Forest Cover Type Classification

Final Report

MAP541 - Kaggle Competition

Camille Epitalon-de Guidis and Martin Quievre

March 2022

Tables of contents

I. Introduction

Context. The US Forest Service (USFS) has determined the actual forest cover type of 15120 forests (30 x 30-meter cell). The cover type is a categorical variable that can take 7 values (labels). For each observation, independent variables were derived from data originally obtained from US Geological Survey (USGS) and USFS data. We are provided with a dataset with 581012 observations with the exact same variables but unlabeled data, (i.e. cover type not provided). The goal is to predict those unknown forest cover based on the various information related to an area.

Objective. Create a machine learning model trained on the training dataset (with labeled data) to *best* predict the cover type of the test dataset (with unlabeled data). In order to measure the performance of a model and to compare it with other models, we use the accuracy (proportion of good predictions among all predictions).

Data. Data is in raw form (not scaled) and contains binary (0 or 1) columns of data for qualitative independent variables (wilderness areas and soil types). In total, there are 13 raw variables:

- `Elevation`
- `Aspect`
- `Slope`
- `Horizontal_Dist_To_Hydro.`
- `Vertical_Distance_To_Hydro.`
- `Horizontal_Distance_To_Road`
- `Hillshade_9am`
- `Hillshade_Noon`
- `Hillshade_3pm`
- `Horizontal_Distance_To_Fire`
- `Wilderness_Area`
- `Soil_Type`

Adopted methodology to reach the objective. For this project, we sequenced our work in several major phases:

1. First the preprocessing of the data (cleaning, enrichment, feature engineering).
2. Then the comparison of the feature choices, taking also into account the tools offered by the dimensionality reduction methods.
3. After having realized that the classification errors occurred essentially within two groups of cover types (between `1`, `2`, `5`, `7` and between `3`, `4`, `6` for instance), we tried to realize a double classification: first separate the observations into 2 more homogeneous groups and then perform a second classification task independently on each of the two subgroups.
4. Then, we tested many different machine learning models, from Logistic Regression to complex H2O models, tuning the hyperparameters.
5. At the end, we kept the set of features and the model which gave us the best results best locally and on Kaggle leaderboard.

Achieved Results. On the training datasets, we reached accuracy scores higher than 90% when overfitting and our best accuracy on Kaggle is around 83.5%.

Resources. This project is supported by a public GitHub repository that contains the entire code we will refer to in this report. This repository can be found at `https://github.com/ceptln/forest_classification`.
Useful information can be found in the `README` file.

II. Explanatory Data Analysis

Context. This part focuses on the exploration of the datasets. It aims at summarizing their main characteristics, checking their quality and properties, visualizing their variables. The idea is to try to see what the data can tell us and which kind of modeling techniques should be used afterwards.

Related resources. The related resources in the repository are the following: `my_toolkit.py` and `kaggle_forest_eda.ipynb`.

a. Primary global analysis

Dataset. The dataset is composed of 56 columns (all int64): the 'Id' of the forest, 10 numerical features, 44 binary columns corresponding to 2 categorical variables OneHotEncoded (`Wilderness_Area` and `Soil_Type`) and the target value (`Cover_Type`). Those dummy columns may require a preprocessing, especially since the statement reports that the `Soil_Type` figures correspond to geographical and climatic realities. Quickly inspecting the number of unique values in each of the 10 "a priori" numerical features, we confirm this initial assumption.

Nulls and N/As. A trivial analysis shows that there is no `NULLs` nor `N/As` value.

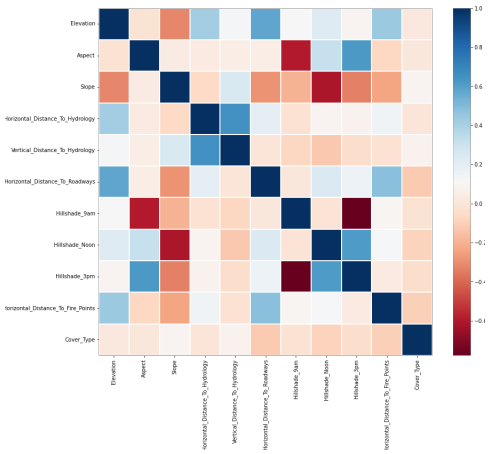
Outliers. To spot potential outliers, we used the *Interquartile Range method* (`my_toolkit.outlier_function`) applied to the numerical variable, stating that any value which does not belong the range

