are 11 categorical features and 27 numerical features.

One of the key categorical features in the dataset is *geo_level*, which represents the geographic region in which the building exists, from the most significant (level 1) to the most specific sub-region (level 3). The rest of the categorical features are mainly relevant to the type of foundation, ground, or roof.

Age is the essential feature among numerical features, which represents the age of the corresponding building in years. The numerical features in the dataset are mainly flag variables, for example, a variable that informs whether the building material has been used for the corresponding building or whether the building was used for certain public or government offices.

2.1.1 Feature Importance

The process of feature importance analysis involves identifying the input variables or features that have the most significant impact on a model's output. Doing so makes it possible to determine which variables are crucial for achieving optimal performance. CatBoost's *feature_importance* function

Submitted to 36th Conference on Neural Information Processing Systems (NeurIPS 2022). Do not distribute.

allows us to interpret the feature importance by calculating the contribution of each feature to the model's performance. The feature importance score is determined by summing up the improvement in the loss function across all the trees within the model. Table 1 is the table of the top 5 features that have the highest importance. Based on the mentioned table, we observe that the features that have impacted the model's performance the most are features with the geographic regions in which the building exists.

Feature	Importance(%)
geo_level_2	24.10
geo_level_1	15.49
geo_level_3	14.67
age	7.09
foundation_type	5.28

Table 1: A table of top 5 features with the highest feature importance

2.1.2 Test Dataset and Unseen Categorical Variable

As mentioned in section 2.1.1, we know that *geo_level_2*, *geo_level_1*, and *geo_level_3* are the most relevant features to the accuracy of the prediction from the model. Based on this fact, we have focused on the values of these features from the test dataset. We have observed some unseen values of *geo_level_2* and *geo_level_3* in the test dataset compared to the ones from the training dataset. We assume that handling unseen values before training will improve the model's accuracy. The detail of this process will be explained in section 4.1.

3 Decision Tree Algorithms and Boosting

The decision tree algorithm is a supervised learning model that recursively partitions the input data into subsets based on a series of decision rules until a prediction can be made. The algorithm constructs its decision rules in the following consequences. The algorithm assesses each data feature and picks the one that yields the highest information gain, reducing uncertainty related to the target variable. This feature is subsequently used as the decision rule for the current node in the tree.

Considering the nature of the dataset, where most of the features are categorical, we have judged that the decision tree algorithm is the best fit to achieve our goal. Furthermore, we have chosen to use a gradient-boosting technique to achieve our goal with higher accuracy. Boosting involves combining multiple weak models to create a single robust model, where each weak model is trained on a subset of the data. The technique is named gradient-boosting because it uses gradient descent optimization to train the weak model.

In this section, we discuss the gradient-boosting models we have adopted, the three gradient-boosted tree algorithms: XGBoost, LightGBM, and CatBoost.

3.1 Extreme Gradient Boosting(XGBoost)

XGBoost Chen and Guestrin [2016] is the gradient-boosting algorithm that ensembles multiple weak decision trees to obtain a robust predictive model. XGBoost addresses the overfitting problem of the decision tree by incorporating L1 or L2 regularization. Moreover, XGBoost can compute large-scale datasets efficiently using parallel computing techniques. When training the XGBoost model, it is essential to tune the best hyperparameters, such as the learning rate, number of trees, and the maximum depth of each tree, to optimize the performance.

XGBoost uses a technique called "one-hot encoding" to handle categorical data. One-hot encoding converts each categorical feature into a set of binary features, where each feature represents a unique value of the original categorical feature.

3.2 Light Gradient Boosting(LightGBM)

LightGBM Ke et al. [2017] is one of the ensemble methods like XGBoost, but it has a crucial difference. While XGBoost uses a set of decision trees to make predictions, LightGBM uses a

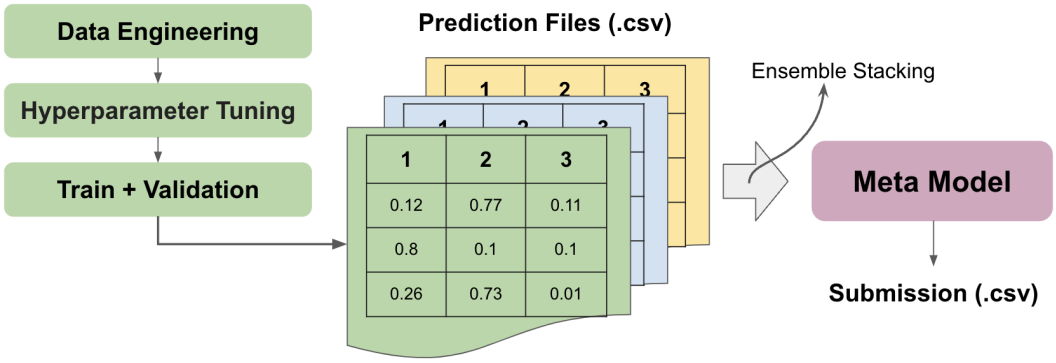

Figure 1: Overall architecture of model pipeline. Each color (green, blue, and yellow) represents the base model pipeline. First, obtain prediction files of base models containing each class's probability. Then, stack the predictions and feed them into a meta-model. Finally, retrieve the final result as a submission.

histogram-based algorithm to find the best split. The histogram-based algorithm allows the LightGBM to require fewer calculations and memory usage than the greedy algorithm from XGBoost.

