Kaggle Challenge: Predicting Forest Cover Type

DELSART Matthieu, LEBRUN François, VALENCONY Tim - 01/04/2024
Data Science for Business, École Polytechnique - HEC

Table of contents

1. Abstract

The objective of this project was to predict the forest cover type among 7 possible classes, using various information related to some specific areas. Our final model to address this multi-class classification problem is an ExtraTree Classifier, with several techniques implemented to address the class imbalance, the most important one being an SVMSmote oversampler, and with features based especially on non linear combinations of the original features as well as unsupervised learning methods. This model enabled us to get a final score of 0.872, up from 0.794 when we had taken this challenge in early September.

2. Methodology

Our organization to solve this multi-label classification problem has been to break down the tasks between the three of us. Although we both contributed to every stage, Matthieu's main focus was on the exploratory data analysis ("EDA"), the oversampling and using unsupervised or semi-supervised method, François worked on the two models implementation, some features engineering and the post-processing part, while Tim focused on looking into dimension reduction techniques, model selection and hyperparameter tuning.

Our overall working process has been the following:

1. Perform the exploratory data analysis and identify potential challenges regarding the train and test set we have. One specific challenge appeared to be the strong class imbalance between the train and the test sets.
2. Build and try different methods to tackle this specific challenge, including a specific cross-validation function and several oversampling techniques, among which we selected SVMSmote
3. With this oversampler and a classical Random Forest as baseline, we then moved to the feature engineering and selection part, using especially nonlinear combinations and unsupervised learning methods to create new features
4. With these features, select the best model, which turned out to be an ExtraTrees Classifier
5. Tune the model and its hyper-parameters with these features, especially using OPTUNA
6. Implement a last step of post-processing to optimize the prediction

3. Exploratory Data Analysis

3.1. Features Exploration

The dataset was composed of 14 variables, including 12 numerical variables and 2 categorical variables, which were both one-hot encoded. The first two main steps we took, in order to get a better understanding of the data, were to first look into each feature separately, especially its distribution, and to then look at the correlations between the variables. We quickly noted that the variables had very different scales and distributions, which could pose a problem for later stages. Some of the key points we noted are:

- Regarding the **"*Id*"**, we noted that this variable has a uniform distribution, with one different Id per row of the dataset. While this could potentially indicate that this feature is just a row number and could be deleted, we remembered from our exploration in September that it was however significant, probably to the fact that the parcels had not been given numbers randomly (and later modeling showed it was one of the most useful variables). In particular, Ids were especially present in the beginning (before 50k) and in the middle (between 200k and 400k). Another interesting thing we learnt thanks to the Ids was that the training set was included into the full testing test, which was something we would have to take into account at the moment of the prediction. Finally, we noted that the "Id" distribution in the training set was far from being uniform, showing that the training set had not been sampled simply randomly from the full one.

- From then on, we kept noting different distributions on the **other numerical variables** between the training and test sets, although to some different extent. We will not extend in detail on the definition of these different variables, which can be found in the problem description but these variables include information regarding the parcel's elevation, slope, orientation, exposition to the sun, as well as distance to water, roads and fire points. All these variables seemed intuitively relevant to us to predict the type of plant that could grow on the parcels.

- **"*Wilderness Area*"** was the first categorical variable, with 4 categories corresponding to 4 different wilderness areas. Here again, we observed a very different distribution in the train and test sets, as can be seen in the table below. As can be seen, the wilderness area n°4 is strongly overrepresented in the training set vs the test test, while the wilderness area n°1 is strongly underrepresented. At the time, this led us to formulate the hypothesis that the training set had been built by sampling especially some wilderness areas compared to some others, which could be an explanation for the different distribution of the variables.

Proportion	Wilderness Area 1	Wilderness Area 2	Wilderness Area 3	Wilderness Area 4
Training set	24%	4%	42%	31%
Test set	45%	5%	44%	6%

Fig. 1. Proportions of each wilderness area in the train and test sets

However, we noted that, even among different wilderness areas, the variables did not have the same distribution, meaning that this hypothesis was wrong. This can be seen for instance in the two graphs below, which represent the distribution of the *"Elevation"* depending on the wilderness areas and shows different distributions, especially for the n°1 and n°3 wilderness areas.

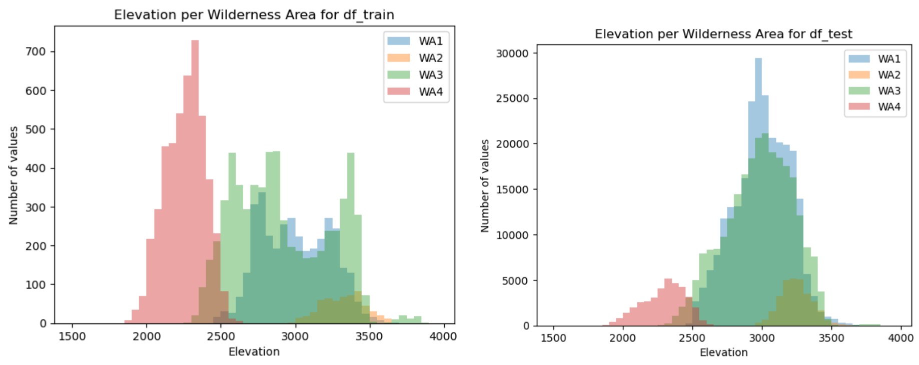


*Fig. 2. Distribution of the "*Elevation*" variable depending on the wilderness areas, in the train and test sets*

- Finally the **"*Soil_Type*"** was a categorical variable made of 40 different categories, with some of them being strongly overrepresented and other underrepresented, or even absent in the case of the training set. Once again, the distribution was very different between the two test sets. We knew from the problem statement that these categories were not arbitrary or the result of an ordinal encoding but corresponded to precise sub-components, which led us to try to encode these sub-components at later stages of the project.

Regarding the correlations between the coefficients, we noted that some features were strongly correlated, some of which could be expected, for example between the different "*Hillshade*" variables. This was something we would have to be careful about during the feature selection and training processes.

