Earthquake damage prediction

Q1 2021-2022

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Index

Motivation and general description of the problem to be analyzed

The motivation for doing this project was primarily an urge to find an original dataset that could interest us and that we could understand each of the variables shown in the data matrix and their technical terms. We mulled over other interesting datasets about the poison level of mushrooms, mobile price classification, Parkinson's disease detection and even human genetic variant classification. But we decided to move away from the typical dataset websites like Kaggle and we found a very interesting one: Data Driven. This is a website oriented to organizing data science competitions that users sign up for fun or to create an actual project like ours. We came across one of the competitions titled Richter's Predictor: Modeling Earthquake Damage, which the aim is to predict the level of damage of a building when there has been an earthquake in the area. We thought it would be an interesting topic, so we decided to download the dataset as if we were signing up for the competition.

Description of the data

Data collection process

Our dataset comes from the Driven Data website:
https://www.drivendata.org/competitions/57/nepal-earthquake/page/136/
Data Driven is an organization that hosts data science competitions about different topics. One of them, a study on the damage of a building that was hit by an earthquake, which is where we got the dataset. In order to get the data we had to sign up for the competition and then we were provided with 3 *.csv* files, one with the values, one for the labels (which had the damage level per building id) and one with a trained dataset.
This data can be preprocessed directly using the methods seen in class with Python with no additional steps in between.

What are the data about

The dataset selected responds to information about a study on the level of damage that a building can have when hit by an earthquake. The aim of this study is to try to predict the ordinal variable *damage_grade* which has 3 possible values depending on the level of damage:

- **1** - Represents a low damage level
- **2** - Represents a medium amount of damage
- **3** - Represents a high level of damage (almost complete destruction of the building)

These values represent the 3 classes that this dataset has. As we will see later, there is a major imbalance between these classes, specifically with class 2 which stands out from the rest. We will balance the dataset accordingly to get better results and precision.

Data matrix

Our data matrix has 39 columns corresponding to the variables, one of them being *building_id* which is a unique random identifier. The other 38 correspond to different characteristics of the building and the piece of land where it is built. There are 8 numerical variables, 22 binary variables and the remaining 8 are categorical variables. Binary variables have a possible numerical value of 0 or 1 and categorical variables have been obfuscated with random lowercase ASCII characters as their factor value. It does not matter the meaning of these values in the categorical variables, as we are not going to do an analysis on them and the main objective is to predict the level of damage and find precision for our dataset.

Regarding the rows, we have more than 250.000, which correspond to each building that has been subjected to this study. As we said, there is a huge imbalance between buildings that have been affected with a medium amount of damage and buildings that either had low or very high values of damage. We will see in the following preprocessing sections how we dealt with this imbalance problem and the methods we used.
Finally, here is a description of the columns taken from the Data Driven website:

- **geo_level_1_id, geo_level_2_id, geo_level_3_id** (type: int): geographic region in which building exists, from largest (level 1) to most specific sub-region (level 3). Possible values: level 1: 0-30, level 2: 0-1427, level 3: 0-12567.
- **count_floors_pre_eq** (type: int): number of floors in the building before the earthquake.
- **age** (type: int): age of the building in years.
- **area_percentage** (type: int): normalized area of the building footprint.
- **height_percentage** (type: int): normalized height of the building footprint.
- land_surface_condition (type: categorical): surface condition of the land where the building was built. Possible values: n, o, t.
- **foundation_type** (type: categorical): type of foundation used while building. Possible values: h, i, r, u, w.
- **roof_type** (type: categorical): type of roof used while building. Possible values: n, q, x.
- **ground_floor_type** (type: categorical): type of the ground floor. Possible values: f, m, v, x, z.
- **other_floor_type** (type: categorical): type of construction used in higher than the ground floors (except for the roof). Possible values: j, q, s, x.
- **position** (type: categorical): position of the building. Possible values: j, o, s, t.
- **plan_configuration** (type: categorical): building plan configuration. Possible values: a, c, d, f, m, n, o, q, s, u.
- **has_superstructure_adobe_mud** (type: binary): flag variable that indicates if the superstructure was made of Adobe/Mud.
- **has_superstructure_mud_mortar_stone** (type: binary): flag variable that indicates if the superstructure was made of Mud Mortar - Stone.
- **has_superstructure_stone_flag** (type: binary): flag variable that indicates if the superstructure was made of Stone.
- **has_superstructure_cement_mortar_stone** (type: binary): flag variable that indicates if the superstructure was made of Cement Mortar - Stone.
- **has_superstructure_mud_mortar_brick** (type: binary): flag variable that indicates if the superstructure was made of Mud Mortar - Brick.
- **has_superstructure_cement_mortar_brick** (type: binary): flag variable that indicates if the superstructure was made of Cement Mortar - Brick.
- **has_superstructure_timber** (type: binary): flag variable that indicates if the superstructure was made of Timber.
- **has_superstructure_bamboo** (type: binary): flag variable that indicates if the superstructure was made of Bamboo.
- **has_superstructure_rc_non_engineered** (type: binary): flag variable that indicates if the superstructure was made of non-engineered reinforced concrete.
- **has_superstructure_rc_engineered** (type: binary): flag variable that indicates if the superstructure was made of engineered reinforced concrete.
- **has_superstructure_other** (type: binary): flag variable that indicates if the superstructure was made of any other material.
- **legal_ownership_status** (type: categorical): legal ownership status of the land where the building was built. Possible values: a, r, v, w.
- **count_families** (type: int): number of families that live in the building.

