Kaggle 2EL1730 Machine Learning Project 2026

TEAM NAME: *Team*
Levani METREVELI, Sai ZHANG

1 Introduction

This project focuses on classifying a geographical area of interest from remote-sensing–derived data. Each observation corresponds to an irregular polygon (a mapped ground footprint) described by its geometry and a set of attributes capturing multi-date information about the polygon's status, together with contextual neighborhood and urban features.

The goal is to predict the man-made feature class associated with each polygon (six categories, e.g. Demolition, Road, Residential, Commercial, Industrial, and Mega Projects). The dataset is provided in GeoJSON format (with separate training and test files).

Submissions are evaluated using the Mean F1-Score.

2 Feature Engineering and Preprocessing

The data are provided as GeoJSON polygons. Each sample includes a polygon `geometry`, multi-date metadata (dates, construction status, image statistics), and contextual descriptors (notably `urban_type` and `geography_type`). Since classical ML models require a purely numeric tabular matrix, our pipeline (i) converts geometry into numeric shape descriptors, (ii) enforces a consistent chronological ordering across all date-dependent variables, (iii) encodes multi-label categories, and (iv) handles missing and infinite values.

2.1 Target encoding and basic cleaning

The labels to predict are in the categorical column `change_type`. We mapped that labels to integers : `Demolition`$\rightarrow 0$, `Road`$\rightarrow 1$, `Residential`$\rightarrow 2$, `Commercial`$\rightarrow 3$, `Industrial`$\rightarrow 4$, `Mega Projects`$\rightarrow 5$. Then we drop that column from `x_train` and also we dropped the column `index` wich was unnecessary.

2.2 Geometric features

To capture footprint differences between classes (e.g., elongated roads vs. compact blocks), we computed from the `geometry` feature : polygon area and perimeter (in meters), centroid coordinates, and simple bounding-box descriptors (width, height, area). We also used a standard compactness index

$$\texttt{compactness} = \frac{4\pi A}{P^2},$$

which is close to 1 for circle-like shapes and decreases for elongated polygons. We removed the raw `geometry` object after these feature extractions.

2.3 Temporal features

Raw date columns sometimes appear out of order. We therefore parsed all `dateX` columns and sorted each row chronologically. From the sorted sequences, we derived summaries : (i) total observation span (`date_span_days`) ; (ii) To quantify construction progression, we converted each categorical status into a numeric stage (0–4) and summarized the sequence with : stage at the first available date, stage at the last available date, their difference (progress), maximum and mean stage, number of status changes, number of backward steps (stage decreases), and the time from the first date to the first occurrence of stage 3. (iii) For each per-date image statistic, we computed its value at the first and last available dates, the overall change (last first), and consecutive changes between sorted dates. After generating these summaries, we discarded the raw per-date columns to reduce dimensionality and prevent the model from overfitting to noisy chronology artifacts.

2.4 Multi-label categorical encoding

The columns `urban_type` and `geography_type` contain comma-separated labels. We converted them to a multi-hot representation by creating one binary indicator per label, then dropped the original string columns.

2.5 Missing values and imputation

We replaced $\pm\infty$ with NaN and imputed missing numeric values using the median (and the most frequent category for categorical features if needed). The final design keeps all features numeric (no `object` columns), which simplifies training and ensures compatibility with LightGBM and alternative estimators.

3 Model tuning and comparison

3.1 Cross-validation and grid search

To tune our models in a robust way, we selected hyperparameters using cross-validation (CV) rather than a single train/validation split. For each candidate configuration, we train the model on several stratified splits of the training set and report the mean and standard deviation of the metrics across splits. This reduces the sensitivity to a "lucky" split and makes model selection more reliable.

Because training is computationally expensive, we did not run an exhaustive grid search. Instead, we restricted the search to a small set of deliberately different configurations (5 in total) and used a 3-split stratified CV. This corresponds to 15 training runs for this model family (plus the final refit on the full training set once the configuration is selected).