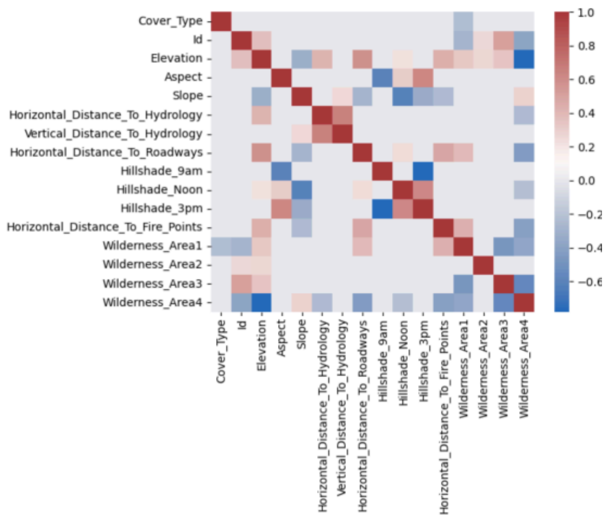


Fig. 3. Covariance matrix of the different features
*For readability, only cases with a value lower than -0.2 and higher than 0.2 are represented. For the same reason, "*Soil_Type*" is not shown here.*

3.2. Baseline model and Target variable

With these findings in mind, we decided to build a baseline model and look into the distribution of the predictions, hoping that it would enable us to better understand how the training set had been built. As a choice of baseline, we used a basic Random Forest Classifier [21], with all the default parameters except for `n_estimators=150` (instead of 100 by default) and setting the random seed at 42. This was also the model which had enabled us to get the best score in early September, allowing us to use it as a benchmark against our posterior efforts.

This baseline prediction allowed us to get an accuracy of 0.795 and showed us very interesting results on the target variables of the test set. As can be seen in the table below, the classes have very different distributions in our predictions, whereas they are perfectly uniformly distributed in the training set, with 2160 instances each. Although this is only a prediction, the fact that this prediction reaches 0.79 accuracy shows that there *is* a strong imbalance among classes, even in the case where the 21% left would be completely differently distributed.

Cover Type	1	2	3	4	5	6	7
Training set proportions	14,3%	14,3%	14,3%	14,3%	14,3%	14,3%	14,3%
Baseline proportions	37,2%	40,6%	6,6%	0,1%	4,9%	4,5%	5,4%

Fig. 4. Proportion of each target instance in the train set and test set (predicted according to baseline)

This led us to formulate the assumption that the training set had been built by sampling without replacement in order to get a uniformly distributed subset across classes. As will be seen below, this assumption was key in our work afterwards. Its credibility of this assumption was later reinforced by the fact that we observed that, among some classes (predicted or not), and especially for the classes with the highest accuracies, the features had very similar distributions (unlike in the case of the wilderness areas, as discussed above).

Finally, we looked at the features importances as analyzed by the model, noting that the most important ones at this point were "*Elevation*", "*Id*", "*Horizontal_Distance_to_Roadways*", and "*Horizontal_Distance_To_Fire_Points*".

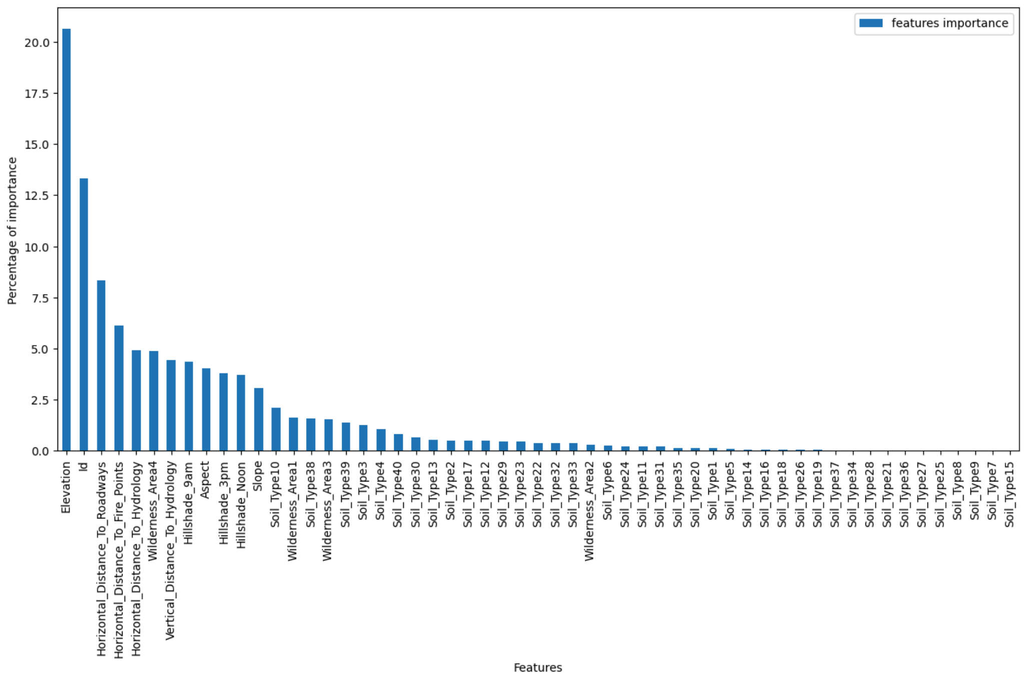
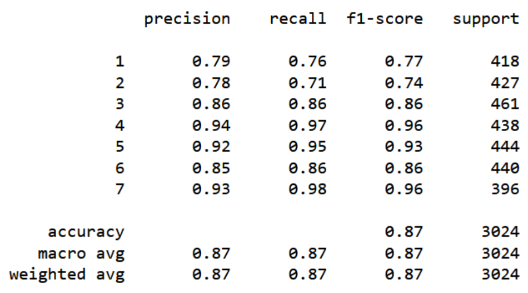


Fig. 5. Most important features according to our baseline (percentage of model variance explained)

4. Tackling class imbalance

One of the main challenges we met during this project was to tackle the strong class imbalance between the train and the test sets. Indeed, as we saw previously, the train dataset is perfectly balanced between classes (exactly 1/7th each). The test dataset, however, is strongly imbalanced with classes 1 and 2 accounting together for more than 80% of the total (although this figure is only a proxy based on our baseline prediction, the fact that it reaches 0.79 accuracy shows that there *is* a strong imbalance).