- **`has_secondary_use`** (type: binary): flag variable that indicates if the building was used for any secondary purpose.
- **`has_secondary_use_agriculture`** (type: binary): flag variable that indicates if the building was used for agricultural purposes.
- **`has_secondary_use_hotel`** (type: binary): flag variable that indicates if the building was used as a hotel.
- **`has_secondary_use_rental`** (type: binary): flag variable that indicates if the building was used for rental purposes.
- **`has_secondary_use_institution`** (type: binary): flag variable that indicates if the building was used as a location of any institution.
- **`has_secondary_use_school`** (type: binary): flag variable that indicates if the building was used as a school.
- **`has_secondary_use_industry`** (type: binary): flag variable that indicates if the building was used for industrial purposes.
- **`has_secondary_use_health_post`** (type: binary): flag variable that indicates if the building was used as a health post.
- **`has_secondary_use_gov_office`** (type: binary): flag variable that indicates if the building was used as a government office.
- **`has_secondary_use_use_police`** (type: binary): flag variable that indicates if the building was used as a police station.
- **`has_secondary_use_other`** (type: binary): flag variable that indicates if the building was secondarily used for other purposes.

Description of pre-processing of data

In this section we are going to treat our data with methods that are not tailored specifically to any type of model. If further data pre-processing is necessary for a model, it will be done in its section.

This dataset is from a competition which has a test dataset already partitioned without any labels, because the challenge is to predict them. That means the changes have to be replicated both in the training dataset (the one with labels) and the test dataset.

We'll sample without replacement 20000 rows randomly from the training dataset, with the seed '1234', which we'll use in all the random processes in this section. If we check the distribution of the target variable before and after the sampling, we can see that it doesn't change significantly, which is what we'd expect from a random sampling:

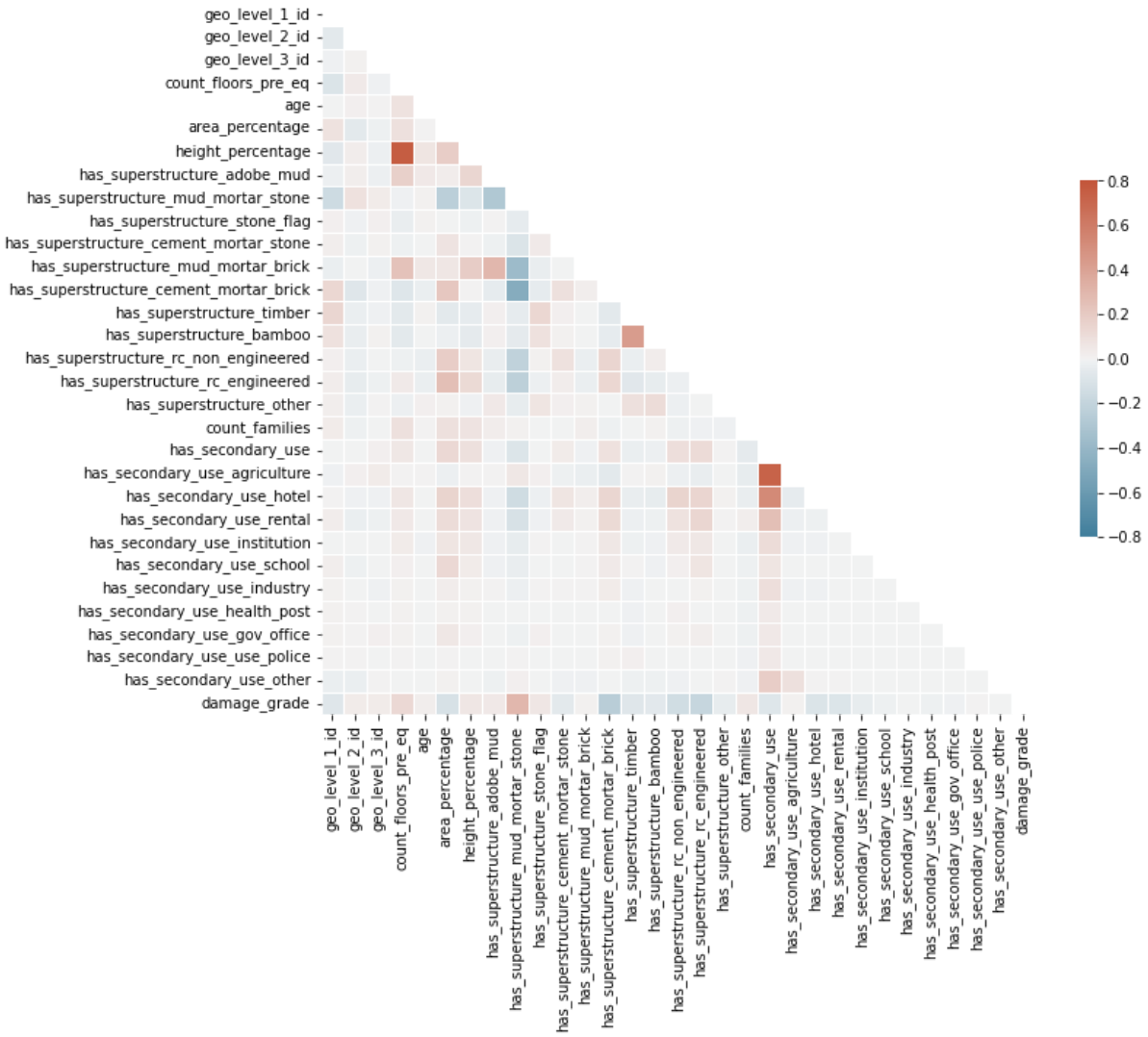
```
Orignal distribution:
    'damage_grade' == 1 -> 9.64%
    'damage_grade' == 2 -> 56.89%
    'damage_grade' == 3 -> 33.47%
After sampling distribution:
    'damage_grade' == 1 -> 9.63%
    'damage_grade' == 2 -> 57.27%
    'damage_grade' == 3 -> 33.1%
```

Feature selection

We'll start by selecting the relevant columns (or features) by using different methods in conjunction.