$$[Q_1 - 3 * IQR,\ Q_3 + 3 * IQR], \text{ with } IQR = Q_3 - Q_1$$

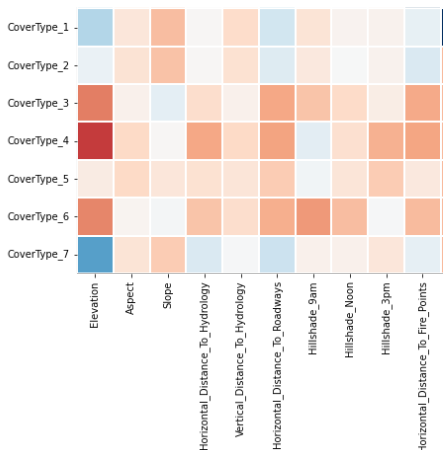
could be one. There is clearly no outlier in the dummy columns (no aberrant observations). As shown in the notebook, we found potential outliers in 5 features. However, looking at their extrema individually, we agreed that there was not outlier in the training dataset, and we kept all rows.

b. Feature-wise analysis

Pearson correlation of numerical features. We plotted the heatmap correlation matrix to study the Pearson correlation between the numerical variables - Figure (a) - and the correlation between each variable and the *OneHotEncoded* target



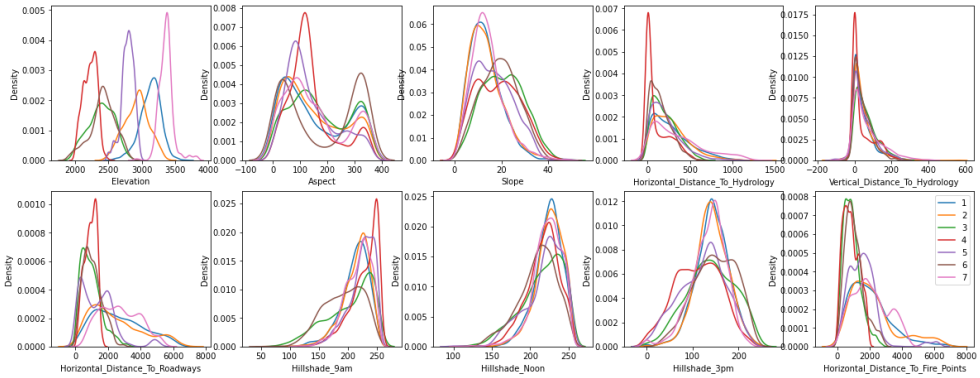
(a) Numerical features correlation



(b) Correlation matrix w.r.t. Cover_Type

value `Cover_Type` – Figure (b). As shown on Figure (b), the 'Elevation' variable seems to have a significant on the target value. This is confirmed by the scatter matrix (see the notebook). We will ponder this more in depth in the analysis of the distribution of each numerical feature. The figure (a) emphasizes a strong negative correlation between `Hillshade_9am` and `Hillshade_3pm` which is not surprising at all and should be addressed in the features selection phase. On the other hand, `Vertical_` and `Horizontal_Distance_To_Hydrology` are correlated (0.65) which make us think that considering the Euclidian distance to hydrology could make sense (Feature Engineering).

Numerical features distribution label-wise. A good way of getting familiar with the dataset and visualizing the different classes is to breakdown by label the numerical features distributions. The Figure (c) clearly confirms that the `Elevation` variable has huge impact on the `Cover_Type`. Indeed, looking at the first chart of Figure (c), we see that we almost surely differentiate a `Cover_Type=4` forest from a `Cover_Type=7` forest just looking at the elevation of the given forest. Unfortunately, this insight does not work with all `Cover_Type` values and the other numerical features are less discriminating.



(c) Numerical features distributions with label-wise breakdown

Categorical features analysis. We first studied the repartition the `Wilderness_Area%` and `Soil_Type%` label-wise. By pivoting them (see notebook), we realized that they could have huge impact on the `Cover_Type`. For example, in the training dataset, `Cover_Type=4` *could* imply `Wilderness_Area=4`. Similarly, all `Soil_Type=37` are `Cover_Type=7`. In addition, the `Soil_Type` distribution is very unbalanced (more than half of the `Soil_Type` stand for less than 10% of the observations).