If not addressed specifically, this imbalance has strong detrimental effects on our prediction quality. The two tables below, generated with our Random Forest baseline and a 80-20 standard split, show that (*Fig. 6 Classification report*) only the classes 1 and 2 have f-1 scores below 0.86, and that (*Fig. 7. Confusion matrix*) the main prediction issue the model has is between distinguishing classes 1 and 2. This strong imbalance is enough to make the overall accuracy drop to 0.87 (in the case of a balanced dataset, see *Fig. 6.*) and to 0.79 for the baseline on the Kaggle dashboard.



```
              precision    recall  f1-score   support

           1       0.79      0.76      0.77       418
           2       0.78      0.71      0.74       427
           3       0.86      0.86      0.86       461
           4       0.94      0.97      0.96       438
           5       0.92      0.95      0.93       444
           6       0.85      0.86      0.86       440
           7       0.93      0.98      0.96       396

    accuracy                           0.87      3024
   macro avg       0.87      0.87      0.87      3024
weighted avg       0.87      0.87      0.87      3024
```

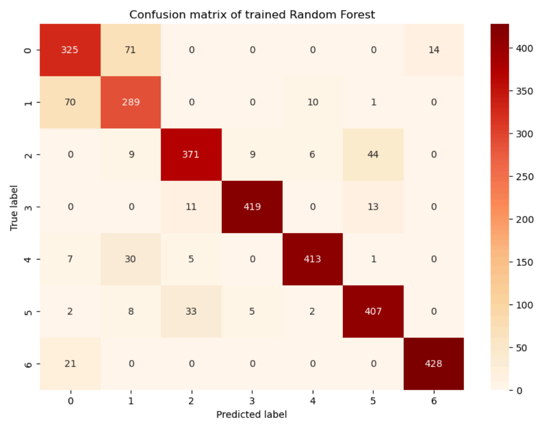


Fig. 6. Classification report on the baseline, using scikit-learn method

Fig 7. Confusion matrix on the baseline predictions

One of our main objectives has thus been to increase the accuracy of the classes 1 and 2, even at the expense of the other classes, as this would overall increase the accuracy. With the assumption that the training set was generated by sampling the test set so as to get a balanced dataset (see above), this situation of imbalance between the classes in the train and the test sets is known in the literature as a **"co-variate shift"** [1]: the distribution of co-variates differs from the source to target domain, but the data are otherwise generated the same. We used different methods to tackle this co-variate shift, finally opting for oversampling techniques and a bit of weighted loss function.

4.1. Importance Weighted Cross Validation

The first step we took to tackle this covariate shift was to implement a **specific cross-validation tool**, taking the imbalance into account. Indeed, the classic cross-validation tools (eg. from scikit-learn) does not provide an unbiased estimate of final accuracy: as we saw above, due to the very different proportions of classes 1 and 2 in the train and test sets and to the fact that the accuracy on these 2 classes is lower, we get 0.87 as accuracy estimate from the cross validation but only 0.79 when submitting on Kaggle. Given that submitting every time on Kaggle was not an option, we needed an estimate of our final score that we could use for further steps.

We decided to follow Sugiyama et al. [1] by building an **Importance Weighted Cross Validation** (IWCV), which can be proved to be an unbiased estimate of the final accuracy even under the covariate shift. The IWCV k-fold estimate is given by the following:

$$\hat{R}^{(n)}_{kIWCV} = \frac{1}{k}\sum_{j=1}^{k}\frac{1}{|T_j|}\sum_{(x_i,y_i)\in T_j}\frac{p_{test}(x_i)}{p_{train}(x_i)}\ell(x_i, y_i, \hat{y}_i)$$

where k is the number of folds, T_j are the validation subsets, ℓ is the loss function and p_{test} and p_{train} are the test and train distributions respectively. In this equation, $p_{test}(x_i)$ and $p_{train}(x_i)$ are unknown. However, with the covariate shift assumption and our assumption that the training set was created by sampling uniformly the test set so as to get a balanced set, we get that:

$$\frac{p_{test}(x_i)}{p_{train}(x_i)} = \frac{p_{test}(y_i)}{p_{train}(y_i)}$$

Finally, given that distribution of the labels over the the test set is unknown, we decided to use it as a proxy for the distribution in our baseline prediction (which is not fully accurate but provides an easy proxy, and later results showed a good coherence between the IWCV estimates with this method and the final Kaggle results). For instance, given that class 2 accounts for 43% of the target dataset in our baseline prediction and 14,3% of the train dataset, its associated coefficient becomes 43% / 14,3% = 3.02. The detailed coefficients per label can be found in the appendix.

Our overall IWCV method was thus the following:

- Divide the training set in k folds, leave every time one portion aside, fit the model on the first portion as in a classical cross-validation;
- On the test portion, compute the accuracy *class by class* and calculate a weighted average of these accuracies using the coefficients obtained above;
- Average these estimates on the k folds.

This method has enabled us to get a **relatively good-quality estimate of the final accuracy,** which was very helpful throughout the later stages of the projects. The coefficients were also reused at later stages (see “4.3 Weighted Loss Function”).

4.2. Two layers of models

The first approach we tried to tackle the imbalance was to implement two successive layers of models:

- The first one to distinguish classes 1 and 2 from the others
- The second one to be able to distinguish, among the 2 obtained ensembles, the classes themselves (class 1 from class 2 in the first ensemble, classes 3, 4, 5, 6 and 7 in the second one)

The approach was motivated by (i) the overwhelming weight of classes 1 and 2 compared to the others (it may be easier to first do a binary classification between ~80% vs ~20% of instances, and then distinguish among these 2) and (ii) the fact that the model confusion is the highest between classes 1 and 2, meaning that a specific model to distinguish these two could be useful [2].