LightGBM uses a technique called "gradient-based one-side sampling" (GOSS) to handle categorical data more efficiently than the traditional one-hot encoding used in other algorithms like XGBoost and "exclusive feature bundling" (EFB) to group similar categorical features together, consecutively.

GOSS is a two-step sampling process used by LightGBM that initially samples data points with large gradients to avoid overfitting and then selectively samples data points with categorical features to decrease the dataset's dimensionality. The second stage of GOSS is governed by a statistical criterion that seeks to retain only the most informative categorical features while discarding the rest.

Subsequently, LightGBM employs an approach called "exclusive feature bundling" (EFB) to combine related categorical features. This technique enhances the algorithm's efficiency by minimizing the number of categorical features that necessitate splitting during tree construction.

3.3 CatBoost

When using XGBoost or LightGBM on the dataset with categorical features, we have to preprocess the data in a way that encodes categorical variables into numerical variables before training. The critical difference between CatBoost Prokhorenkova et al. [2018] and the above algorithms is that CatBoost has a built-in categorical feature handling mechanism called Symmetric Binary Sparsity Tree (SBST), which eliminates the need for converting categorical variables into numerical variables. SBST algorithm in CatBoost identifies combinations of features that can effectively split the data. These combinations are characterized by their presence in a significant portion of the data points in one group and rarity in the other.

4 Solution

In this section, we discuss the model's overall pipeline and several attempts we devised to arrive at the ultimate solution. The overall pipeline of the model is depicted in Figure 1.

4.1 Data Engineering

The first step is to transform data into a suitable format for our models. We split data into two parts; numerical and categorical. Since we use gradient boosting-based models, numerical feature normalization is not needed. Also, we utilize the innate functionality of the models to handle categorical features, as discussed in section 3.

As we mentioned in section 2, two categorical features, *geo_level_2_id* and *geo_level_3_id*, have some values that only appear in the training set and not in the test set. For *geo_level_2_id*, we fill in the unseen values in the test set with the dominant value with the same *geo_level_1_id* in training data. Likewise, for *geo_level_3_id*, we fill in the unseen values in the test set with the dominant

value with the same *geo_level_2_id* in training data. This method ensures the locality information of the features that *geo_level_1_id* geographically contains *geo_level_2_id* and *geo_level_2_id* geographically contains *geo_level_3_id*. If we naively replace unseen values with the most frequent values in the training set, the locality information is not guaranteed and is lost, thus leading to performance degradation.

4.2 Hyperparameter Tuning

We use Optuna Akiba et al. [2019] to find the optimal hyperparameters. Appendix A summarizes the hyperparameters that are tuned. We set the number of trials to 20. The hyperparameters that we tuned are listed below. We fix other hyperparameters as the default values.

- learning_rate: learning rate of training.
- max_leaves: number of decision leaves in a single tree.
- max_depth: maximum depth of a single tree
- min_data_in_leaf: minimum number of observations in a leaf.
- min_child_weight: minimum sum of weight in a leaf.
- l2_leaf_reg: L2 regularization term of the cost function.

Two hyperparameters, max_leaves and max_depth, control the complexity of the model. The model is prone to overfitting if the number of leaves are too much or depth is too deep. It is crucial to control the values to avoid overfitting. Also, min_data_in_leaf and min_child_weight are important to tune since they control the number of observations in a leaf. If the amount is too small, the model is likely to overfit the training data.

4.3 Train and Validation

We train the model using the optimal hyperparameters that we got from Optuna. We set n_estimators, equivalent to the number of iterations, to 10000 and apply early stopping after 100 steps of no improvement. To apply early stopping, we use 5-fold cross-validation using scikit-learn library called StratifiedKFold. StratifiedKFold manages the imbalance of the class labels by preserving the percentage of samples for each class. This is helpful for our model since the class labels of training data are skewed. For the evaluation metric, we define a custom F1 metric function, which is the metric that we test on.

For each fold, we make predictions on validation and test data that contain the probability of each class. We make predictions on validation data because if we combine all the validation predictions of all the folds, it makes up the predictions for the whole training data, which we need for ensemble stacking in the future. Similarly, we need the prediction of test data because we need it to feed into the ensemble meta-model to get the final result for the submission. As a result of the training, we gain two prediction files for training and test data.