An artificially balanced training dataset. A deeper analysis of these categorical variables across the train and test dataset highlights the fact that the training set is artificially balanced. Indeed, it is composed of 2160 forests of each cover type. There is no way to closely estimate the number (or proportion) of each cover type in the test set. However, the study of the categorical values showed that they are unevenly distributed between the training and the test dataset. A few examples below:

	Wild_Area1	Wild_Area4	Soil_Type1	Soil_Type2
Count in train set	3568	4681	627	1006
Count in test set	260796	36968	7525	4823
Proportion train set	23.6%	31.0%	4.2%	6.7%
Proportion in testset	44.9%	6.4%	1.3%	0.8%

This suggests that the natural distribution between these 7 forest types is not uniform as suggested by the training dataset. This observation is not surprising, it is even rather rational not to find the homogenous number of each cover type in real life. This will create a bias in the prediction.

III. Preprocessing and feature engineering

Context. This part focuses on the transformation that we decided to make on the original dataset.

Related resources. The related resources in the repository are the following: `my_toolkit.py` and `kaggle_forest_preprocessing.ipynb`.

a. Data enrichment from Soil_Type

ELU matching, climatic and geographic zone encoding. The `Soil_Type_%` binary features all correspond to a specific code (the USFS Ecological Landtype Units – ELUs). The ELU is comprised of four digits: the first one stands for the climatic zone and the second for the geologic zone. We thus created new dummy variables: `ClimZone_%` from 1 to 8 and `GeoZone_%` from 1 to 7. We will see later whether we keep them or not (all or some of them).

ELU enriching with textual data. Each ELU code (and thus `Soil_Type_%`) matches a specific text description given on Kaggle. Each description is unique but there is valuable information common to many descriptions. In particular, we have identified 4 categories created from scratch: family (Catamount, Leighcan, etc), rock type (Cryoborolis, Aquolis, etc), and rock intensity (rubbly, very stony, etc). In a nutshell, we created new dummy features taking the value=1 if the given word appears in the description. Quick example, with `Soil_Type_26=1` which means ELU=7702 and description=Aquolis - Catamount families complex, very stony'. It gives:

```
family_dict = {
    'F_Cathedral' : 'Cathedral',
    'F_Ratake' : 'Ratake',
    'F_Vanet' : 'Vanet',
    'F_Gothic' : 'Gothic',
    'F_Troutville' : 'Troutville',
    'F_Legault' : 'Legault',
    'F_Catamount' : 'Catamount',
    'F_Bullwark' : 'Bullwark',
    'F_Gateview' : 'Gateview',
    'F_Rogert' : 'Rogert',
    'F_Leighcan' : 'Leighcan',
    'F_Como' : 'Como',
    'F_Bross' : 'Bross',
    'F_Moran' : 'Moran'
}
```

(d) Example of dictionary

...	Soil_Type_26	...	F_Catamount	...	R_Aquolis	...	very_stony	...
0	1	0	1	0	1	0	1	0

This data enrichment is performed in the `my_toolkit.enrich_data()` function, based on the dictionaries at the bottom of `my_toolkit.py`. As mentioned before, we will see later which new variable we chose to keep.

b. Feature Engineering

Feature Engineering on numerical features. As mentioned earlier, some feature engineering could be valuable in the initial datasets (linear combinations of features, logarithmic and polynomial transformations). We computed:

- the `Euclidian_Distance_To_Hydrology` to properly capture the distance from hydrology,
- the `Mean_Hillshade` to get a better insight of the global hill shade,
- the mean of all horizontal distances (to hydrology, to fire points, to roadways) to get an idea of the isolation of the forest,
- the mean of `Elevation` and `Vertical_Distance_To_Hydrology`,
- the means of pair-wise horizontal distances (`Mean_Distance_Hydrology_Firepoints`, `Mean_Distance_Hydrology_Roadways` and `Mean_Distance_Roadways_Firepoints`).
- the logarithm transformation of each numerical features (`Log_Elevation`, `Log_Aspect`, etc.)
- the polynomial combinations of numerical features: elevation to the power (x^2, x^3, …) and interactions between features (`x*y`) thanks to the PolynomialFeatures module of scikit-learn, with or without performing a StandardScaling (from scikit-learn too) before the polynomial transformation.

