Kaggle Project

Team : The Feature Engineers

Lina HEMMAZ
Yacine GHANASSI
Alexian HÉLAINE

2A-CentraleSupélec

2025

1 Feature engineering

1.1 Geometric features

Geometric features capture the shape and spatial characteristics of each zone. For example, large areas with low compactness might correspond to industrial zones or mega projects, while small, compact shapes could indicate residential or commercial areas. The centroid coordinates also provide spatial localization that may correlate with certain types of changes.
Therefore, we created the following features :

— Area, computed as

$$area = geom.area$$

— Perimeter, computed as the length of the polygon's boundary, i.e.,

$$perimeter = geom.length$$

— Compactness, defined as

$$compactness = \frac{4\pi * area}{perimeter^2}$$

which measures how circular the shape is. A value closer to 1 indicates a shape near a circle, whereas lower values suggest more elongated or irregular forms.

— Centroid Coordinates : Extracted as centroid-x and centroid-y from the polygon's centroid.

We implemented thoses feature with the motivation of using a logistic regression classifier later.

```python
def extract_geometric_features(df):
    features = []
    for geom in df['geometry']:
        if geom.is_empty:
            features.append([0, 0, 0, 0, 0])
        else:
            area = geom.area
            perimeter = geom.length
            compactness = 4 * np.pi * area / (perimeter ** 2) if perimeter > 0 else 0
            centroid_x, centroid_y = geom.centroid.x, geom.centroid.y
            features.append([area, perimeter, compactness, centroid_x, centroid_y])
    return pd.DataFrame(features, columns=["area", "perimeter", "compactness", "centroid_x", "centroid_y"
```

FIGURE 1 – Geometric features

1.2 Images features

We also used the mean and standard deviation of the color channels (red, green, blue) over 5 dates. This results in features such as :
img-red-mean-date1, img-blue-std-date3,etc.

These features capture the visual appearance of an area over time. Changes in the average color or its variation may indicate modifications in land use, construction, or demolition. For instance, a significant change in the red channel could reflect construction activity or a change in vegetation.

1.3 Temporal Features

Temporal features help capture the dynamics of changes over time. One-hot encoding the change statuses makes categorical temporal information usable by machine learning algorithms, while the time

```python
def extract_image_features(df):
    feature_columns = []
    for i in range(1, 6):
        for color in ["red", "green", "blue"]:
            feature_columns.extend([f"img_{color}_mean_date{i}", f"img_{color}_std_date{i}"])
    return df[feature_columns] if set(feature_columns).issubset(df.columns) else pd.DataFrame()
```

FIGURE 2 – Images features

differences provide insights into the pace of change (e.g., rapid changes could indicate demolition or fast-paced urban development). Therefore we prefered using the time difference than the dates.

For consecutive date columns (e.g.,date1 and date2), we calculate the time difference in days. We also used, as advised, OneHotEncoder. We encode categorical change statuses (e.g., change-status-date1 to change-status-date5) into binary indicators.

```python
def process_temporal_features(df):
    temporal_features = pd.DataFrame()
    status_columns = [f"change_status_date{i}" for i in range(5)]
    if status_columns:
        one_hot_encoder = OneHotEncoder(handle_unknown='ignore')
        one_hot_encoded = one_hot_encoder.fit_transform(df[status_columns]).toarray()
        status_df = pd.DataFrame(one_hot_encoded, columns=one_hot_encoder.get_feature_names_out(status_columns))
        temporal_features = pd.concat([temporal_features, status_df], axis=1)
    date_columns = [f"date{i}" for i in range(5)]
    if all(col in df.columns for col in date_columns):
        for i in range(len(date_columns) - 1):
            col_name = f"time_diff_{date_columns[i]}_{date_columns[i + 1]}"
            temporal_features[col_name] = (pd.to_datetime(df[date_columns[i + 1]], dayfirst=True) - pd.to_datetime(df[date_columns[i]], dayfir
    return temporal_features
```

FIGURE 3 – Temporal features

1.4 Neighbour features

Neighborhood features introduce contextual information regarding the environment surrounding each area. The urban and geography types indicate the broader setting, which can be crucial in determining the nature of changes (e.g., urban vs. rural developments). That is why, after using OneHot Encoder, we used Urban type and Geography type (handling multiple values separated by commas). We had decision trees classifers in mind while implementing the neighborhood features.

```python
def process_neighbourhood_features(df):
    neighbourhood_features = pd.DataFrame()
    if 'urban_type' in df.columns:
        urban_types_df = pd.get_dummies(df['urban_type'], prefix='urban_type')
        neighbourhood_features = pd.concat([neighbourhood_features, urban_types_df], axis=1)
    if 'geography_type' in df.columns:
        geography_types_df = df['geography_type'].str.get_dummies(sep=',')
        neighbourhood_features = pd.concat([neighbourhood_features, geography_types_df], axis=1)
    return neighbourhood_features
```

FIGURE 4 – Neighborhood features

1.5 Reduction dimentionality

We used PCA in order to do a reduction dimensionality that we used for every model classifiers.

2 Model tuning and comparison

In our classification task, we experimented with various machine learning models to identify the most effective approach. Given the nature of our dataset, which consists of both numerical features (e.g., area, perimeter) and categorical variables, we tested Logistic Regression, Random Forest, and XGBoost, among other models. Below, we discuss our reasoning, parameter tuning strategies, and the performance results of each model.