By correlation

We compute the correlation matrix and we visualize it with the `heatmap` method of the *seaborn* library. This produces the following result:



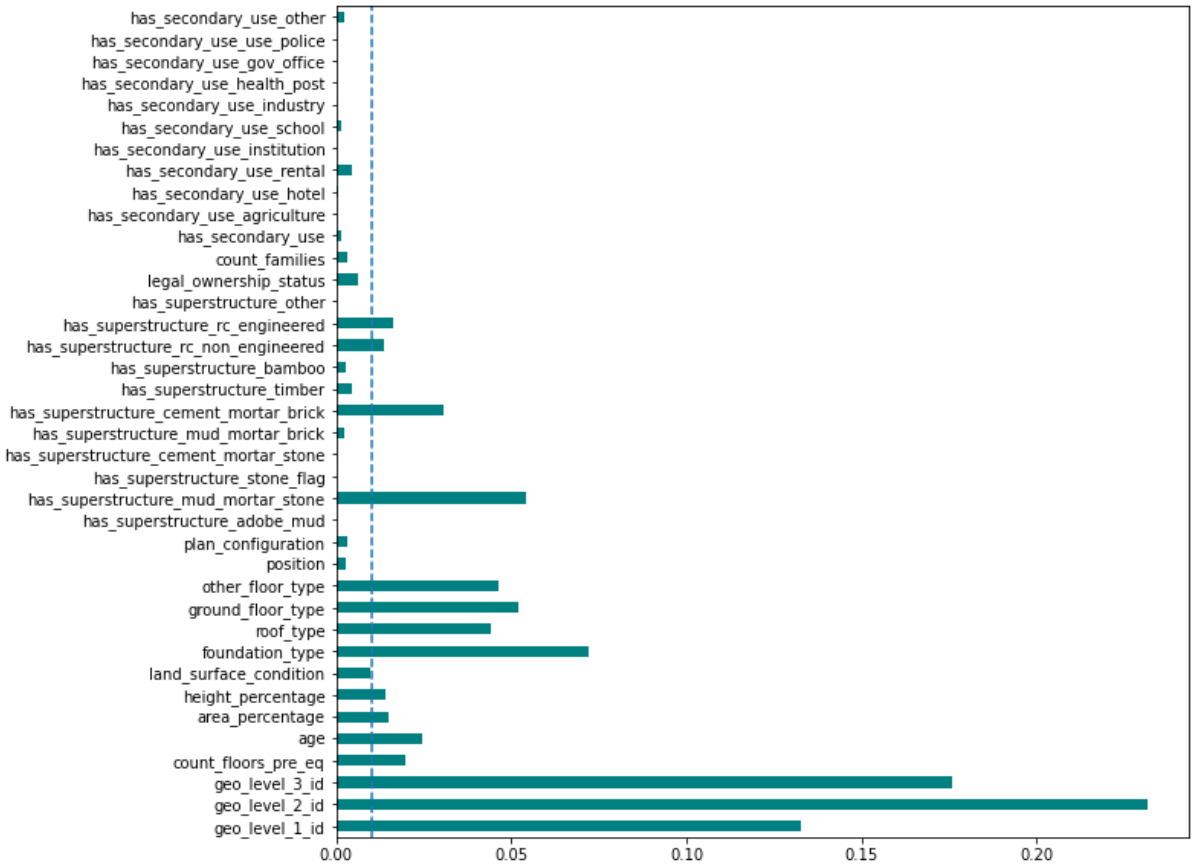
Then, we filter the pairs with correlation higher than a chosen threshold, which in this case, as per research in various sources, is recommended to be in the range between 0.5 and 0.7. We'll use 0.5 to see the maximum number of pairs and then decide which features to keep and to exclude in conjunction with the following subsections.

```
There's 3 feature pairs which are highly (>0.5) correlated, which are:
                  first                          second     value
0  count_floors_pre_eq                 height_percentage  0.761464
1    has_secondary_use  has_secondary_use_agriculture  0.730049
2    has_secondary_use        has_secondary_use_hotel  0.530272
```

The **0** pair is evidently strongly correlated, as both relate to height of the building (more floors imply more height). The other two pairs are strongly correlated because the first of the pairs, '`has_secondary_use`', is a binary feature, which if it's one, then the rest of the '`has_secondary_use_****`' columns, which represent a multiple choice question, will have a 1 in some of them. It seems that for the majority of the buildings that have a secondary use, this secondary use is 'agriculture' and/or 'hotel'.

By Information Gain

We are going to use the '`mutual_info_classif`' method from the *sklearn* library, which returns an array with the estimated mutual information between each feature and the target. To call this method, we need to slightly transform our dataset by encoding the categorical features' values to numerical values. We'll use the class `LabelEncoder` from the *sklearn* library. The following plot shows the results:



We can see that there's a vertical discontinuous blue line that represents the 0.01 value. This threshold is arbitrary and was chosen because it included and excluded a reasonable number of features, as it seems to divide more interesting features pretty well. The included and excluded features are the following:

```
Selected features:
['geo_level_1_id',
 'geo_level_2_id',
 'geo_level_3_id',
 'count_floors_pre_eq',
 'age',
 'area_percentage',
 'height_percentage',
 'foundation_type',
 'roof_type',
 'ground_floor_type',
 'other_floor_type',
 'has_superstructure_*']
```

```
Excluded features:
['land_surface_condition',
 'position',
 'plan_configuration',
 'legal_ownership_status',
 'count_families',
 'has_secondary_use',
 'has_secondary_use_*']
```

Note that not all the columns relating to the '`has_superstructure_*`' are above the threshold values, but we're still keeping them because some of the columns are significant, and we prefer to either keep or drop them all, as they relate to the same topic / question.