Standardization and normalization. We first implemented a standardization of numerical feature, and an *OrdinalEncoding* of the `Wilderness_Area_%` and `Soil_Type_%` columns (scikit-learn), in particular when using a `LogisticRegression()` model. However, as we will see later in this paper, we quickly realized that a tree-based model would give better scores. As, these models do not require any kind of scaling, we decided not to use standardization nor normalization.

c. Feature selection

The features selection framework. We created a Python class (`ClassifTools`) to deal with all these new parameters that we created and test different combinations of them – Figure (e). Those class properties are used in particular in the function `cl.enrich_data()` which performs the desired preprocessing on the training and the testing set. This setup enables us to train and evaluate our model on preprocessed datasets going from 15 to 101 variables (and even more since we can add as many polynomial terms and degrees as we like). It goes without saying that finding the best combination of features is no easy task.

```
# Class parameters
df_train = df_train
df_test = df_test
model = LogisticRegression()
add_eng_features = True
columns_to_convert_to_log = ['Elevation', 'Aspect
polynomial_degree = 2
columns_to_polynomial = ['Hillshade_9am', 'Hillsh
add_climate = True
add_geographic = False
add_family = True
add_rocky = True
add_stony = True
keep_initial_rows = False
decomposition = None
columns_to_decomp = None
columns_to_drop = ['Id', 'ClimZone_3', 'GeoZone_5
random_state = 2

# Class initialization
cl = ClassifTools(df_train, df_test, model,
    add_eng_features,
    columns_to_convert_to_log,
    polynomial_degree, columns_to_polynomial,
    add_climate, add_geographic,
    add_family, add_rocky, add_stony,
    keep_initial_rows, columns_to_drop,
    decomposition, columns_to_decomp,
    random_state)
```

(e) ClassifTools class initialization

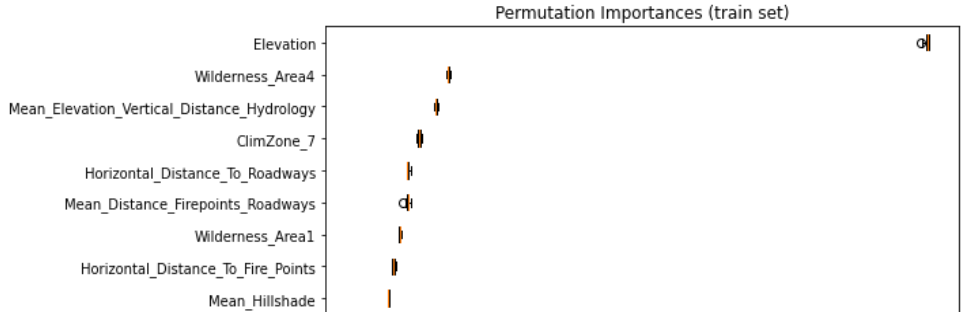
The best selection of variables. To determine whether a variable has an impact on the target value, we used 3 different methods.

1. Plotting the correlation matrix

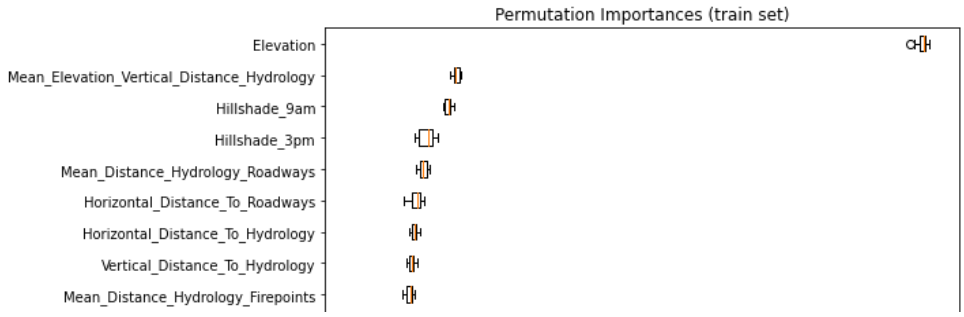
We have already seen the first method: as shown by Figure (c), plotting the correlation matrix of each variable with respect to each one of the class values (`Cover_Type=1`, `2`, …, `7`) gives us valuable learnings. We can assume that the more diverse the colors are between the different cover types, the more discriminating the variable is. This may not be very rigorous reasoning, but it has paid off on this project. For instance, on Figure (c), we clearly see that the value of Elevation feature is much more significant than the one the `Vertical_Distance_To_Hydrology`. This method provided us with a list of insignificant variables, such as `ClimZone3` and `F_Gothic`. We thus performed this correlation matrix analysis on a big number of variable and eliminated the less insightful ones. If this method is less rigorous than the two others, it has a major advantage: it is agnostic of the chosen model, i.e. it makes it possible to make a first sorting and to separate from the variables which bring almost no information before choosing the model.