2.1 Logistic regression

Initially, we applied Logistic Regression to our dataset, leveraging the numerical features such as area and perimeter. However, we quickly observed that Logistic Regression was not well-suited for our problem. The model assumes a linear decision boundary, which may not effectively capture the complex relationships within the data, especially given the mix of numerical and categorical variables. Even after normalizing the numerical features and encoding categorical variables using one-hot encoding, the model's performance remained suboptimal. Therefore, while Logistic Regression provided a baseline, we decided to explore more flexible models. We tried to implement cross validation logistic regression with polynomial features, to have a better score. However, the code would take too much time and space, that it didn't work and was not interesting.

2.2 Random Forest

Recognizing the limitations of Logistic Regression, we implemented a Random Forest classifier. Random Forest is particularly well-suited for datasets with mixed feature types because it can handle both numerical and categorical variables without requiring extensive preprocessing. Additionally, it is robust to outliers and resistant to overfitting due to its ensemble nature. For Random Forest we used all the features we created, in order to exploit the numerical and categorical mix.

Parameter tuning :

— Number of Trees (n-estimators) : We tested values ranging from 50 to 500, ultimately selecting 100 as the best balance between accuracy and computational efficiency.

— Max Depth (max-depth) : The model was run with the default max depth (None), allowing trees to grow until all leaves are pure or contain less than the minimum sample split.

— Minimum Samples per Split (min-samples-split) : We kept the default of 2, meaning a node must have at least two samples to be split.

— Feature Selection (max-features) : The default value ('auto', equivalent to 'sqrt' for classification) was used to optimize feature selection per split.

Surprinsingly we tried to better the score with more estimators (n=500), limiting the depth to reduce overfitting. Also, whereas in the first RF we directly use the colomn train-features, here we extracted geometric, images and neighbohood features. Nonetheless we had a smaller score (0.85866). Wherease with a simpler tuning of the random forest we had the best score. The key differences between the two codes mainly concern feature extraction and processing. The first code introduces an additional feature, mean-intensity, which averages the mean and standard deviation values of the image-related features. This could provide a more compact representation of image data, potentially improving model performance. Additionally, the first code ensures that categorical features like change-status-date are properly one-hot encoded while verifying their existence in the dataset, making it more robust. Another critical improvement in the first code is the alignment of train and test features, ensuring consistency and preventing errors when making predictions. The second code lacks these refinements and does not include feature selection, model training, or evaluation.

We also tried a version where we introduced class weights and we changed the use of PCA : the updated version introduces a more strategic approach by visualizing the explained variance ratio to determine the optimal number of components to retain, improving feature representation. The results were similar to the first RF model.

```python
# --- 9. ENTRAÎNEMENT DU MODÈLE RANDOM FOREST ---
rf = RandomForestClassifier(n_estimators=500, max_depth=30, random_state=42, n_jobs=-1)
rf.fit(X_train, y_train)
```

FIGURE 5 – Random Forest Optimized

Overfitting Prevention : To mitigate overfitting, we applied cross-validation and used out-of-bag (OOB) error estimation. We also pruned deep trees and limited the number of features considered at each split. The model performed significantly better than Logistic Regression, making it a strong candidate for further optimization.

We had good results for random forest :

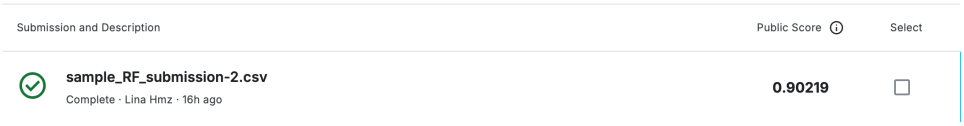

FIGURE 6 – Score Random Forest

2.3 XGBoost

We tried XGBoost thinking that it would perform better than the Random Forest. In the first model, feature preprocessing is minimal. We also used all the features in the first place. There is no feature scaling applied. This means that features with different magnitudes may have an uneven influence on the model, potentially leading to suboptimal convergence.

Unlike the second model, class balancing is not considered, which could make it harder for the model to correctly classify underrepresented categories. Additionally, the feature set is used as-is, without transformations like normalization or weighting. While this approach is faster, it may lead to lower performance compared to the second model, which incorporates more refined preprocessing techniques.

In the second model, feature preprocessing is improved by applying StandardScaler to normalize the features. We thought that this would ensure that all features have a similar scale, which helps gradient-based models like XGBoost converge more efficiently. Furthermore, class balancing is introduced by computing weights for each class (scale-pos-weight). This helps the model learn better in cases where some categories are underrepresented. Feature selection itself remains unchanged, but these preprocessing steps were supposed to improve model stability, training efficiency, and overall performance. It did not really.

However, generally the score of XGBoost was inferior than the score for Random Forest. Moreover, the "optimized" version of XGBoost gave us a smaller score on the kaggle submission.

2.4 Neural Netwoks

We tried to implement MLP and CNN classifier, focusing of the images features. However the scores were very low (0.00608) and we couldn't figure out which parametrization would make it better. We decided to drop it then and focus on Random Forest classifier.