By Fisher score

In this section we will evaluate the features by their Fisher scores ranking. Ideally, we would do this by the Fisher score itself, but we'll have to do solely with the ranking, it seems due to problems in the implementation of the library, which doesn't let us choose the score mode. For this, we'll use the '`fisher_score`' method from the *skfeature* library. This method ranks the features by their Fisher scores in a descending order.

foundation_type	37
has_secondary_use_rental	36
has_secondary_use_institution	35
land_surface_condition	34
height_percentage	33
roof_type	32
has_superstructure_mud_mortar_brick	31
has_secondary_use_agriculture	30
position	29
count_floors_pre_eq	28
age	27
has_secondary_use_hotel	26
has_superstructure_bamboo	25
has_secondary_use	24
has_superstructure_stone_flag	23
geo_level_1_id	22
has_superstructure_rc_non_engineered	21
has_superstructure_rc_engineered	20
has_superstructure_timber	19

geo_level_2_id	18
has_superstructure_mud_mortar_stone	17
has_superstructure_cement_mortar_stone	16
area_percentage	15
geo_level_3_id	14
count_families	13
plan_configuration	12
has_superstructure_cement_mortar_brick	11
has_superstructure_adobe_mud	10
has_superstructure_other	9
other_floor_type	8
ground_floor_type	7
legal_ownership_status	6
has_secondary_use_school	5
has_secondary_use_gov_office	4
has_secondary_use_industry	3
has_secondary_use_health_post	2
has_secondary_use_other	1
has_secondary_use_use_police	0

Features selected / dropped

We are going to base this decision mainly on the Information Gain criteria, discarding features by the correlation criteria and we'll use the Fisher score ranking to validate the selection.

So, the selected features are the ones that were above 0.01 information gain coefficient except one of these features, which is '`height_percentage`', due to the strong correlation between it and the other selected feature, '`count_floors_pre_eq`'. We excluded the former rather than the latter because it has a lower information gain coefficient, and we set it as the predominant criteria.

In conclusion, the final selected features are the following:

```
['geo_level_1_id',
 'geo_level_2_id',
 'geo_level_3_id',
 'count_floors_pre_eq',
 'age',
 'area_percentage',
 'foundation_type',
 'roof_type',
 'ground_floor_type',
 'other_floor_type',
 'has_superstructure_adobe_mud',
 'has_superstructure_mud_mortar_stone',
 'has_superstructure_stone_flag',
 'has_superstructure_cement_mortar_stone',
 'has_superstructure_mud_mortar_brick',
 'has_superstructure_cement_mortar_brick',
 'has_superstructure_timber',
 'has_superstructure_bamboo',
 'has_superstructure_rc_non_engineered',
 'has_superstructure_rc_engineered',
 'has_superstructure_other']
```

The Fisher score ranking gives us the following results:

```
Media del rango de todas las columnas:            18.5
Media del rango de las columnas seleccionadas:   20.14
---- Columnas seleccionadas que según el test de Fisher están más del
lado de las excluidas:
      ['geo_level_3_id',
       'area_percentage',
       'ground_floor_type',
       'other_floor_type',
       'has_superstructure_adobe_mud',
       'has_superstructure_cement_mortar_stone',
       'has_superstructure_cement_mortar_brick',
       'has_superstructure_other']
Media del rango de las columnas excluidas:       16.25
---- Columnas excluidas que según el test de Fisher están más del lado
de las seleccionadas:
      ['land_surface_condition',
       'position',
       'has_secondary_use',
       'has_secondary_use_agriculture',
       'has_secondary_use_hotel',
       'has_secondary_use_rental',
       'has_secondary_use_institution']
```

Firstly, we can see that the means of the ranks of the selected and the excluded features are different, with the former being higher, which totally makes sense. We expected slightly more difference, but the problem with this comparison is that the mean is not done by score but by rank, and then we don't know if this difference is more significant as the rank wouldn't represent big jumps between ranks.
Either way, this shows the selected features which are even below the mean of the rank of the excluded features and vice versa. The majority of the features represented in this mismatches are part of the multichoice questions, which makes sense, since some have been selected despite not having significant information gain, and vice versa.It would be interesting to analyze the rest, but the lack of score coefficients makes this difficult so we have to chuck this up to the differences between the criterias.

One Hot Encoding

Once we have selected which columns we are interested to keep, we encode the categorical variables to various binary ones. We get 25 columns from the 17 we kept in the previous section.

Variance Threshold

This subsection would be more interesting if we had normalized or standardized the data in the Pre-processing section, but we decided to do that depending on the model we are training. So we will only discard features which have no variance, i.e. only have one value in all the rows. We do this with the *sklearn* library's `VarianceThreshold` class.
Luckily, we didn't find any column with this characteristic.

Mean encoding

Mean encoding represents a probability of your target variable, conditional on each value of the feature. To do this, it calculates the mean of the target variable value grouped by each value of the feature. Our target variable is categorical, but luckily is also ordinal, so we don't need to hot encode it, as the mean will make sense.
Then, we will do this for each geographic feature. The training dataset doesn't present any problem, but the test dataset does. This is the percentages of missing values:

```
    geo_level_1_id_enc: 0.0%
    geo_level_2_id_enc: 0.0058%
    geo_level_3_id_enc: 0.38%
```

That's because, as it doesn't have labels, if there's a geographic id without any representation in the training dataset, we can't compute its mean. In this case, we will assign to it the mean of the geographic id that contains the problematic geographic id, as the three geographic features identify geographic areas by increasing specificity, i.e. the former contains the latter. If the mean for the first geographic id is missing we would assign the mean of the whole dataset. Luckily, there's none like that.