2. Performing a Permutation Importance study

The scikit-learn module 'inspection' provides us with the method called `permutation_importance()`. This inspection technique computes the permutation feature importance which is the decrease in the model score when a single feature value is randomly shuffled. It breaks the relationship between the feature and the target, thus the drop in the model score is indicative of how much the model depends on the feature. The great thing is that it is compatible with any kind of model (from sklearn or not).



(f) Permutation Importance for LightGBM



(g) P. I. for LogisticRegression

3. A manual GridSearch on those class properties and accuracy comparison

On a larger scale, this method consists in varying the properties of the class in loops, to compute and store the model score for each combination. This was done in the `kaggle_forest_notebook.ipynb` notebook and showed for example that adding the `GeoZone_%` was not a valuable move with respect to the model score. Therefore, in our final model, we set `add_geographic=False`.

Dimensionality reduction. Finally, in this preprocessing part we performed some dimensionality reduction transformation such as a *Principal Component Analysis* (PCA) and a *Multiple Correspondence Analysis* (MCA), in particular to deal with the binary features of the initial datasets (`Wilderness_Areas` and `Soil_Type_%`). Rather than extracting insights from the soil type and then deleting the initial columns, we tried to see if there would not be a way of keeping their raw information while decreasing the dimensionality (in other words, represent them in a low-dimensional Euclidean space). We had read here and there PCA it had no effect on categorical variables, let alone dummies, but we wanted to know for sure… and it was not a lie. Performing a PCA on the `Soil_Type` ended up on worst results than when properly deleting the initial columns. However, MCA ended up on result that were very close to the ones obtained without dimensionality reduction. However, MCA did not manage to outperform our best model as shown in the table below.

Method used	`LogisticReg`	`ExtraTrees.`	`LGBMClassifier`
None	0.4848	0.8849	0.8918
PCA	0.4748	0.8601	0.8287
MCA	0.4848	0.8919	0.8634

As we explained in `kaggle_forest_dimensionality_reduction.ipynb`, we tested PCA and MCA with a large range of components (`n_components`): from 3 to 25. Note that we tested dimensionality reduction combined with 3 models (`LogisticRegression`, `ExtraTreesClassifier` and `LGBMCClassifier`) with hyperparameters set by default. We used the preprocessed datasets (enriched).

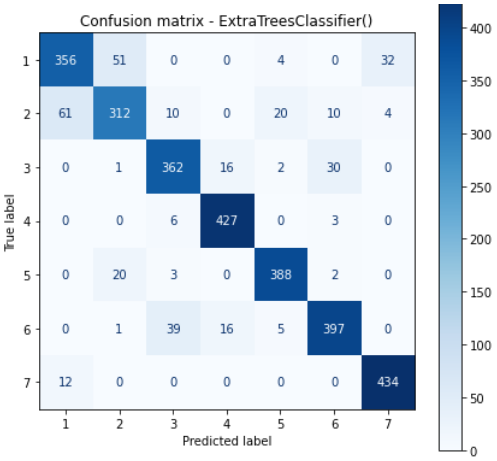
Interpretation. Our interpretation of such results is that down the road the information given by the variables that we extracted from the ELU description (and thus the soil type) is more valuable than the information given by the raw soil type data.

IV. Double Classification strategy

Context. This part focuses on a strategy we tried to implement to bypass and take advantage of a problem identified in early tests.

Related resources. The related resources in the repository are the following: `my_toolkit.py` and `kaggle_forest_double_classif.ipynb`.

