Richter's Predictor: Modeling Earthquake Damage

Tariq Sanda sanda@usc.edu
Nikhil Rajavasireddy rajavasi@usc.edu
Anand Shankar Dyavanapalli dyavanap@usc.edu

Abstract

This paper presents a machine learning-based approach to predict the damage grade of buildings affected by earthquakes. The team utilized Python packages, such as `scikit-learn`, `pandas`, and `numpy`, to develop and train models. An extensive dataset comprising 39 columns, including a unique building ID and 38 features, was preprocessed and analyzed. The team explored six different models, including LightGBM, CatBoost, XGBoost, and a voting ensemble. Model performance was measured using the micro-averaged F1 score, balancing the precision and recall of the classifier. The best model was a voting ensemble consisting of eight independently trained XGBoost sub-models with oversampling of the minority class, achieving a final F1 score of 0.7528. The paper highlights the importance of hyperparameter tuning, ensemble stacking, and efficient training algorithms for improving model performance. The team also acknowledges the black box nature of complex models, emphasizing the need for domain knowledge to identify relevant features.

1 Data Preprocessing

To collaborate more effectively, our team used Google Colaboratory to run Python code within notebooks. We began by reading in the data using `pandas` and adjusted accordingly if the models required the use of `numpy`. For the training data, we merged the features and the associated labels for easier visualization and understanding during the preprocessing phase. We ran exploratory analysis on both the training and test data to ensure data processing was synchronized.

Our first step was to gain a general sense of the data using the `head`, `describe`, and `dtypes` functions in the `pandas` library. Fortunately, we found no missing values in the dataset, and the `dtypes` function helped us identify categorical features. We converted these categorical features to numerical using one-hot encoding and standardized all numerical and boolean features using standard scaling. By rerunning `dtypes` on the data, we verified that all categorical features were changed to type `int64`.

We then split the training data into an 80-20 train-validation subset using the train-test split function. This subset was used to tune hyperparameters and ensure our models did not overfit. To prepare the data for modeling, we performed feature correlation analysis and determined no features were overly represented, and addressed outliers; the column representing the `age` feature had a handful of old buildings that was skewing the data set.

2 Generic Model Training Approach

Our team identified six models of interest and had a general approach to training them. We first identified the parameters of interest and randomly chose initial values. We then trained the models using these initial parameter values to get baseline training and validation scores. For hyper-parameter tuning, we initially used GridSearchCV, but we also used AutoML. We ran grid search on each parameter to estimate the best value that would generate the best classification score. In each iteration,

36th Conference on Neural Information Processing Systems (NeurIPS 2022).

we updated the parameter and used the newly updated set of parameters to estimate the next untrained parameter, until we had trained all parameters. Finally, we retrained the model using the final set of parameters.

During the model training process, we checked two main items. Firstly, we checked whether the new F1 score had increased and compared the train score with the validation score to ensure there was no significant difference, which could point to overfitting. Secondly, we verified whether the loss function had reduced every time we trained the data set.

Once we were confident that the model was appropriately trained, we retrained it on the entire training dataset and then used that model to predict the labels on the `test_values.csv`. We then submitted our predictions to get the submission score. This approach provides a clear description of the steps taken to train the machine learning model and effectively evaluate its performance.

2.1 LightGBM

LightGBM is a gradient boosting framework that uses tree-based learning algorithms. It is designed to be efficient and scalable, capable of handling large datasets and high-dimensional feature spaces. The framework is known for its fast training speeds and lower memory usage compared to other gradient boosting methods. It achieves these improvements through the use of histogram-based algorithms, which approximate continuous feature values into discrete bins, and leaf-wise tree growth, which allows for more complex models and better optimization.

For our LightGBM model, we used the LightGBM library to fit the data. The only preprocessing steps we took were one-hot encoding the categorical features and applying standard scaling to the numerical features. We did not tune any hyperparameters of the model for this particular experiment.

The performance of our LightGBM model was as follows: we achieved a training set micro-averaged F1 score of 0.7915, a validation set score of 0.7410, and a submission score of 0.7394. Although we did not perform hyperparameter tuning in this case, the model still demonstrated competitive performance compared to other models we tested, highlighting the potential of the LightGBM framework.

2.2 CatBoost

CatBoost is an algorithm for gradient boosting on decision trees.

Creating the initial model with just the given default parameters, we got a validation accuracy score of 0.6166 at the 999th boosting step.

Using cross-validation, we were able to get a validation accuracy score of 0.7049 at the 996th boosting step.

We tried using the `use_best_model` parameter to see if there was any difference between a simple model and the "best model", which essentially reduces the ensemble to its best iteration. This, however, resulted in no discernible difference at all, with both validation accuracy scores equal to 0.7147.

We then used CatBoost's `get_feature_importance` function to see which feature(s) had the most significant impact(s) on the model's result. By an overwhelming margin, `geo_level_1_id` had the biggest impact, with a feature importance score of 54.5138. `geo_level_2_id` came in a distant second place, with a feature importance score of 9.2663.