Balancing

This dataset is significantly unbalanced, which may lead to bias in the models. As we have rows to spare, we will undersample the larger classes to the same number of rows as the smaller class. We'll do this with the '`RandomUnderSampler`' class from the imblearn module. The results are the following:

```
-------- Before balancing -> 20000 rows in total
    'damage_grade' == 1 -> 9.63%
    'damage_grade' == 2 -> 57.27%
    'damage_grade' == 3 -> 33.1%
-------- After balancing -> 5778 rows in total
    'damage_grade' == 1 -> 33.33%
    'damage_grade' == 2 -> 33.33%
    'damage_grade' == 3 -> 33.33%
```

Saving the data

Different pre-processing steps may affect differently the methods we will investigate in this report, so we'll save the data in different steps in different files, with the following combinations (they all contain Hot Encoding, as it is necessary):

Mean encoding →	Without	With
Balancing ↓		
Without	*train_values_Prep.csv* *test_values_Prep.csv* *train_labels_Prep.csv*	*train_values_Prep_ME.csv* *test_values_Prep_ME.csv*
With	*train_values_Prep_Bal.csv* *train_labels_Prep_Bal.csv*	*train_values_Prep_ME_Bal.csv*

Evaluation criteria of data mining models

To evaluate our models, we have chosen to split 70 % of our data as training and the rest as the test data. This was done, so the model had a large enough sample size to train the models but keeping our validation large enough to reduce variance.

In the cases that we have used cross validation, we decided to use , generally, 10 fold cross validation except in the case of Support Vector Machines (SVM), for which we used 4. The reason for which we choose 10, k-fold cross-validation with moderate k values (10-20) reduces the variance while increasing the bias. And the decision to use 4 in SVM was due to the fact that for our dataset the computational time was quite large, our early tries with 10-fold cross validation resulted in 8 hours of computation and resulted in an error.

As the metric used for evaluation, it was decided by the group to use the accuracy of the models after training to determine how effective they were. Because our dataset was unbalanced with the majority of elements being class two, we had always to be careful that a good accuracy was not due to over-representation of this class.

Moreover because in the preprocessing step, we created a balanced dataset of our data. We reserved the right to use this dataset as training for the models, in case that they had a significant boost in the accuracy of the model.

Execution of machine learning methods

Naive-Bayes

To perform this method we have decided to make the predictions with respect to the 4 datasets that we have done in the preprocessing. For each of these datasets two predictions will be made, the first one by fit (image on the left) and the second one by cross validation (image on the right). The results obtained for each dataset are as follows (All the results are calculated in the notebook called “NaiveBayes_Terratremols.ipynb”):

- Balanced Dataset without mean encoding

Class Name	precision	recall	f1-score	support
1	0.77	0.62	0.68	554
2	0.47	0.10	0.17	591
3	0.45	0.89	0.60	589

accuracy			0.54	1734
macro avg	0.56	0.54	0.49	1734
weighted avg	0.56	0.54	0.48	1734

Class Name	precision	recall	f1-score	support
1	0.79	0.61	0.69	1926
2	0.45	0.12	0.18	1926
3	0.46	0.90	0.61	1926

accuracy			0.54	5778
macro avg	0.57	0.54	0.49	5778
weighted avg	0.57	0.54	0.49	5778

In the case of balancing our data without mean encoding, we can observe that the minority class, in our case class 1, is where a considerable improvement can be found about the other predictions. The improvement is still not the optimum but it is close to being a good prediction for this minority class.

- Balanced Dataset with mean encoding

Class Name	precision	recall	f1-score	support
1	0.78	0.59	0.67	554
2	0.42	0.11	0.17	591
3	0.46	0.90	0.61	589

accuracy			0.53	1734
macro avg	0.55	0.53	0.48	1734
weighted avg	0.55	0.53	0.48	1734

Class Name	precision	recall	f1-score	support
1	0.80	0.56	0.66	1926
2	0.40	0.13	0.19	1926
3	0.46	0.91	0.61	1926

accuracy			0.53	5778
macro avg	0.55	0.53	0.49	5778
weighted avg	0.55	0.53	0.49	5778

In the case of balancing the data with mean encoding, it can be seen that there are no considerable variations. The conclusions that can be drawn are the same due to the great similarity. The accuracy has increased by only 1%.

- Preprocessed Dataset without mean encoding

Class Name	precision	recall	f1-score	support
1	0.40	0.59	0.48	567
2	0.68	0.12	0.21	3476
3	0.39	0.90	0.54	1957
accuracy			0.42	6000
macro avg	0.49	0.53	0.41	6000
weighted avg	0.56	0.42	0.34	6000

Class Name	precision	recall	f1-score	support
1	0.41	0.57	0.48	1926
2	0.67	0.12	0.21	11453
3	0.39	0.90	0.55	6621
accuracy			0.42	20000
macro avg	0.49	0.53	0.41	20000
weighted avg	0.56	0.42	0.35	20000

In the case of unbalanced data without mean encoding, the accuracy drops below 50%, being 42%. In relation to the balanced data, only class 2 improves, being this the majority class in our dataset. In spite of this, the decrease of more than 10% with respect to the prediction makes this case unfeasible.

- Preprocessed Dataset with mean encoding

Class Name	precision	recall	f1-score	support
1	0.39	0.65	0.48	567
2	0.72	0.11	0.19	3476
3	0.39	0.90	0.54	1957
accuracy			0.42	6000
macro avg	0.50	0.55	0.40	6000
weighted avg	0.58	0.42	0.33	6000

Class Name	precision	recall	f1-score	support
1	0.39	0.62	0.48	1926
2	0.71	0.11	0.19	11453
3	0.40	0.91	0.55	6621
accuracy			0.42	20000
macro avg	0.50	0.55	0.41	20000
weighted avg	0.57	0.42	0.34	20000

As in the case of the two sections of the balanced dataset, in this case we can also observe that there is hardly any difference between using mean encoding and not. The accuracy remains the same and therefore the result is not very good either.

Conclusion Naive-Bayes

In conclusion, for our dataset, the use of Naive Bayes does not obtain any good results in any aspect.

We can conclude that it makes no difference whether mean encoding is used or not for the different cases, and that for each case the use of fit or cross validation is also the same, since the results are very similar.

One of the few things that could be taken advantage of is that by balancing the dataset with undersampling, the minority classes benefit.

