Analysis of Richter’s Predictor using XGBoost and Ensemble Learning

Authored by: Tharun Ganapathi Raman

1. Introduction

This project aims to compare and evaluate various machine learning models, specifically in the context of earthquake damage prediction, as inspired by the "Richter’s Predictor" research paper.

The focus is on assessing the effectiveness of different algorithms and their ensemble in accurately classifying the extent of damage caused by earthquakes.

Through this comparative analysis, the project seeks to identify the most efficient model or combination of models, providing valuable insights for improved predictive capabilities in earthquake damage assessment and contributing to advancements in disaster management strategies

2. Dataset

The dataset contains information on the structure of buildings and other socio-economic information about each building. Each row represents a building that was affected by the Gorkha earthquake.

There are 39 features with building id being the unique identifier. In this paper, we are predicting damage grade, which is the level of damage to the building that was hit by the earthquake.

There are three grades of damage - low level damage, medium amount of damage and high level damage, each represented by an ordinal variable(1, 2, 3) in the dataset

2. A. Import statements

```
import pandas as pd
import numpy as np

from sklearn.model_selection import train_test_split, RandomizedSearchCV
from sklearn.feature_selection import VarianceThreshold
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import f1_score
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import VotingClassifier
from sklearn.metrics import confusion_matrix, classification_report

from imblearn.over_sampling import SMOTE
from imblearn.under_sampling import RandomUnderSampler
from imblearn.over_sampling import RandomOverSampler

from xgboost import XGBClassifier

import matplotlib.pyplot as plt
import seaborn as sns

from scipy import stats
```

2. B. Loading Dataset

```
train_data = pd.read_csv('train_values.csv')
train_labels = pd.read_csv('train_labels.csv')
```

```
train_data.head()
```

```
train_labels.head()
```

	building_id	damage_grade
0	802906	3
1	28830	2
2	94947	3
3	590882	2
4	201944	3

3. Approach

The approach consists of three parts: Feature Selection, XGBoost Model, Parameter Tuning.

3. A. Feature Selection

Variance threshold has been our main technique for choosing features. removing all the columns having a variance of less than 0.02 by applying a variance threshold of 0.02.

It can be observed that the majority of the columns have secondary uses, and some of the superstructure columns have very little variation.

```
categorical_cols = train_data.select_dtypes(include=['object', 'category']).columns
train_final = pd.get_dummies(train_data, columns=categorical_cols)

selector = VarianceThreshold(threshold=0.02)
X = selector.fit_transform(train_final)

y_label = train_labels['damage_grade'].astype(int)
le = LabelEncoder()
y = le.fit_transform(y_label)
```

3. B. XGBoost

A sophisticated machine learning algorithm called XGBoost is applied to different kinds of predictive modelling.

It is renowned for being effective in tasks involving regression and classification. The way the method operates is by adding trees one after the other, with each new tree fixing mistakes in the ones that came before.

By giving previously incorrectly classified points greater weights during this process, the model's accuracy is improved over time. In essence, it minimises predictive errors by combining the ideas of gradient descent and boosting.

3. C. Parameter Tuning using Randomized Searching

The booster type in XGBoost is determined by general parameters, which also shape the basic behaviour of the model. The performance of the booster is adjusted by booster parameters, which include the number of rounds (nrounds), learning rate (eta), regularisation (gamma), and tree depth (max depth).

The learning task parameters for multi-class classification tasks are configured to use multi-class log-loss to assess the model's accuracy on validation data and multi-softmax for the objective function.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Define the parameter grid
param_grid = {
    'max_depth': [3, 6, 10, 15],
    'n_estimators': [50, 100, 200, 300],
    'learning_rate': [0.01, 0.05, 0.1, 0.2],
    'subsample': [0.5, 0.7, 0.9, 1.0],
    'colsample_bytree': [0.5, 0.7, 0.9, 1.0],
    'reg_alpha': [0, 0.01, 0.1, 1],
    'reg_lambda': [0, 0.01, 0.1, 1]
}

# Initialize the XGBClassifier
xgb_model = XGBClassifier(objective='multi:softprob', num_class=3)

# Set up RandomizedSearchCV
random_search = RandomizedSearchCV(
    xgb_model,
    param_distributions=param_grid,
    n_iter=25,
    scoring='f1_weighted',
    cv=3,
    verbose=1,
    random_state=42
)

# Fit RandomizedSearchCV to the training data
random_search.fit(X_train, y_train)

# Print the best parameters and the best score
print(f"Best parameters found: {random_search.best_params_}")
print(f"Best F1 score: {random_search.best_score_}")