4.4 Ensemble Stacking

Ensemble stacking is utilized to enhance the accuracy of predictive models by integrating the outputs of multiple models. Specifically, base models are trained on the same dataset, and their predictions are aggregated through a meta-model to produce the final prediction. The meta-model is trained on the outputs of the base models, allowing it to learn how to optimally combine the predictions to improve overall performance. This approach can effectively alleviate the weaknesses of each base model.

In order to perform ensemble stacking, we need prediction files consisting of the probability of each class of training and test data. Here, since we utilize three base models, CatBoost, XGBoost, and LightGBM, we have six prediction files, training and test prediction file for each model. We aggregate the training prediction files of each model into a single prediction file. For example, each data point in a prediction file has three features representing the probability of three classes. If we aggregate three prediction files, we have a single prediction file with nine features. Thus, the shape of a single prediction file is [*number_of_data_points*, 9]. Similarly, we also aggregate test prediction files for future prediction.

Models	Hyperparameter Tuning	F1-score (micro)
CatBoost	×	0.7492
XGBoost	×	0.7416
CatBoost	✓	0.7493
LightGBM	✓	0.7500
CatBoost + Pseudo-labeling	✓	0.7498
LightGBM + Pseduo-labeling	✓	0.7500
CatBoost + LightGBM	✓	0.7516
CatBoost + LightGBM + Pseudo-labeling	✓	0.7515
CatBoost + LightGBM + XGBoost + Pseudo-labeling	✓	0.7521
CatBoost + LightGBM + XGBoost	✓	**0.7523**

Table 2: The experiment results. The results are divided into three sections: single model, single model + pseudo-labeling, and ensemble stacking + hyperparameter tuning. Note that ensemble stacking without pseudo-labeling achieved the hightest score.

For the meta-model, we select the logistic regression model. The role of the meta-model is to perform a weighted average of the prediction results of the base models. Since the task is a classification, logistic regression is suitable for our task. The meta-model learns the linear combination of each base model's prediction results and predicts the aggregated test prediction to get the final result.

4.5 Pseudo-labeling

Pseudo-labeling is used to increase the size of the training data by generating additional labeled data from unlabeled test data. The basic idea behind it is to use a model to make predictions on unlabeled test data and to treat the predictions that surpassed a certain threshold as the ground truth labels. In our experiment, we set the probability threshold to 0.8, indicating that the prediction confidence should be higher than 0.8 to be considered a pseudo-label. Pseudo-labeling enhances the performance of a single model without any ensemble method. However, the performance degrades when pseudo-labeling is utilized in the ensemble pipeline. Thus, we do not integrate pseudo-labeling in the final model pipeline.

5 Results

We report the experimental results that we tried through the project and discuss several observations gained from the experiments. Table 1 summarizes the experimental results. Here are the observations.

- Even though hyperparameter tuning increases the performance, it does not contribute as much as we expected. For example, by tuning the hyperparameters for CatBoost, we have 0.0001 gain of F1-score. This is mainly because we were not able to explore the hyperparameter space as much as we should have. The trial, which was set to 20, was too small.
- As mentioned in section 4.5, pseudo-labeling helps boost performance when using a single model but degrades performance when more than one models are ensembled. One possible explanation for this is that treating unlabeled data as labeled data based on the result of incomplete base models can introduce misleading noise to the data that the meta model will be trained on.
- Applying more models in the ensemble stacking enhanced the score monotonically. This proves that the three models were well ensembled to provide better results than the results gained from a single model alone.

6 Discussion

In this section, we share our insightful thoughts and lessons we learned during the competition.

- Picking important hyperparameters to tune is important. We could waste time tuning trivial hyperparameters.

- We should spend enough time tuning important hyperparameters. Specifically, when tuning using Optuna, we should set the trial number to an adequately large number so that it can explore hyperparameter space sufficiently.
- We should try different ensemble methods. We would never know in advance what the ensemble results would be.
- Incorporating various metrics like precision and recall is useful for debugging the performance results.
- Consistency is the key. Because of the daily submission limits, submitting the experimental results daily is important.

References

Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019.

Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016.

Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. *Advances in neural information processing systems*, 30, 2017.

Liudmila Prokhorenkova, Gleb Gusev, Aleksandr Vorobev, Anna Veronika Dorogush, and Andrey Gulin. Catboost: unbiased boosting with categorical features. *Advances in neural information processing systems*, 31, 2018.

A Hyperparameters

Hyperparameters	Search Range	Tuned Models
learning_rate	[0.01, 0.3]	CatBoost, LightGBM, XGBoost
max_leaves	[32, 1024]	LightGBM, XGBoost
max_depth	[5, 10]	CatBoost, LightGBM, XGBoost
min_data_in_leaf	[20, 100]	CatBoost, LightGBM
min_child_weight	[1, 50]	XGBoost
l2_leaf_reg	[1.0, 10.0]	CatBoost