In the case of not balancing the dataset, we observe that class 2 is very well predicted(We predict 67%, taking into account that class 2 has almost 60% of all the elements). The problem is also that many class 2 earthquakes are predicted when they do not belong to class 2.

K-NN

We will use four different preprocessed datasets (with and without mean encoding, and balanced/unbalanced) and see how the knn algorithm performs with each of them.
We will use simple and 10-fold cross validation to determine these results.
We will also try to see if the number of data affects our results or not, and we will try to tune the parameters of the algorithm to get the best accuracy we can.

- Simple cross-validation

```
Results for the ['N', 'N', 'N']:
---- Without scaling:
     0.5865
---- With scaling:
     0.6008333333333333
Results for the ['ME', 'ME', 'N']:
---- Without scaling:
     0.6465
---- With scaling:
     0.6861666666666667
Results for the ['N_Bal', 'N', 'N_Bal']:
---- Without scaling:
     0.4411764705882353
---- With scaling:
     0.5380622837370242
Results for the ['ME_Bal', 'ME', 'N_Bal']:
---- Without scaling:
     0.6136101499423299
---- With scaling:
     0.6753171856978085
```

Here we see that we are trying each dataset with both standard scaling and without it.
We can see that scaling always gives better results, and we also find that the two datasets that use mean encoding for the **geo_level** variables give way better results. Between these two, we see that the balanced has a slightly better accuracy, but that's because most of the time it predicts the predominant label, so it's worse at predicting the minority ones.

- 10-fold cross-validation + adding minmax scaling

Now we will try every dataset as before, but using 10-fold cross-validation, and adding another type of scaling, so we will get four different results for each dataset.

```
Results for the ['N', 'N', 'N']:
---- Without scaling:
     Acc: 0.5976 - W F1: 0.5882576168905678
---- With standard scaling:
     Acc: 0.6055 - W F1: 0.5988264908051683
---- With minmax scaling:
     Acc: 0.6108 - W F1: 0.6051471612281314
---- With both scaling:
     Acc: 0.6106999999999999 - W F1: 0.6050401777753327
Results for the ['ME', 'ME', 'N']:
---- Without scaling:
     Acc: 0.6496500000000001 - W F1: 0.6401926526469546
---- With standard scaling:
     Acc: 0.69015 - W F1: 0.6847800349426938
---- With minmax scaling:
     Acc: 0.6996500000000001 - W F1: 0.6948380487618148
---- With both scaling:
     Acc: 0.6995000000000001 - W F1: 0.6946983703958566
Results for the ['N_Bal', 'N', 'N_Bal']:
---- Without scaling:
     Acc: 0.4600199096867823 - W F1: 0.45225272518789233
---- With standard scaling:
     Acc: 0.5588421197819529 - W F1: 0.5548557464054487
---- With minmax scaling:
     Acc: 0.5600630873207679 - W F1: 0.555713629837365
---- With both scaling:
     Acc: 0.5600630873207679 - W F1: 0.555713629837365
Results for the ['ME_Bal', 'ME', 'N_Bal']:
---- Without scaling:
     Acc: 0.6244364419230839 - W F1: 0.6239344004860174
---- With standard scaling:
     Acc: 0.6782627598903768 - W F1: 0.677749351494216
---- With minmax scaling:
     Acc: 0.6820725864002447 - W F1: 0.68105425339715
---- With both scaling:
     Acc: 0.6822455967808675 - W F1: 0.6812411889270361
```

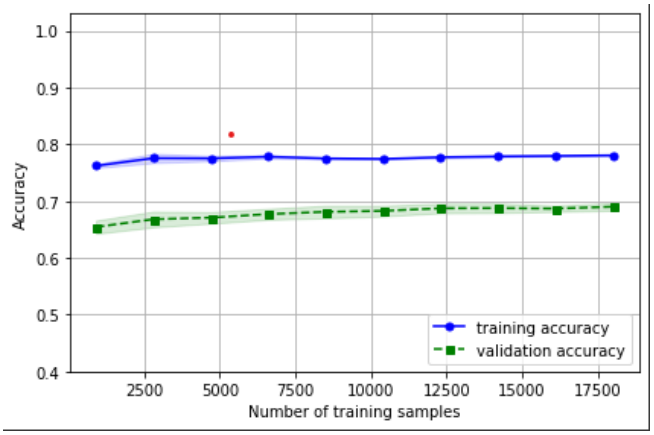
Here we can see the same as before, the mean encoded datasets give better results. What's interesting is the newly added scaling. It seems to give slightly better results than the standard scaling, and the same as using both scalers more or less.
The interesting thing here is that the unbalanced dataset gives a better f1-score, so it is a bit counterintuitive.
So from now on we will use the unbalanced dataset with minmax as it's the one that performed better.

Class Name	precision	recall	f1-score	support
1	0.51	0.43	0.47	1926
2	0.71	0.79	0.75	11453
3	0.70	0.59	0.64	6621
accuracy			0.69	20000
macro avg	0.64	0.60	0.62	20000
weighted avg	0.69	0.69	0.68	20000

These are the more detailed results that we get with this dataset, it seems to be bad at predicting elements from class 1 but fairly decent for classes 1 and 2.

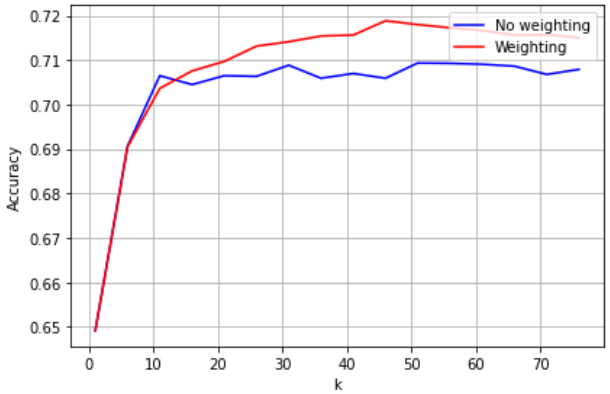
- Let's make a quick check to know if the quantity of data is relevant to the results

  We will do this because our original dataset has a lot of data, so we could potentially use more data in exchange for a slower execution time of the algorithms.