# Use the best estimator to make predictions
y_pred = random_search.best_estimator_.predict(X_test)

# Calculate F1 score for the test set
f1 = f1_score(y_test, y_pred, average='weighted')
print(f"Test F1 Score: {f1}")
```

```
Fitting 3 folds for each of 25 candidates, totalling 75 fits
Best parameters found: {'subsample': 1.0, 'reg_lambda': 0.01, 'reg_alpha': 0.1, 'n_estimators': 100, 'max_depth': 15, 'learn
Best F1 score: 0.7266674217787902
Test F1 Score: 0.7290614336418777
```

Now, training the model with best parameters

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

xgb_model = XGBClassifier(
    subsample=1.0,
    reg_lambda=0.01,
    reg_alpha=0.1,
    max_depth=15,
    learning_rate=0.1,
    n_estimators=100,
    colsample_bytree=0.9
)

xgb_model.fit(X_train, y_train)

y_pred = xgb_model.predict(X_test)
```

```
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy of the XGBoost model: {accuracy}")

score = f1_score(y_test, y_pred, average='weighted')
print(f"F1 Score of the XGBoost model: {score}")

# Generate the confusion matrix
conf_matrix = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:")
print(conf_matrix)

# Generate a classification report
class_report = classification_report(y_test, y_pred)
print("\nClassification Report:")
print(class_report)

# Accuracy
ensemble_accuracy = accuracy_score(y_test, y_pred)
print(f'\nEnsemble accuracy: {ensemble_accuracy}')
```

```
Accuracy of the XGBoost model: 0.735289038967019
F1 Score of the XGBoost model: 0.7290614336418777
Confusion Matrix:
[[ 2541  2559    70]
 [ 1039 24987  3461]
 [  104  6564 10796]]

Classification Report:
              precision    recall  f1-score   support

           0       0.69      0.49      0.57      5170
           1       0.73      0.85      0.79     29487
           2       0.75      0.62      0.68     17464

    accuracy                           0.74     52121
   macro avg       0.73      0.65      0.68     52121
weighted avg       0.74      0.74      0.73     52121


Ensemble accuracy: 0.735289038967019
```

4. Variations in Training using Sampling techniques

We investigated techniques such as oversampling, which replicates minority class instances, undersampling, which decreases the size of the majority class, and SMOTE, which synthesises new minority class samples while reducing instances of the majority class in order to create a balanced dataset and enhance model performance, in order to address class imbalance.

4. A. Under and Over Sampling

As we can see, applying sampling techniques caused the classifier's accuracy rate to drop. Undersampling is ineffective because it throws away potentially valuable information.

The primary drawback of oversampling is that it increases the likelihood of over-fitting by creating exact replicas of the examples that already exist. In fact, it is not uncommon for a learner to create a classification rule that covers a single, replicated example when oversampling is used.The learning time is also negatively impacted by an increase in the quantity of training examples.

```
# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Initialize the undersampler and oversampler
undersampler = RandomUnderSampler(random_state=42)
oversampler = RandomOverSampler(random_state=42)

# Apply undersampling
X_train_under, y_train_under = undersampler.fit_resample(X_train, y_train)

# Apply oversampling
X_train_over, y_train_over = oversampler.fit_resample(X_train, y_train)
```

```
# Train XGBoost model with undersampled data
xgb_under = XGBClassifier(
    subsample=1.0,
    reg_lambda=0.01,
    reg_alpha=0.1,
    max_depth=15,
    learning_rate=0.1,
    n_estimators=100,
    colsample_bytree=0.9
)
xgb_under.fit(X_train_under, y_train_under)
y_pred_under = xgb_under.predict(X_test)

# Train XGBoost model with oversampled data
xgb_over = XGBClassifier(
    subsample=1.0,
    reg_lambda=0.01,
    reg_alpha=0.1,
    max_depth=15,
    learning_rate=0.1,
    n_estimators=100,
    colsample_bytree=0.9
)
xgb_over.fit(X_train_over, y_train_over)
y_pred_over = xgb_over.predict(X_test)

# Evaluate performance of models
accuracy_under = accuracy_score(y_test, y_pred_under)
accuracy_over = accuracy_score(y_test, y_pred_over)
f1_score_under = f1_score(y_test, y_pred_under, average='weighted')
f1_score_over = f1_score(y_test, y_pred_over, average='weighted')

# Print out the results
print(f"Accuracy of the XGBoost model with undersampling: {accuracy_under}")
print(f"F1 Score of the XGBoost model with undersampling: {f1_score_under}")
print(f"Accuracy of the XGBoost model with oversampling: {accuracy_over}")
print(f"F1 Score of the XGBoost model with oversampling: {f1_score_over}")
```

```
Accuracy of the XGBoost model with undersampling: 0.6519636998522669
F1 Score of the XGBoost model with undersampling: 0.656846392373794
Accuracy of the XGBoost model with oversampling: 0.717599432090712
F1 Score of the XGBoost model with oversampling: 0.7188492231699043
```