We then moved onto tuning the model parameters, aided by the `hyperopt` package. After 50 iterations, we got an optimal `l2_leaf_reg` parameter value of 2.0, and an optimal `learning_rate` parameter value of 0.4942.

We then moved onto getting all of the cross-validation data using the best parameters. Training on fold [0/3], we got the best accuracy score of 0.7287 on iteration 482/500. Training on fold [1/3], we got the best accuracy score of 0.7310 on iteration 497/500. Training on fold [2/3], we got the best accuracy score of 0.7276 on iteration 489/500. The mean of these three scores gave us our precise validation accuracy score of 0.7289.

2.3 XGBoost

XGBoost (eXtreme Gradient Boosting) is a powerful open-source machine learning library used to build and train gradient boosting decision tree models. The approach is to create new models that predict the residuals or errors of prior models and then combines them to make the final prediction. Since this is a classification problem, we used `multi:softprob` as the objective function and log loss as the evaluation metric.

Initially, we trained the XGBoost model with all default parameters and obtained an F1 score of 0.7184 and a `negative_log_loss` of 0.631765 for the training data, and an F1 score of 0.7119 and a `negative_log_loss` of 0.647392 for the validation data.

We then began tuning the model, starting with `max_depth` and `min_child_weight` parameters. We used GridSearchCV and found the best parameters to be `max_depth=9` and `min_child_weight=1`, with a score of -0.5895. Next, we tuned the `gamma` parameter and found the optimal value to be 0.4, with a score of -0.5990. After adjusting these parameters, the model's F1 score increased to 0.7682 for the training data and the negative log loss decreased to 0.5427. For the validation data, the F1 score increased to 0.7324 and the negative log loss decreased to 0.6043. We saw an overall improvement in the model's performance and continued the process of tuning. Finally, we tuned the `colsample_bytree` and `subsample` parameters and obtained the best score of -0.5983 with `colsample_bytree=0.9` and `subsample=0.7`.

This additional tuning resulted in XGBoost being the model with the best F1 score.

2.4 Voting Ensemble

A soft-voting ensemble was created using Azure AutoML, an automated machine learning framework provided by Microsoft. Azure AutoML simplifies the process of building, training, and deploying machine learning models by automatically trying a wide range of algorithms, including popular approaches such as LightGBM, XGBoost, and CatBoost. Additionally, it performs hyperparameter optimization to further improve model performance. Azure AutoML leverages advanced techniques like Bayesian optimization, random search, and early stopping to effectively explore the hyperparameter space and efficiently find the best-performing models.

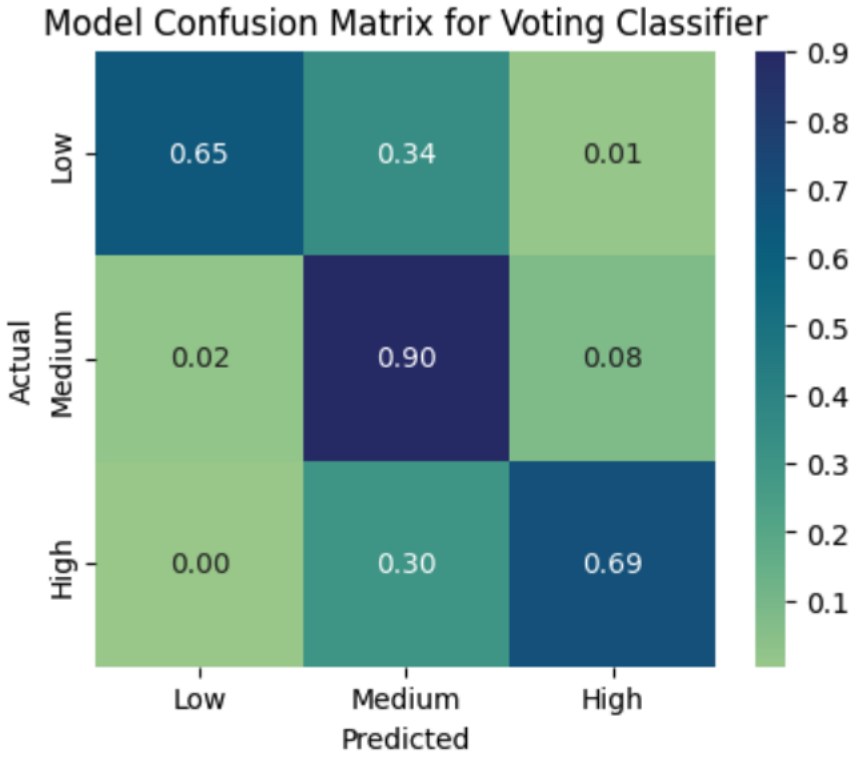
Since we had previously selected the most promising models as LightGBM, XGBoost, and CatBoost, AutoML was run with only these models selected. Each training job consisted of the following steps:

1. Preprocess the training data i.e. one-hot encode and standard normalize.
2. Pick a model (one of LightGBM, XGBoost, and CatBoost).
3. Choose a random number from a given range of values for each hyperparameter.
4. Train the model using 80% of the training data and the other 20% to evaluate model performance.
5. Choose the next set of the hyperparameter values using the Tree-structured Parzen Estimator method, a form of Bayesian optimization.
6. Repeat the previous two steps until a specified time limit is reached or the micro-averaged F1 score does not improve.