It seems that we are lucky, and our results aren't dependent on the quantity of data, so we can continue using the reduced dataset.

- Now it's time to find the best parameters for knn.
  To do this, we will plot the accuracy with the different parameters, so we can choose the best k and if it should use weighting or not.



Here we can see clearly that weighting is better than not doing it. Also, the accuracy goes up until k equals 45 more or less, then it seems to stabilize around 71.5 accuracy.
Now we will use the grid search to get the best parameters, which as we can see in the image turned out to be k = 46 and with weighting.

Finally, we will apply the model with these parameters on the test data to see if the results are more or less what we expect.

We can see that the model has improved mainly when it has to predict elements from class 1 so even if the total accuracy isn't that much better, the model should be better at predicting the real class of an element instead of predicting class 2 most of the time.

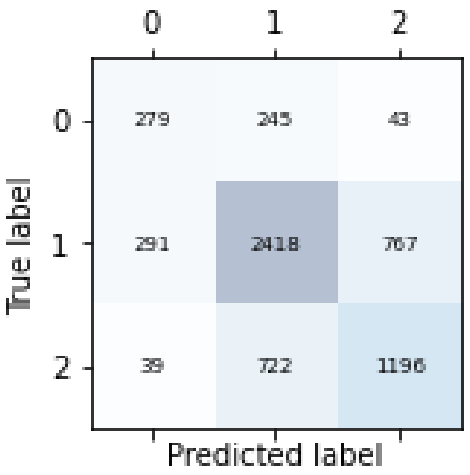
Class Name	precision	recall	f1-score	support
1	0.61	0.38	0.47	618
2	0.71	0.85	0.77	3431
3	0.75	0.57	0.65	1951
accuracy			0.71	6000
macro avg	0.69	0.60	0.63	6000
weighted avg	0.71	0.71	0.70	6000

Decision Trees

Using balanced or unBalanced dataset:

First, we have to decide if we balance our dataset or not, to do this we run a comparison between doing it balanced and not, and we choose the better performing one as the choice for the rest of the training. To do this, we train both datasets with the default DecisionTreeClassifier of sklearn. The results are:

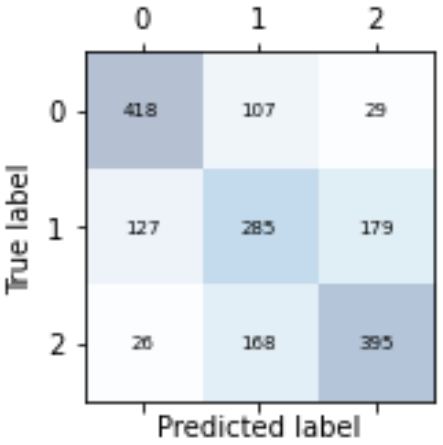
Unbalanced



Class Name	precision	recall	f1-score	support
1	0.45	0.49	0.47	567
2	0.71	0.70	0.71	3476
3	0.60	0.61	0.60	1957

accuracy			0.65	6000
macro avg	0.59	0.60	0.59	6000
weighted avg	0.65	0.65	0.65	6000

Balanced



Class Name	precision	recall	f1-score	support
1	0.72	0.75	0.74	554
2	0.52	0.49	0.50	591
3	0.66	0.67	0.67	589

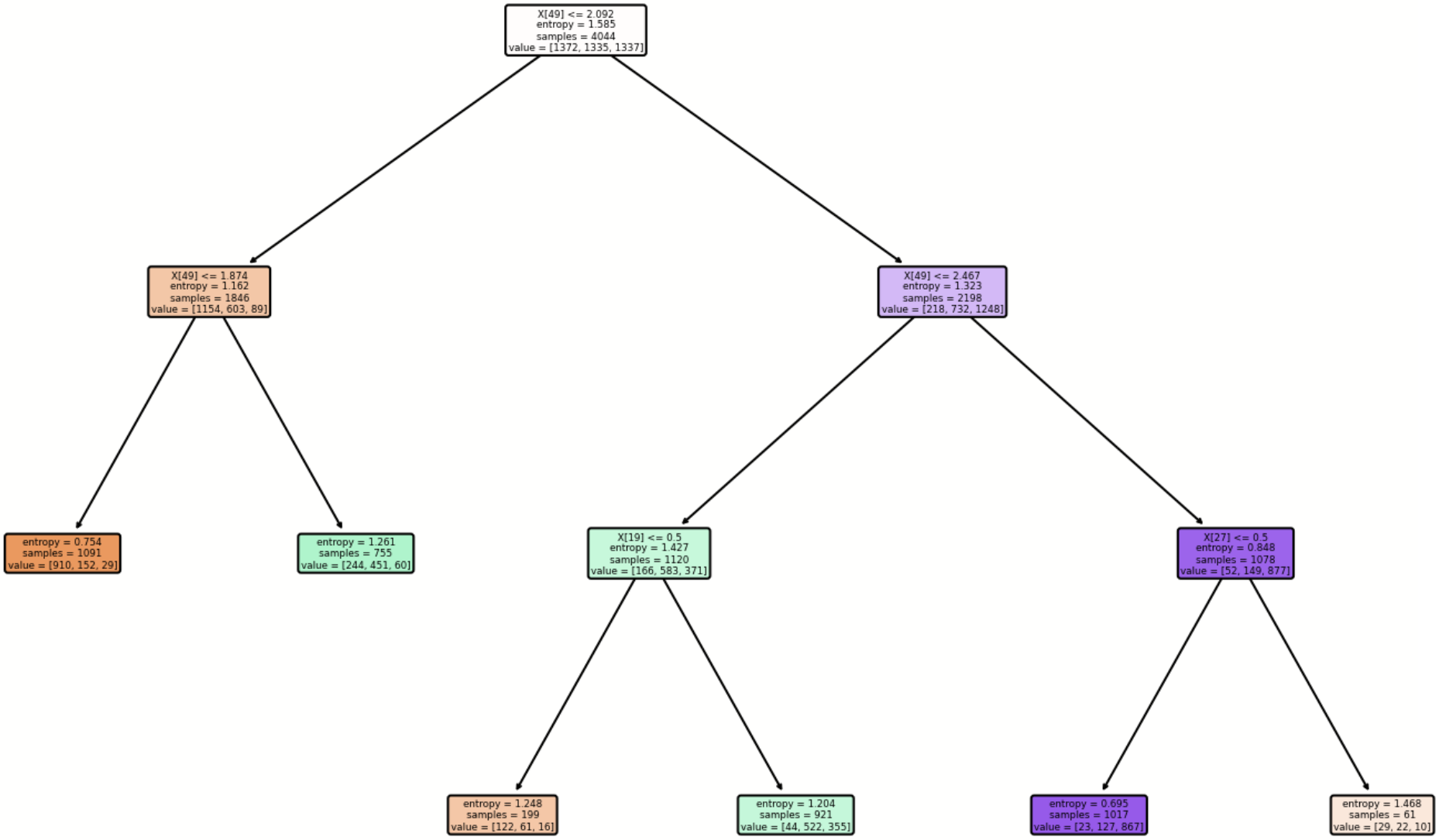
accuracy			0.64	1734
macro avg	0.63	0.64	0.64	1734
weighted avg	0.63	0.64	0.63	1734