4. B. SMOTE

When creating synthetic examples, the SMOTE sampling method ignores the possibility that nearby examples may belong to different classes. This leads to an increase in class overlap and extra noise.

```
# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Apply SMOTE to generate synthetic samples and balance the classes
smote = SMOTE()
X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)

# Now, train XGBoost model with the SMOTE'd training data
xgb_model = XGBClassifier(
    subsample=1.0,
    reg_lambda=0.01,
    reg_alpha=0.1,
    max_depth=15,
    learning_rate=0.1,
    n_estimators=100,
    colsample_bytree=0.9
)

xgb_model.fit(X_train_smote, y_train_smote)

# Predict on the original test set
y_pred = xgb_model.predict(X_test)

# Evaluate the performance
accuracy = accuracy_score(y_test, y_pred)
```

```
score = f1_score(y_test, y_pred, average='weighted')
conf_matrix = confusion_matrix(y_test, y_pred)
class_report = classification_report(y_test, y_pred)

# Print out the results
print(f"Accuracy of the XGBoost model with SMOTE: {accuracy}")
print(f"F1 Score of the XGBoost model with SMOTE: {score}")
```

```
    Accuracy of the XGBoost model with SMOTE: 0.7299744824542891
    F1 Score of the XGBoost model with SMOTE: 0.7243925475435237
```

5. Comparing with Basic Models

5. A. Logistic Regression

Predictive modelling usually begins with logistic regression, which assumes a linear correlation between variables. This assumption is a drawback, particularly for multi-class problems where it is less suitable than more complex algorithms due to its tendency to underperform and overfit.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Initialize the Logistic Regression model
logreg = LogisticRegression(max_iter=1000, random_state=42)

# Fit the model on the training data
logreg.fit(X_train, y_train)

# Make predictions on the test set
y_pred_logreg = logreg.predict(X_test)

# Calculate accuracy on the test set
accuracy_logreg = accuracy_score(y_test, y_pred_logreg)
print(f"Test accuracy with Logistic Regression: {accuracy_logreg}")
```

```
    Test accuracy with Logistic Regression: 0.5657412559237159
```

5. B. KNN Modeling

The k-NN algorithm is based on the principle of similarity, assuming that similar data points are in close proximity. This makes k-NN effective for problems where solutions hinge on identifying similar objects, as it groups data points based on their nearness to each other.

(i) Check the best K value

```
# Use a random subset of the training data for quicker cross-validation
subset_size = 500  # Adjust this to your desired subset size
subset_indices = np.random.choice(range(len(X_train_scaled)), subset_size, replace=False)
X_train_subset = X_train_scaled[subset_indices]
y_train_subset = y_train[subset_indices]

# Tune the 'k' parameter
best_score = 0
best_k = 1
for k in range(1, 31, 5):  # Test every 5th value
    knn = KNeighborsClassifier(n_neighbors=k)
    scores = cross_val_score(knn, X_train_subset, y_train_subset, cv=3, scoring='accuracy', n_jobs=-1)
    mean_score = scores.mean()
    if mean_score > best_score:
        best_score = mean_score
        best_k = k

# Train k-NN with the best 'k'
knn_best = KNeighborsClassifier(n_neighbors=best_k)
knn_best.fit(X_train_scaled, y_train)
```

```
# Evaluate on the test set
y_pred_knn = knn_best.predict(X_test_scaled)
accuracy_knn = accuracy_score(y_test, y_pred_knn)

print(f"Best k (using subset): {best_k}")
print(f"Best cross-validated accuracy (using subset): {best_score}")
print(f"Test accuracy: {accuracy_knn}")
```