Once a series of the best performing models have been trained, Azure AutoML assembles these models into a single ensemble using a `VotingClassifier` and determines the weight of each model's prediction. This classifier operates by taking the majority vote of the individual models for each prediction.

The Azure AutoML framework streamlines the machine learning process by automating many of the tedious and time-consuming tasks involved in model selection and tuning. It employs state-of-the-art techniques and intelligently explores the model space to identify high-performing models and combine them into a robust ensemble. By leveraging Azure AutoML, we were able to efficiently build a powerful voting ensemble model for our data set. See Figure 1 for the results of the confusion matrix for the voting ensemble.

Figure 1: Confusion Matrix



3 Data Insights

The training data set is unbalanced, which causes all models to suffer on the least represented prediction. The normalized confusion matrix for even the best models shows that `damage_grade=1` has a recall_ score close to 0.5, whereas `damage_grade=2` had `recall_scores` in the 0.8s. Our efforts to improve the training data set, such as using the Synthetic Minority Oversampling Technique (SMOTE) to over sample the least represented label, did not noticeably improve submission scores. Additionally, analyzing the feature importance to the labels predicted. We found that only the top 4 features really had a significant impact in accurately classifying the damage grade.

4 Results

The F1 score reached on the first 3 models was significantly lower than the baseline we got from the advanced models. So we didn't proceed further on tuning hyper-parameters and focused on the other models (resulting in the N/A in Table 1). Very early in the approach our team decided to focus our efforts on XGBoost with the understanding it had the best ability to provide the highest score. Using XGBoost as the basis for the voting ensemble, using AutoML developed 8 independently trained XGBoost sub-models, where each sub-model had hyper-parameters tuned separately. Weighting was tuned after each sub-model was completed. The final and best F1 score reached was 0.7528. See Table 1

5 Lessons Learned

From our experience training a machine learning model, we learned that hyperparameter tuning can significantly improve submission scores, sometimes by as much as 0.1 micro-averaged F1 score. We were able to achieve F1 score improvements for almost every base model we used by tuning

Table 1: Model F1 Scores

Model Name	Training Set	Validation Set	Test Set (Submission)
Multinomial Logistic Regression	0.5662	N/A	N/A
SVM	0.5689	N/A	N/A
Nearest Neighbor Classification	0.6353	N/A	N/A
CatBoost	0.7049	0.7289	0.7297
LightGBM	0.7915	0.7410	0.7394
XGBoost	0.7903	0.7507	0.7411
Voting Ensemble	0.7903	0.7507	0.7528

the hyperparameters. Ensemble stacking also helped to increase submission scores, although the improvement was usually modest after hyperparameter tuning.

Training models on a CPU can be time-consuming, and using a GPU can significantly reduce the training time. This is especially important considering that XGBoost is a processing-intensive model, and using a faster training model like LightGBM would have been more efficient. If our team had more time, we would have thoroughly implemented LightGBM to optimize training time and model performance.

We also discovered that hyperparameter tuning is time-consuming due to cross-validation, and running the algorithm on all features, especially those that don't provide much prediction information. This does not just lead to longer processing time but can also lead to overfitting. We would have liked to explore dimension reduction techniques like PCA to help speed up the training algorithms. We also found that different models resulted in a different set of best features for predicting labels, indicating the importance of having domain knowledge to identify relevant features.

In addition, we learned that multi-classification using this dataset did not yield the best accuracy, and that combining output labels to create a binomial classification problem could improve F1 scores. We also discovered that many ML models have built-in features that can handle missing data entries, enable categorical features, and use one-hot encoding.

Finally, we realized that the more complex models we used, the harder it was to understand how they made predictions, highlighting the black box nature of machine learning. While these models could make good predictions, we couldn't pinpoint which specific features and weighting were used to calculate the damage for a specific building.

References

[1] https://medium.com/analytics-vidhya/richters-predictor-modeling-earthquake-damage-b44e3dbdaef

[2] Learing to tune xgboost's hyperparameters https://www.analyticsvidhya.com/blog/2016/03/complete-guide-parameter-tuning-xgboost-with-codes-python/

[3] How to build an CatBoost model: https://catboost.ai/en/docs/concepts/tutorials

[4] How to build an XGBoost model: https://xgboost.readthedocs.io/

[5] How to build a LightGBM model: https://lightgbm.readthedocs.io/

[6] A Python library that helps with hyperparamter tuning: https://optuna.readthedocs.io/

[7] Azure AutoML: https://learn.microsoft.com/en-us/azure/machine-learning/concept-automated-ml?view=azureml-api-2

[8] The blog post that helped us get started: https://drivendata.co/blog/richters-predictor-benchmark/