Confusion matrix learnings. When performing a classification with multiple categories (labels), one of the most valuable tools for analyzing model errors or biases is to plot the confusion matrix. Basically, on this confusion (or error) matrix, each row of the matrix represents the instances in an actual class while each column represents the instances in a predicted class. While testing several models, we quickly realized the existence of a pattern in these confusion matrices. Indeed, whatever the model (as for as the accuracy is above 75%), that the classification errors occurred essentially within two groups of cover types (between `1`, `2`, `5`, `7` and between `3`, `4`, `6` for instance).



(h) Confusion matrix (ExtraTrees)

Idea. To the extent that these groups are more homogeneous, creating a model dedicated to each group and training it with specific observations may seem like a promising idea. Of course, this is done at the expense of the number of observations, because we train our model on half as many data each time.

Double classification implementation. As we explained, we first development a model able to split our observation between the 2 groups (group 1: cover types `1`, `2`, `5`, `7` / group 2: cover types `3`, `4`, `6`). After different tests (see `kaggle_forest_double_classif.ipynb`) we managed to reach a satisfying accuracy of 0.9894 on our validation set (with `LGBMClassifier()`, no tuning). Then, we trained different models on each group. Locally, we got 0.8756 accuracy on group 1 and 0.929 on group two (still on validation sets). Once our 3 models were training, we used the first one to separate the test dataset between two groups, and applied the two other models to the dedicated groups.

Results. If the results seemed promising on the training dataset, they turned out to be less convincing once our predictions uploaded on Kaggle. Our best accuracy score using this method is 0.804 which is below our best model score.

V. Model comparison

Context. The preprocessing is done and we are now going to compare the performances of different models based on various sets of features and hyperparameters tuning.

Related resources. The related resources in the repository are the following: `my_toolkit.py` and `kaggle_forest_model_comparison.ipynb`.

Approach. We split this phase in two main parts. First, comparing the performances of the different models on the raw data and on the best combination of features (for each model) both locally and on the test set (via Kaggle). In a second step, we measure the improvements in terms of accuracy of

hyperparameters tuning for each promising models, both locally and on the test set (via Kaggle).

a. Preprocessing performance

Raw data vs. preprocessed data performance comparison. For this model comparison, we chose a large range of classifiers:

- `LogisticRegression()`,
- `KNeighborsClassifier()`,
- `RandomForestClassifier()`,
- `HistGradBoostClassif.()`,
- `ExtraTreesClassifier()`,
- `LGBMClassifier()`,
- `CatBoostClassifier()`,
- `XGBClassifier()`

Why these models? The explanatory dataset analysis phase clearly showed that there were no proper linear relationships between the different features. We decided to test `LogisticRegression()` more as a baseline estimator. Given the input variables (both quantitative and qualitative), be quickly headed for tree-based, ensemble-based and boosting-based models which are almost always the most performant ones in this setting. Finally, a deep learning model would not be appropriate here as the number of observations as well as the number of parameters is not so large and we are not dealing with time series (for which RNNs can be interesting in some settings).

We thus measured the performance (accuracy score) of the chosen models both on raw data and the best combination of features.

Model	Initial features (baseline performance)		Best combination of features	
-	Locally	Kaggle	Locally	Kaggle
`LogisticRegression`	0.3747	Not tested	**0.3912**	0.2414
`KNeighborsClassifier`	0.7497	Not tested	**0.7629**	Not tested
`RandomForestClassif.`	0.8872	Not tested	**0.8915**	Not tested
`HistGradientBoost.`	0.8929	0.7619	**0.8985**	**0.8019**
`ExtraTressClassifier`	0.8849	0.7931	**0.9024**	**0.81411**
`LightGBM`	0.8918	0.78683	**0.8998**	**0.80440**
`Catboost`	0.8760	Not tested	**0.8852**	Not tested
`XGBoost`	0.8918	0.8054	**0.8972**	**0.8088**

Note that the best combination of features for each model was done in `kaggle_forest_model_comparison.ipynb`.

Comments. The first striking thing is that our models are more performant (in terms of accuracy) on the preprocessed data than on the raw data both locally and on the testing set, although the improvement is not as huge as we could have

imagined. Moreover, we clearly see that our all the models that we tested both locally and on the platform overfit the data. This is not very surprising given the sizes of the training and test sets (composed of almost 40 times more observations).