```
    Best k (using subset): 16
    Best cross-validated accuracy (using subset): 0.5460164009330736
    Test accuracy: 0.6356938661959671
```

(ii) KNN Model with best K value

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# The best 'k' value from your previous analysis
best_k = 16

# Train k-NN with the best 'k'
knn_best = KNeighborsClassifier(n_neighbors=best_k)
knn_best.fit(X_train_scaled, y_train)

# Evaluate on the test set
y_pred_knn = knn_best.predict(X_test_scaled)
accuracy_knn = accuracy_score(y_test, y_pred_knn)
print(f"Test accuracy: {accuracy_knn}")

# Generate the confusion matrix
conf_matrix = confusion_matrix(y_test, y_pred_knn)
print("Confusion Matrix:")
print(conf_matrix)

# Generate a classification report
class_report = classification_report(y_test, y_pred_knn)
print("\nClassification Report:")
print(class_report)

# Accuracy
ensemble_accuracy = accuracy_score(y_test, y_pred_knn)
print(f'\nEnsemble accuracy: {ensemble_accuracy}')
```

```
    Test accuracy: 0.6356938661959671
    Confusion Matrix:
    [[ 2004  2893   273]
     [ 1302 23557  4628]
     [  155  9737  7572]]

    Classification Report:
                  precision    recall  f1-score   support

               0       0.58      0.39      0.46      5170
               1       0.65      0.80      0.72     29487
               2       0.61      0.43      0.51     17464

        accuracy                           0.64     52121
       macro avg       0.61      0.54      0.56     52121
    weighted avg       0.63      0.64      0.62     52121


    Ensemble accuracy: 0.6356938661959671
```

The class imbalance in the dataset affects the k-NN model, especially when more than 50% of the data are in class 2. Class 2 data points may be more prevalent than those of the actual class around a particular point as a result of this imbalance.

5. C. Random Forest Classifier

The Random Forest classifier, an ensemble learning method, constructs multiple decision trees on various subsets of the training set. It generates predictions from each tree and selects the best solution. This approach is less affected by noise compared to a single decision tree, as the collective use of numerous trees helps in delivering more accurate and reliable results.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

rf_best = RandomForestClassifier(max_features = None,
                            max_depth = 45,
                            min_samples_split = 3,
                            min_samples_leaf = 30,
                            random_state=42)

# Fit the model on the training data
rf_best.fit(X_train, y_train)

# Make predictions on the test set
y_pred_rf = rf_best.predict(X_test)

# Calculate accuracy on the test set
accuracy_rf = accuracy_score(y_test, y_pred_rf)
print(f"Test accuracy with Random Forest: {accuracy_rf}")

# Generate the confusion matrix
conf_matrix = confusion_matrix(y_test, y_pred_rf)
print("Confusion Matrix:")
print(conf_matrix)

# Generate a classification report
class_report = classification_report(y_test, y_pred_rf)
print("\nClassification Report:")
print(class_report)

# Accuracy
ensemble_accuracy = accuracy_score(y_test, y_pred_rf)
print(f'\nEnsemble accuracy: {ensemble_accuracy}')
```

```
Test accuracy with Random Forest: 0.7296866905853686
Confusion Matrix:
[[ 2170  2941    59]
 [  839 25368  3280]
 [   71  6899 10494]]

Classification Report:
              precision    recall  f1-score   support

           0       0.70      0.42      0.53      5170
           1       0.72      0.86      0.78     29487
           2       0.76      0.60      0.67     17464

    accuracy                           0.73     52121
   macro avg       0.73      0.63      0.66     52121
weighted avg       0.73      0.73      0.72     52121


Ensemble accuracy: 0.7296866905853686
```

Even with its robustness, the Random Forest classifier's performance in handling data imbalance is only slightly better than the k-NN model, suggesting that it is not fully effective for this particular problem.

6. Ensemble Learning

Ensemble learning is the process of working cooperatively on a dataset with a collection of models, referred to as an ensemble. By utilising the combined advantages of distinct models, this method improves the overall model's stability and predictive accuracy.

Voting determines the final label predictions in ensemble models. There are two types of voting: hard voting, which selects the label that is most common among individual predictions, and soft voting, which selects the label that has the highest average probability by averaging the probability estimates from various classifiers.

Soft voting is used in our study's ensemble model, which is more efficient because it takes into account the estimated probabilities and uncertainties of each classifier in the final prediction.

6. A. KNN + Random Forest

When k-NN and Random Forest are used together in an ensemble, performance is marginally lower than when they are used separately which come from their membership in various classification model families.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Initialize the individual classifiers with their best parameters
knn_best = KNeighborsClassifier(n_neighbors=16)
rf_best = RandomForestClassifier(max_features = None,
                            max_depth = 45,
                            min_samples_split = 3,
                            min_samples_leaf = 30,
                            random_state=42)

# Create the ensemble model using soft voting
ensemble_model = VotingClassifier(
    estimators=[('knn', knn_best), ('rf', rf_best)],
    voting='soft'
)

# Fit the ensemble model on the training data
ensemble_model.fit(X_train_scaled, y_train)

# Make predictions with the ensemble model
y_pred_ensemble = ensemble_model.predict(X_test_scaled)

# Calculate accuracy on the test set
accuracy_ensemble = accuracy_score(y_test, y_pred_ensemble)
print(f"Test accuracy with KNN+Random Forest ensemble: {accuracy_ensemble}")
```

```
    Test accuracy with KNN+Random Forest ensemble: 0.697972026630341
```