To implement this approach, it was therefore first necessary to classify the Cover Types according to whether or not they were less class 1 and 2 or not, which is equivalent to if their class was less than 2. We thus got 2 separate datasets, one with Cover Types 1 - 2 and another one with Cover Types 3-4-5-6-7. Our task now was to focus on the correct classification of the first dataset, so as to have a model that differentiates between 1 and 2. To do this, we trained various models (xgBoost, LightGbm, RFclassifier ...). Thanks to cross-validation, the final choice fell on the faster and slightly more powerful LightGBM [3]. Then, regarding the problem of multiclass classification between 3-4-5-6-7 we proceeded in the same way. In the end, we thus had 3 models, with their parameters each, for a total accuracy of 0.83.

1-2 vs 3-4-5-6-7	`LGBMClassifier(objective='binary', is_unbalance=True, scale_pos_weight=3.5)`
1 vs 2	`LGBMClassifier(objective='binary', is_unbalance=True, scale_pos_weight=1.1)`
3 vs 4 vs 5 vs 6 vs 7	`LGBMClassifier(objective='multiclass', num_class=5)`

The `scale_pos_weight` parameter was defined as the estimated target ratio between the two classes in the two cases of binary classification.

4.3. Weighted Loss Function

Our second approach was to use a weighted loss function when training the model. In order to improve the accuracy on two specific classes (classes 1 and 2), one approach can indeed be to increase the loss of the instances associated with these classes, thus "forcing" the model to focus more on these two classes when training. This can be implemented easily with the `class_weight` parameter in scikit-learn models or the `class_weights` one in the LightGBM library, which used a lot with this method especially. Here, we used the same coefficients as the ones in the "Importance Weighted Cross Validation" part.

This method gave much better results than the baseline and slightly better ones than the "2-layers of models" one, especially with combining it with LightGBM Classifier (around 0.835).

Later on, when we used oversampling rather than this method in order to tackle the class imbalance (see below), we however added a final layer of class weights in order to compensate for the sometimes extreme effect of oversampling. Indeed, as the oversampling makes the classes 1 and 2 overrepresented in the train set, when hesitating on classes with lower amounts of data the algorithm will most likely predict prominent classes in the train set. Therefore, to have better accuracy on the underrepresented classes (classes 3 through 7), we gave them more weight. This proved to be efficient, and helped us reach the final stretch of precision needed for our final prediction.

4.4. Oversampling

In the final approach, we tackled the co-variate shift using oversampling. This method involves increasing the number of instances in some classes by either duplicating existing instances or generating new synthetic instances (see for example the imbalance-learn library [6], on which we relied in this part). Although oversampling is most often used in order to oversample a minority class in the case of an unbalanced dataset, we decided to use it in a balanced dataset to **oversample the classes that *should be* the majority classes (classes 1 and 2) and to use the obtained synthetic dataset as our training set.**

An advantage of oversampling is to easily enable a relatively similar distribution of the labels in the train and test sets if we have an estimate of the labels distribution in the test set. Given the ~40% of classes 1 and 2 in the baseline prediction, we initially chose to overample class 1 and class 2 to 30,000 samples each (instead of 2,160) allowing to approximate these proportions in the synthetic set (these proportions were later tuned as hyperparameters at the moment of the tuning stage). Another advantage of oversampling techniques is that some of them help produce a "borderline" between classes, making them easier to distinguish while improving classification results (see ADASYN [4] and SVMSmote [5]).

We tried several oversampling techniques, relying on the imbalanced-learn [6] library which is built on-top of scikit-learn library and which was easy to use. The main ones were:

- **RandomOverSampler**: Performs random oversampling by oversampling the target classes by picking samples at random with replacement. This gave poor results.

- **SMOTE** (Synthetic Minority Oversampling Technique) [7][8]: Generates synthetic data by interpolating the k-nearest neighbors of the target classes. This gave better results than the Random Over Sampler.

- **ADASYN** (Adaptive Synthetic Sampling Approach for Imbalanced Learning) [9]: Generates synthetic samples based on the density distribution of the target class samples. It first calculates the density distribution of the minority class samples and then generates synthetic samples in proportion to the density distribution, especially focusing in the sparse regions of the feature space, which can enable to improve the classification performance. The results were much better (around 0.84 when used with a Random Forest), which is why we used it as a new baseline during most of the project until we discovered SVMSMote.

- **SVMSmote** (Support Vector Machines - Synthetic Minority Oversampling Technique) [10]: Uses SMOTE in combination with SVM to select the most informative samples to generate synthetic data. More specifically, the SVM is trained on the target class samples and is used to identify the borderline samples that are most difficult to classify. These borderline samples are then used as the basis for generating new synthetic samples using the SMOTE algorithm. By focusing on the borderline samples, SVMSmote aims to generate synthetic samples that are more informative and useful for improving the classification performance. This technique, which we discovered at later stages of the project, proved **much more successful than everything we had tried before** to tackle the covariate shift issue, with more than 0.85 accuracy when combined with a simple Random Forest Classifier.

One main drawback of oversampling is that it relies on synthetic data, thus threatening to generate overfitting if the synthetic data are too close to the original ones, or on the contrary to decrease the predictions accuracy by introducing too much noise in the training data. Furthermore, by increasing (almost 6x before tuning) the size of the training dataset, it makes training much longer, which poses a problem when combined with cross-validation. However, given the strong increase in accuracy we saw with these techniques, we decided to **use mainly oversampling to tackle the imbalanced data issue, and more specifically the SVMSmote.** The combination of Random Forest and SVMSmote became the new baseline for the later stages. This also forced us to adapt our IWCV function in order to avoid data leakage (see [11]) by first separating, on each fold, the train and test sets, and then oversampling on the train set only.