Example : LightGBM. For gradient-boosted decision trees, we designed five configurations with contrasted bias–variance trade-offs : (i) a baseline setup ; (ii) a more conservative variant with stronger regularization and fewer effective degrees of freedom (smaller leaves / larger minimum leaf size / feature subsampling) ; (iii) a higher-capacity variant (more leaves, smaller learning rate) that can improve performance but may overfit ; (iv) an explicitly regularized variant using both L1 and L2 penalties ; (v) a high-capacity "many trees" variant (large number of estimators and small `min_child_samples`) but carries the highest overfitting risk.

#	`Hyperparameters`	Train Macro-F1	Train Weighted-F1	Val Macro-F1	Val Weighted-F1
1	`dict()`	0.9372 ± 0.0009	0.8856 ± 0.0016	0.5451 ± 0.0061	0.7722 ± 0.0008
2	`dict(num_leaves=63, min_child_samples=25, colsample_bytree=0.8, reg_lambda=1.0)`	0.9367 ± 0.0009	0.8869 ± 0.0017	0.5509 ± 0.0061	0.7738 ± 0.0006
3	`dict(learning_rate=0.03, num_leaves=127, min_child_samples=10)`	0.9388 ± 0.0006	0.8873 ± 0.0006	0.5476 ± 0.0055	0.7727 ± 0.0002
4	`dict(reg_alpha=0.5, reg_lambda=2.0, min_child_samples=10, num_leaves=63)`	0.9405 ± 0.0010	0.8935 ± 0.0017	0.5523 ± 0.0055	0.7744 ± 0.0006
5	`dict(n_estimators=2500, num_leaves=127, min_child_samples=7)`	0.9999 ± 0.0000	0.9998 ± 0.0000	0.5461 ± 0.0044	0.7792 ± 0.0016

TABLE 1 – 3-split CV results (mean ± std).

Based on the validation weighted F1-score, configuration #5 achieves the best mean performance (0.7792 ± 0.0016). Its standard deviation remains low overall (although it is about twice larger than the other configurations), which is still reassuring regarding stability across the three folds. However, it reaches an almost perfect training weighted F1 (near 1.00), suggesting a higher risk of overfitting. Configuration #4 provides a more conservative trade-off : it remains competitive on validation (0.7744 ± 0.0006)

while relying on stronger explicit regularization (L1+L2). Since Kaggle allows up to two submissions, we submit both #5 (score-oriented) and #4 (robust, regularized) to hedge against potential generalization issues on the hidden test set.

3.2 Model benchmarking

Beyond tuning a single estimator, we tested several model families with different inductive biases, in order to understand which approaches best exploit our engineered tabular features. The benchmark includes boosting methods (LightGBM, XGBoost), a bagging baseline (Random Forest), as well as non-tree alternatives (linear SVM, k-NN, and a feed-forward neural network). Depending on the model, we applied either light tuning or a restricted search when training cost was significant.

Model	Family	Val Macro-F1	Val Weighted-F1
LightGBM	Boosting	0.5461	0.7792
XGBoost	Boosting	0.5569	0.7665
Random Forest	Bagging	0.5293	0.7401
k-NN	Distance-based	0.4685	0.7118
Linear SVM	Linear classifier	0.4604	0.6658
Neural Net	MLP	0.4692	0.6859

Table 2 – Summary of model benchmarks validation scores.

Remarks. Overall, tree-based boosting methods perform best, which is consistent with their ability to capture non-linear interactions and heterogeneous feature effects in tabular data. Random Forest is competitive but slightly behind. Distance-based k-NN and linear SVM provide reasonable baselines. Finally, the neural network model remains below the boosted trees despite early stopping, suggesting that the current tabular representation is better exploited by tree ensembles than by a plain feed-forward network.

4 Conclusion

We built a complete pipeline to turn GeoJSON polygon data into a clean tabular dataset suitable for classical machine learning. Across several model families, boosted tree methods achieved the strongest results on our engineered features, with LightGBM providing the best weighted F1 on validation and XGBoost remaining competitive. Overall, the combination of careful feature engineering and boosted trees offers the best performance–robustness trade-off for this task, and it provides a solid baseline for future improvements.