AutoML. We also tried AutoML models from the `Tpot` and `H2O` Python packages. The `H2O` AutoML model recommended the use of the following model: `leaderStackedEnsemble_AllModels_4_AutoML_5_20220328_123855`. It gave good results (0.81656 on Kaggle) but was outperformed by the tuned classical models (see below).

b. Hyperparameters tuning performance

Hyperparameters tuning. Now that we have our best set of features for each chosen model, it is time to pick the three *best* ones (in terms of accuracy) and to tune their hyperparameters to keep improving the classification task. We now focus on `HistGradientBoostingClassifier()`, `ExtraTressClassifier()` and `LGBMClassifier()` only.

Model	Default hyperparameters		Tuned hyperparameters	
-	Locally	Kaggle	Locally	Kaggle
`HistGradientBoost.`	0.8985	0.8019	**0.9032**	**0.8078**
`ExtraTreesClassifier`	0.9024	0.8141	**0.9054**	**0.8233**
`LightGBM`	0.8998	0.8044	**0.9163**	**0.8353**

Similarly, note that the hyperparameters tuning of each model was done in `kaggle_forest_model_comparison.ipynb`, using `GridSearchCV` and `RandomSearchCV` in particular.

Comments. Hyperparameters tuning paid off. The model that we choose the be the final one is `LGMBClassifier()` with the following set of hyperparameters:

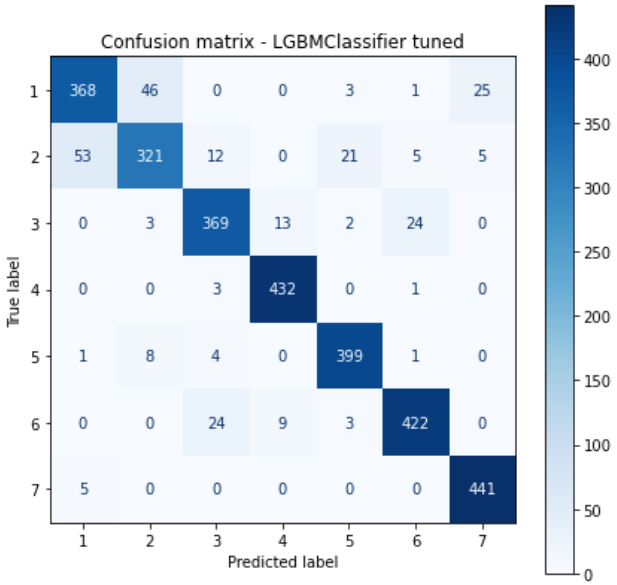
- `learning_rate=0.01`
- `n_estimators=2048`
- `num_leaves=1024`
- `max_depth=32`

This setting enabled us to gain more than 0.2 of accuracy on the testing dataset. By increasing the n_estimators and num_leaves hyperparameters, we clearly increasing the complexity of the model but also the risk of overfitting. `n_estimators` specifies the number of boosting iterations (trees to build). Basically, the more trees the more accurate the model can be (at the potential cost of longer training time and overfitting). And as `n_estimators` increases, the `learning_rate` should be larger. According to the documentation, one simple way is that `num_leaves = 2^(max_depth)` however, in lightgbm a leaf-wise tree is deeper than a level-wise tree.

VI. Summary & Conclusions

Summary. To summarize our work, we first preprocessed the data by cleaning and enriching the data. We performed some feature engineering which improved our results. We tested an alternative strategy consisting of performing a double sequential classification, grouping the 7 classes in 2 larger groups treated independently. It did give interesting accuracy scores but not as good as the ones obtained by tuning single machine learning models such as `LGBM` or `HistGradientBoostingClassifier`.

Best model and achieved results. The model that we have chosen is `LGMBClassifier()` with a high number of trees (2048) and a rather slow learning rate (0.01). The training datasets, we reached accuracy scores higher than 0.9163 when overfitting and our best accuracy on Kaggle is around 0.8353. You can find just below the confusion matrix of the local predictions on the validation set.



(i) Confusion matrix of final model

On Kaggle, we selected two prediction submissions: the one corresponding to this previous model, and the one resulting from our double classification approach.

Resources. Again, this project is supported by a public GitHub repository that contains the entire code we will refer to in this report. This repository can be found at `https://github.com/ceptln/forest_classification`.