5. Feature engineering and selection

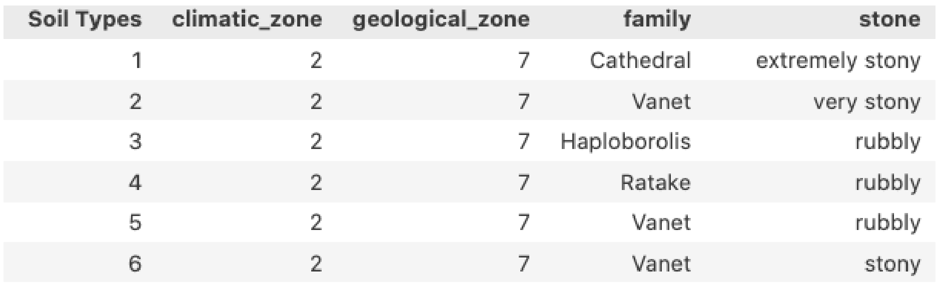
Throughout the feature engineering and selection process, our heuristic has been the following: (i) add all features for which we noticed even a slight improvement (with 0.002 as a threshold) and then (ii) eliminate the features whose deletion increased or did not affect the model's accuracy.

5.1. One-hot reversal

Given our successive tries, we had in mind that our final model would be based on decision trees (whether it be decision trees bagging like in Random Forests or gradient boosting algorithms based on decision trees such as LightGBM). Since these models can usually handle categorical variables quite easily, and in order to reduce complexity, execution time and reduce possible overfitting, we decided to one-hot decode the *Soil Type* and *Wilderness Area*. This entailed a slight increase in our model accuracy.

5.2. Soil Types and Ecological Land Type Units

The first addition of features we had in mind was to take advantage of the Soil Type thanks to the Ecological Landtype Units (ELU), which were explained in depth in the problem statement. Following this statement, we created 4 different features (i) a climatic zone from the first ELU digit, (ii) a geological zone from the second one, and, from the description of these categories, (iii) the complexity of the soil, and (iv) the stone presence in the soil. These new features were later eliminated during the feature selection process.



Soil Types	climatic_zone	geological_zone	family	stone
1	2	7	Cathedral	extremely stony
2	2	7	Vanet	very stony
3	2	7	Haploborolis	rubbly
4	2	7	Ratake	rubbly
5	2	7	Vanet	rubbly
6	2	7	Vanet	stony

Fig 8: Examples of the 4 features created from the the Soil Type variable

5.3. Non Linear Features

In order to capture the complex and non-linear relationships between the features and the target variable, we decided to implement non-linear features, especially on the most important variables according to our baseline ("*Elevation*", "*Id*", "*Horizontal_Distance_to_Roadways*", and "*Horizontal_Distance_To_Fire_Points*").

5.3.1. Interaction Features

The first non-linear features we looked into were interaction features (also called cross features), obtained by multiplying some features with each other one by one. Without much prior knowledge on the subject or leads suggested by the data exploration, our approach has been to **try all possible combinations and keep all the ones for which an accuracy improvement could be observed,** with 0.002 as a threshold compared to our baseline (which was composed of the Random Forest and the SVMSmote at the time).

The ones we finally kept (after the feature elimination process describes below) all concern interactions between the 4 most important features:

- *"Id"* x "*Elevation*"
- *"Id"* x "*Horizontal_Distance_To_Fire_Points*"
- "*Elevation*" x "*Horizontal_Distance_to_Roadways*"
- "*Elevation*" x "*Horizontal_Distance_To_Fire_Points*"
- "*Horizontal_Distance_To_Fire_Points*" x "*Horizontal_Distance_to_Roadways*"

5.3.2. Other non linear Features

In order to take into account nonlinear relationships between features of the dataset, we also introduced new bases of features relying on nonlinear functions, especially the square, square root, and logarithmic functions. Once again, without much prior assumptions, our process was to apply these functions to all the features and keep the ones for which an improvement versus the baseline could be observed. Some of them were later eliminated in the feature selection process, especially the square root transformations which proved here redundant with the logarithm ones. After the feature selection process, we kept three of these nonlinear features: the logarithm transformations of "*Horizontal_Distance_To_Roadways*", "*Horizontal_Distance_To_Fire_Points*" and "*Id*".

5.4. Unsupervised Learning

We decided early to implement some unsupervised learning features, for 2 main reasons. (1) With less than 3% of labeled data in the whole dataset, this problem is very similar to the **"semi supervised framework"** seen in class where unsupervised learning is used before supervised learning in order to **learn the main representations in the data**. (2) Given our use of oversampling techniques and the fact that the pure synthetic data represented 80% of the whole training set, we thought that learning underlying representations in the whole dataset and then applying it to the training set (including the synthetic set) could act as a **form of "regularization technique" on the synthetic data** and help neutralize data that would be too far away from the originals.

Every time, our approach was thus to first **learn some information from the whole dataset in an unsupervised way** (which includes the original training set), and then **propagate this information to the synthetic data.** For simplicity reasons, we only used the original numerical features (excluding engineered features and categorical ones) when doing unsupervised learning. We met **three main limitations** when using this approach. (1) The size of the whole dataset (~580k rows) imposed a technical complexity and prevented us to use non highly-scalable algorithms. (2) On the clustering part, the fact that we used synthetic data limited the number of unsupervised techniques we could use as only "inductive" methods could be used (methods that can be applied to unseen data, as opposed to "transductive" methods, see for instance [12]). (3) As discussed below, the very different scales in the data, especially in the *"Id"* feature vs the others, posed some issues, sometimes leading us to leave this one aside.

5.4.1. Dimension Reduction

The first direction we took to implement unsupervised learning was to use dimension reduction techniques and take the newly created features as additional features. As explained above, we first learned the dimension reduction techniques on the whole original dataset and then applied it to the synthetic dataset, adding new features to both of them. The main techniques we evaluated were **PCA** [13], **t-SNE** [14], and **Truncated SVD** [15].