6. B. KNN + XGBoost

When compared to XGBoost alone, the k-NN and XGBoost combined ensemble performs slightly worse. This is mainly because the class imbalance in the dataset has a negative effect on k-NN's performance, which in turn affects the ensemble voting process and results in less accurate predictions.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Initialize the classifiers with the previously determined best parameters
knn_best = KNeighborsClassifier(n_neighbors=16)
xgb_best = XGBClassifier(
    subsample=1.0,
    reg_lambda=0.01,
    reg_alpha=0.1,
    max_depth=15,
    learning_rate=0.1,
    n_estimators=100,
    colsample_bytree=0.9
)

# Create the ensemble model using soft voting
ensemble_model = VotingClassifier(
    estimators=[('knn', knn_best), ('xgb', xgb_best)],
    voting='soft'
)

# Fit the ensemble model on the training data
```

```
ensemble_model.fit(X_train_scaled, y_train)

# Make predictions with the ensemble model
y_pred_ensemble = ensemble_model.predict(X_test_scaled)

# Calculate accuracy on the test set
accuracy_ensemble = accuracy_score(y_test, y_pred_ensemble)
print(f"Test accuracy with KNN+XGBoost ensemble: {accuracy_ensemble}")
```

```
    Test accuracy with KNN+XGBoost ensemble: 0.7233360833445253
```

6. C. KNN + Random Forest + XGBoost

The ensemble that uses XGBoost, Random Forest, and k-NN performs marginally better than XGBoost alone. The Random Forest classifier, which may lessen or eliminate the negative effects of k-NN's sensitivity to data imbalance, is probably the reason for this improvement in the ensemble's overall performance.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)


# Initialize the individual classifiers with their best parameters
# Replace 'n_neighbors', 'n_estimators', and 'max_depth' with the best parameters you have identified
knn_best = KNeighborsClassifier(n_neighbors=16)
rf_best = RandomForestClassifier(max_features = None,
                            max_depth = 45,
                            min_samples_split = 3,
                            min_samples_leaf = 30,
                            random_state=42)
xgb_best = XGBClassifier(
    subsample=1.0,
    reg_lambda=0.01,
    reg_alpha=0.1,
    max_depth=15,
    learning_rate=0.1,
    n_estimators=100,
    colsample_bytree=0.9)

# Create the ensemble model using soft voting
ensemble_model = VotingClassifier(
    estimators=[
        ('knn', knn_best),
        ('rf', rf_best),
        ('xgb', xgb_best)
    ],
    voting='soft'
)

# Fit the ensemble model on the training data
ensemble_model.fit(X_train_scaled, y_train)

# Make predictions with the ensemble model
y_pred_ensemble = ensemble_model.predict(X_test_scaled)

# Calculate accuracy on the test set
accuracy_ensemble = accuracy_score(y_test, y_pred_ensemble)
print(f"Test accuracy with KNN+Random Forest+XGBoost ensemble: {accuracy_ensemble}")

# Generate the confusion matrix
conf_matrix = confusion_matrix(y_test, y_pred_ensemble)
print("Confusion Matrix:")
print(conf_matrix)

# Generate a classification report
class_report = classification_report(y_test, y_pred_ensemble)
print("\nClassification Report:")
print(class_report)

# Accuracy
```

```
ensemble_accuracy = accuracy_score(y_test, y_pred_ensemble)
print(f'\nEnsemble accuracy: {ensemble_accuracy}')
```

```
    Test accuracy with KNN+Random Forest+XGBoost ensemble: 0.7329099595172771
    Confusion Matrix:
    [[ 2294  2820    56]
     [  794 25623  3070]
     [   72  7109 10283]]

    Classification Report:
                  precision    recall  f1-score   support

               0       0.73      0.44      0.55      5170
               1       0.72      0.87      0.79     29487
               2       0.77      0.59      0.67     17464

        accuracy                           0.73     52121
       macro avg       0.74      0.63      0.67     52121
    weighted avg       0.74      0.73      0.72     52121


    Ensemble accuracy: 0.7329099595172771
```

7. Result

The results of the project show that the XGBoost model performs better than other models with a high accuracy and F1 score. Even though ensemble models that combine XGBoost, Random Forest, and k-NN exhibit marginal gains, their efficiency does not outperform that of XGBoost used alone. Lower accuracy was shown by the k-NN and logistic regression models, demonstrating XGBoost's superiority in solving this multi-class classification problem.