PCA (Principal Components Analysis) is a linear dimensionality reduction technique that is used to project high dimensional data into lower dimensional space while preserving as much of the variance in the data as possible. It works by finding the directions of maximum variance in the data and projecting the data onto these directions which are known as the principal components. It worked well in terms of results, but we identified that its first component was simply a repeat of the ID column, with the Pearson correlation coefficient between the two being almost equal to 1. This is due to the fact that the "*Id*" has much higher variance than the other features since its scale is much higher (from 0 to almost 600,000) and it is uniformly distributed (one point per unit). In reaction to this phenomenon, we tried several approaches : scale the dataset before running the PCA, drop the "*Id*" column before running the PCA, and drop the first feature created by the PCA after running in on all the columns, which is the approach we finally chose, with 4 components, before finally replacing PCA by the Truncated SVD.

t-SNE (t-distributed Stochastic Neighbor Embedding) is a machine learning algorithm for visualizing high-dimensional data by embedding it into a lower dimensional space (2 or 3). It works by computing pairwise similarities between data points in the high-dimensional space and finding a low-dimensional representation that preserves these similarities as well as possible. t-SNE showed promising results at first, especially due to its nonlinearity and its ability to learn complex patterns, but it turned out to be much too heavy computationally and it was highly inefficient to use during the tuning of other parameters in the last step of our pipeline.

Finally, we decided to use **Truncated SVD** [16] (Singular Value Decomposition) which, like PCA, is a matrix decomposition technique, but which focuses on obtaining the best low rank approximation of the matrix instead of seeking to find the linear combination of features that captures the most variance in the data. Also, unlike PCA, Truncated SVD does not assume that the features have the same scale, which is wrong in our case. Given the better results obtained, we thus decided to replace the PCA features by those obtained by the Truncated SVD. As in the case of PCA, its first component was highly correlated to the '*Id*' column, leading us to drop it. We tested different numbers of components to extract, and it turned out that the optimal amount of components for this technique is four (including the first one we dropped).

5.4.2. Clustering

The second main unsupervised learning direction was to use clustering techniques. Once again, we fitted the clustering algorithms on the whole original dataset and then propagated it to the synthetic data. The main techniques we evaluated were ***k*-means** [17], **DBSCAN** (and its derived HDBSCAN) [18] and **Gaussian Mixture Models** [19].

***K*-means** was a first natural choice for us due to its inductive nature and its ability to scale to large datasets. Here again, however, we met the issue of the "*Id*" feature: given that *k*-means creates the clusters based on euclidean distance and the fact that "*Id*" had a much higher scale and variance than the other features, we discovered that running the algorithm without any preprocessing was virtually equivalent to binning the "*Id*" in k categories. For that reason, after trying to scale the features before running the algorithms, we decided to leave this feature aside when running the algorithm. Regarding the other hyperparameters, we used the *k*-means ++ initialization [20] to try to avoid suboptimal clustering and set the number of initializations to 5 to further reduce this risk. Finally, the number of clusters k was empirically selected based on the ones giving the best results, here to 12.

DBSCAN was promising due to its ability to capture non-flat geometry and uneven cluster sizes, especially compared to *k*-means. However, leaving aside the recurring problem of the "*Id*" which we treated as in the other cases, we faced here the issues of a low scalability of the algorithm and its transductive nature (concretely, the lack of a `.predict` method) that could be applied to the unseen, synthetic data. We tried to tackle this problem by using a k-nearest algorithm as a prediction method and, later, with the HDBSCAN algorithm, which also had the advantage of limiting the number of hyperparameters we had to learn. However, this was found to be quite complex and did not show very good results, leading us to abandon it.

Gaussian Mixture was our final main trial of clustering algorithm, especially due to its ability to capture non spherical clusters as opposed to *k*-means. However, this did not give very interesting results. We used a Bayesian Gaussian Mixture [19] to try to identify the optimal of clusters but this did not give consistent results, probably due to the size of the dataset and to the lack of underlying "Gaussian-like components". After trying a few other clustering methods, especially Mean-Shift and Affinity Propagation, which were too computationally expensive for the size of the dataset, we decided to stop with our original *k*-means.

5.5. Feature Selection

After all these features, our main concern was redundancy that could have increased the model's complexity too much and would have led to overfitting. We thus proceeded in 2 main steps to eliminate as many features as possible :

1. **Eliminate the highly correlated features**, even if they seemed to be slightly increasing the accuracy. This led us in particular to eliminate the first dimension of the PCA and Truncated SVD which, as discussed above, was a virtual copy of the "*Id*" column.
2. **Run a feature elimination algorithm we built** according to the following process: at each step, for all features, calculate the score that could be obtained when removing this feature. Then, removing the feature for which this score would increase the highest and redo the previous step. Finally, stopping when removing any feature would decrease the overall score. This method led us to eliminate several features, especially the soil ones (for which we only kept the basic categories), as well as some cross-features and non-linear transformation features (in particular all the square and square root transformations).

6. Model Selection

Our choice of baseline model evolved while we were dwelling on how to tackle the imbalance issue. Our first baseline model was a **Random Forest Classifier** [21], with a score of 0.79. It seemed to us that it was the best algorithm for a good baseline for our task, and it was easy to implement. Then when we started looking into the class imbalance issue, we discovered that **LightGBM** was providing slightly better results as a baseline, and especially so when using the `class_weights` parameter to specify a custom loss function. LightGBM also had the advantage of being faster to train than our Random Forest, and gave a score around 0.83. Finally, when we decided to adopt oversampling as our main strategy to tackle the class imbalance, we found that Random Forest was working much better than the LightGBM in combination with these techniques. From then on, our baseline for feature engineering and the other stages was composed of a Random Forest Classifier and the SVMSmote oversampling, with a score around 0.85.

As a reminder, Random Forest is an ensemble method which uses decision trees and bagging methods for training. However, contrary to simple bagging and decision tree methods, the Random Forest algorithm adds randomness by selecting a random subset of features when splitting the nodes, rather than searching for the best feature. This increases tree diversity, allowing to balance bias and variance for a more effective model.

Extra Trees [22] makes use of these paradigms and proposes an alternative method on top of the Random Forest method. In this method, trees are made even more random by using random thresholds for each feature, instead of searching for the best thresholds like regular decision trees. This method trades off higher bias for lower variance, and speeds up training considerably compared to regular random forests, as finding optimal thresholds for every feature at each node is time-consuming. Given that Extra Trees were giving us the same scores as the Random Forest but with a much lower training time, we started using it as our baseline. It later proved to have a higher performance when tuned, leading us to finally keep it for our final prediction.

7. Hyper-parameter Tuning

For the hyper-parameter tuning, we relied on the **Optuna** library [23], which is a powerful and flexible library for hyperparameter optimization. Thanks to it, we were able to tune several hyperparameters at the same time in a more efficient and quicker way that a gridsearch would have, using a well structured objective function. The list of the hyperparameters we tuned is the following:

- Oversampling
    - The amount of synthetic data to generate for class 1
    - The amount of synthetic data to generate for class 2
    - Random seed for SVMSmote - see below

- ExtraTree hyperparameters:
    - `min_samples_per_leaf`*:* the minimum amount of samples per leaf in the decision trees
    - `min_samples_per_split:` the minimum amount of samples per split in the decision trees
    - `n_estimators:` the number of estimators used in the bagging process
    - `max_features:` the number of features selected at each split

Although we used fixed random seeds chosen randomly everywhere for reproducibility, we decided to specifically tune the oversampler's one. This is because we noticed especially big changes when changing the seed of the oversampler (around 1% accuracy), which can be explained by the fact that the oversampler creates 80% of the synthetic data, with a big impact whether we "were lucky" or not. For that reason, we decided to optimize this seed as much as possible. The other parts of the code were seeded for reproducibility but randomly or "with a bit of luck" (trying a few seeds manually and selecting one that gave satisfactory results). Note that the Truncated SVD random seed was optimized using the discrete uniform distribution defined in Optuna [23], and not the classical Optuna process.

The final results of the parameters tuning are given in the following table:

Hyperparameters	**Values**
Class 1 - Oversampling amount	60,000
Class 2 - Oversampling amount	93,500
Random Seed - SVMSmote	84
`min_samples_per_leaf`	1
`min_samples_per_split`	2
`n_estimators`	745
`max_features`	None

8. Post-processing

8.1. Cleaning our prediction

As we had noticed during the Exploratory Data Analysis that all the training set is included in the test set, we implemented a last step to automatically replace all predictions for the training data by their actual true value.

8.2. Stacking attempt

Since we had saved all the predictions we had made so far, we were interested in using a stacking method on all of our predictions by applying a hard majority vote - which was much faster to implement than scikit-learn's stacking classifier. The idea was that it would allow us to leverage on the estimators we had implemented in the past, especially the boosting methods we had tried early and which are very different from our final model. However, this was not fruitful in the end and we abandoned it.

9. Conclusion and takeaways

Our **final model score is 0.872**, which represents a **drop of almost 40% in the misclassified instances vs our original score**, and which we are proud of. We have learnt a lot during this challenge, which was the first multiclass classification problem we worked on in depth. The main area we progressed on was how to **tackle class imbalance** between the training and testing sets, leading us to try a variety of techniques and read extensive documentation. The second main area we progressed on was the use of **unsupervised learning techniques**: although our investigations in these areas had in the end a limited impact on our final score, it was the first time we looked in depth into unsupervised learning, especially on the clustering techniques. Finally, one of the challenges this project posed was the question of how to **deal with features of highly different scales** and variances, knowing that here scaling was very little helping - if at all. Here again, we learnt a great deal, especially on the choice to leave some features aside.

Potential refinements and ideas for future projects that we did not have the time to look into include: (i) outlier detection, which could prove especially interesting when used before oversampling; (ii) further look into the oversampling synthetic data, and solve potential issues there (we noticed some strange behaviors, for instance that the oversampler created new categories not existing in the original dataset); (iii) further look into the patterns of our outputs, which we started doing but could have spent more time on; (iv) further look into the feature engineering part, for which our approach was efficient but we feel could have been more rigorous. Overall, we have learnt a lot during this project and we look forward to applying these lessons to new challenges.

10. APPENDIX

10.1. References

[1] Masashi Sugiyama, Matthias Krauledat, Klaus-Robert Müller, “Covariate Shift Adaptation by Importance Weighted Cross Validation”, *Journal of Machine Learning Research* (JMLR), 8:985–1005, 2007.

[2] Zhou, Pei-Yuan & Mo, Wenting & Tian, Chunhua & Li, Li & Rui, Xiaoguang & Wang, Haifeng. (2014). A Two-Stage Classification Framework for Imbalanced Data with Overlapping Labels. Proceedings of 2014 IEEE International Conference on Service Operations and Logistics, and Informatics, SOLI 2014. 10.1109/SOLI.2014.6960749.

[3] LGBMClassier from lightgbm library:
https://lightgbm.readthedocs.io/en/latest/pythonapi/lightgbm.LGBMClassifier.html

[4] ADASYN from Imbalanced learn library:

https://imbalanced-learn.org/stable/references/generated/imblearn.over_sampling.ADASYN.html

[5] SVMSMOTE from Imbalanced learn library:

https://imbalanced-learn.org/stable/references/generated/imblearn.over_sampling.SVMSMOTE.html

[6] Imbalance Learn library, https://imbalanced-learn.org/stable/index.html

[7] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. “Smote: synthetic minority over-sampling technique”. *Journal of artificial intelligence research*, 16:321–357, 2002.

[8] SMOTE from Imbalanced learn library:
https://imbalanced-learn.org/stable/references/generated/imblearn.over_sampling.SMOTE.html#smote

[9] Haibo He, Yang Bai, Edwardo A Garcia, and Shutao Li. Adasyn: adaptive synthetic sampling approach for imbalanced learning. In *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*, 1322–1328. IEEE, 2008.

[10] H. M. Nguyen, E. W. Cooper, K. Kamei, “Borderline over-sampling for imbalanced data classification,” *International Journal of Knowledge Engineering and Soft Data Paradigms*, 3(1), pp.4-21, 2009.

[11] Imbalance Learn: Common pitfalls and recommended practices - Data Leakage
https://imbalanced-learn.org/stable/common_pitfalls.html

[12] Scikit-learn documentation on clustering methods: https://scikit-learn.org/stable/modules/clustering.html

[13] Scikit-learn documentation on PCA:
https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html

[14] Scikit-learn documentation on t-SNE:
httss://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE.html#sklearn.manifold.TSNE

[15] Scikit-learn documentation on Truncated SVD:
https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.TruncatedSVD.html#sklearn.decomposition.TruncatedSVD

[16] Christopher D. Manning, Prabhakar Raghavan and Hinrich Schütze (2008), *Introduction to Information Retrieval*, Cambridge University Press, chapter 18: Matrix decompositions & latent semantic indexing.

[17] Scikit learn documentation on *k*-means

https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html#sklearn.cluster.KMeans

[18] Scikit learn documentation on DBSCAN and HDBSCAN:
https://scikit-learn.org/stable/modules/generated/sklearn.cluster.DBSCAN.html#sklearn.cluster.DBSCAN

[19] Scikit learn documentation on Gaussian Mixtures and Bayesian Gaussian Mixtures:
https://scikit-learn.org/stable/modules/mixture.html#mixture

[20] Arthur, D.; Vassilvitskii, S. (2007). "*k*-means++: the advantages of careful seeding". *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics Philadelphia, PA, USA. pp. 1027–1035.

[21] Scikit-learn documentation on Random Forest:
https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

[22] Scikit-learn documentation on Extra Trees:
https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.ExtraTreesClassifier.html

[23] Optuna:
https://harupy-optuna.readthedocs.io/en/latest/reference/generated/optuna.distributions.DiscreteUniformDistribution.html

10.2. Code

Python
```python
import numpy as np
import pandas as pd

from imblearn.over_sampling import SVMSMOTE
from sklearn.cluster import KMeans
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.decomposition import TruncatedSVD

from utils_2 import concat_Soil_Type, concat_Wilderness_Area, clean_predictor

def preprocess(df):
    # Un-one-hot-encoding the categorical variables
    df = concat_Soil_Type(df, drop_value=True)
    df = concat_Wilderness_Area(df, drop_value=True)
    return df

# Reading the data
df_test = pd.read_csv("test-full.csv")
df_train = pd.read_csv("train.csv")

# Parameters
ovs1 = 60_000
```

```
ovs2 = 93_500
smote_random = 84
min_samples_leaf = 1
min_samples_split = 2
n_estimators = 745

# Preprocessing
df_train = preprocess(df_train)
df_test = preprocess(df_test)

# Separating target and features
X_train = df_train.drop(columns=['Cover_Type'])
y_train = df_train['Cover_Type']
base_cols = list(X_train.columns)

# 1. OVERSAMPLING CLASS 2 AND 1
ovs_strat = {1: ovs1, 2: ovs2}

# Oversampling
svmsmote = SVMSMOTE(sampling_strategy=ovs_strat, random_state=smote_random)
X_train_synth, y_train_synth = svmsmote.fit_resample(X_train, y_train)
X_train_synth = pd.DataFrame(X_train_synth, columns=X_train.columns)


# 2. NON-LINEAR FEATURES
# Cross-features
mult_combinator = {"Horizontal_Distance_To_Fire_Points": ["Id", "Elevation",
"Horizontal_Distance_To_Roadways"],
                   "Elevation": ["Id", "Horizontal_Distance_To_Roadways"]}

new_cols = []
for key in mult_combinator:
    for value in mult_combinator[key]:
        new_cols.append(f"{key} * {value}")
        df_test[f"{key} * {value}"] = df_test[key] * df_test[value]
        X_train_synth[f"{key} * {value}"] = X_train_synth[key] * \
            X_train_synth[value]

# Log features
log_combinator = ['Id', 'Horizontal_Distance_To_Roadways',
                  'Horizontal_Distance_To_Fire_Points']

for label in log_combinator:
    temp = np.where(df_test[label] > 0.5, df_test[label], -10)
    df_test[f"log({label})"] = np.log(temp, where=temp > 0.5)
    temp = np.where(X_train_synth[label] > 0.5, X_train_synth[label], -10)
    X_train_synth[f"log({label})"] = np.log(temp, where=temp > 0.5)
       # The np.where is here to avoid negative values in the log

# 3. UNSUPERVISED LEARNING

# KMEANS - Without ID
km = KMeans(n_clusters=30, n_init=5, init="k-means++", random_state=0)
df_test["kmean"] = km.fit_predict(
    df_test.loc[:, "Elevation":"Horizontal_Distance_To_Fire_Points"])
X_train_synth["kmean"] = km.predict(
    X_train_synth.loc[:, "Elevation":"Horizontal_Distance_To_Fire_Points"])

# TruncatedSVD - With ID and deleting the first column
svd = TruncatedSVD(n_components=4, n_iter=7, random_state=42)
svd_cols = ["SVD_1", "SVD_2", "SVD_3", "SVD_4"]
df_test.loc[:, svd_cols] = svd.fit_transform(
    df_test.loc[:, "Id":"Horizontal_Distance_To_Fire_Points"]
)

X_train_synth.loc[:, svd_cols] = svd.transform(
    X_train_synth.loc[:, "Id":"Horizontal_Distance_To_Fire_Points"]
)

X_train_synth.drop(columns=['SVD_1'], inplace=True)
df_test.drop(columns=['SVD_1'], inplace=True)

# 4. CLASS WEIGHTS - Weighted loss function, calculated using weights.py class_weight
= {
    1: 15, 2: 1, 3: 16,
    4: 157, 5: 50, 6: 28,
    7: 23
}

# 5.CLASSIFYING
clf = ExtraTreesClassifier(n_jobs=-1, max_features=None,
                           min_samples_leaf=min_samples_leaf,
                           min_samples_split=min_samples_split,
                           n_estimators=n_estimators,
                           random_state=69,
                           class_weight=class_weight)


clf.fit(X_train_synth, y_train_synth)
y_pred = clf.predict(df_test)
predictions_df = clean_predictor(y_pred) # Final layer of cleaning
predictions_df.to_csv(f'with_all_features.csv', index=False)
```