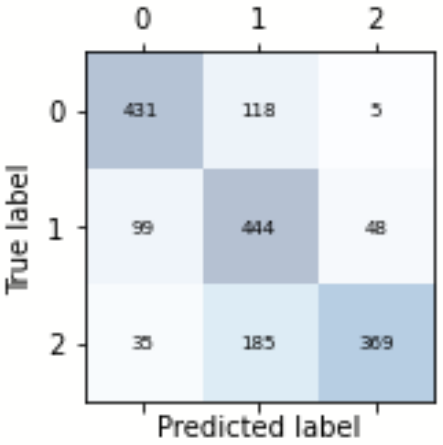
As we can observe, the overall accuracy of the two datasets are quite similar. However, in the unbalanced dataset we have a sub 50 % accuracy for class 1. For this reason we have chosen to proceed with our testing with the balanced dataset.

Representation of our Decision tree:

For the next segment, we create a representation of a decision tree with the parameters min_samples_split=2 and min_impurity_decrease=0.02.With this parameters, we obtain a more legible tree, had we chosen the default settings we would have gotten an unlegible graph with hundreds of branches and leaves.

```
DecisionTreeClassifier(criterion='entropy', min_samples_split=2,
min_impurity_decrease=0.02)
```


X[49] <= 2.092
entropy = 1.585
samples = 4044
value = [1372, 1335, 1337]
X[49] <= 1.874
entropy = 1.162
samples = 1846
value = [1154, 603, 89]
X[49] <= 2.467
entropy = 1.323
samples = 2198
value = [218, 732, 1248]
entropy = 0.754
samples = 1091
value = [910, 152, 29]
entropy = 1.261
samples = 755
value = [244, 451, 60]
X[19] <= 0.5
entropy = 1.427
samples = 1120
value = [166, 583, 371]
X[27] <= 0.5
entropy = 0.848
samples = 1078
value = [52, 149, 877]
entropy = 1.248
samples = 199
value = [122, 61, 16]
entropy = 1.204
samples = 921
value = [44, 522, 355]
entropy = 0.695
samples = 1017
value = [23, 127, 867]
entropy = 1.468
samples = 61
value = [29, 22, 10]



Class Name	precision	recall	f1-score	support
1	0.76	0.78	0.77	554
2	0.59	0.75	0.66	591
3	0.87	0.63	0.73	589

accuracy			0.72	1734
macro avg	0.74	0.72	0.72	1734
weighted avg	0.74	0.72	0.72	1734

As we can observe this tree has a much better accuracy for every class. In our next segment we will search the best parameters for a decision tree for our data.

Selecting the best parameters:

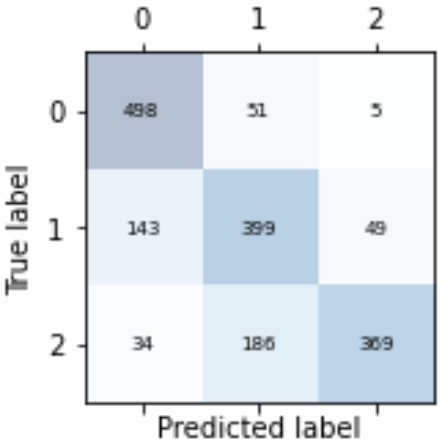
We decided to test 5 different parameters:

- Criterion: The function to measure the quality of a split. It can be either “gini” for the Gini impurity or “entropy” for the information gain.
- splitter: The strategy used to choose the split at each node. Possible strategies are “best” to choose the best split and “random” to choose one split at random.
- min_impurity_decrease: A node will be split if this split induces a decrease of the impurity greater than or equal to this value. It will be range between 0.0 and 0.5
- min_samples_split: The minimum number of samples required to be at a leaf node. It will be selected between 2 and 20.
- class_weight: The different weight of each class. The options will be; they are all balanced, a class has half importance with respect to the others, or a split of one class having weight 1, another weight 2 and the last one weight 4.

This search was performed with a Grid Search with cross validation of 10. This result in best parameters being:

- Criterion: gini
- splitter: best
- min_impurity_decrease: 0.0102
- min_samples_split: 2
- class_weight: {1: 2, 2: 2, 3: 1}

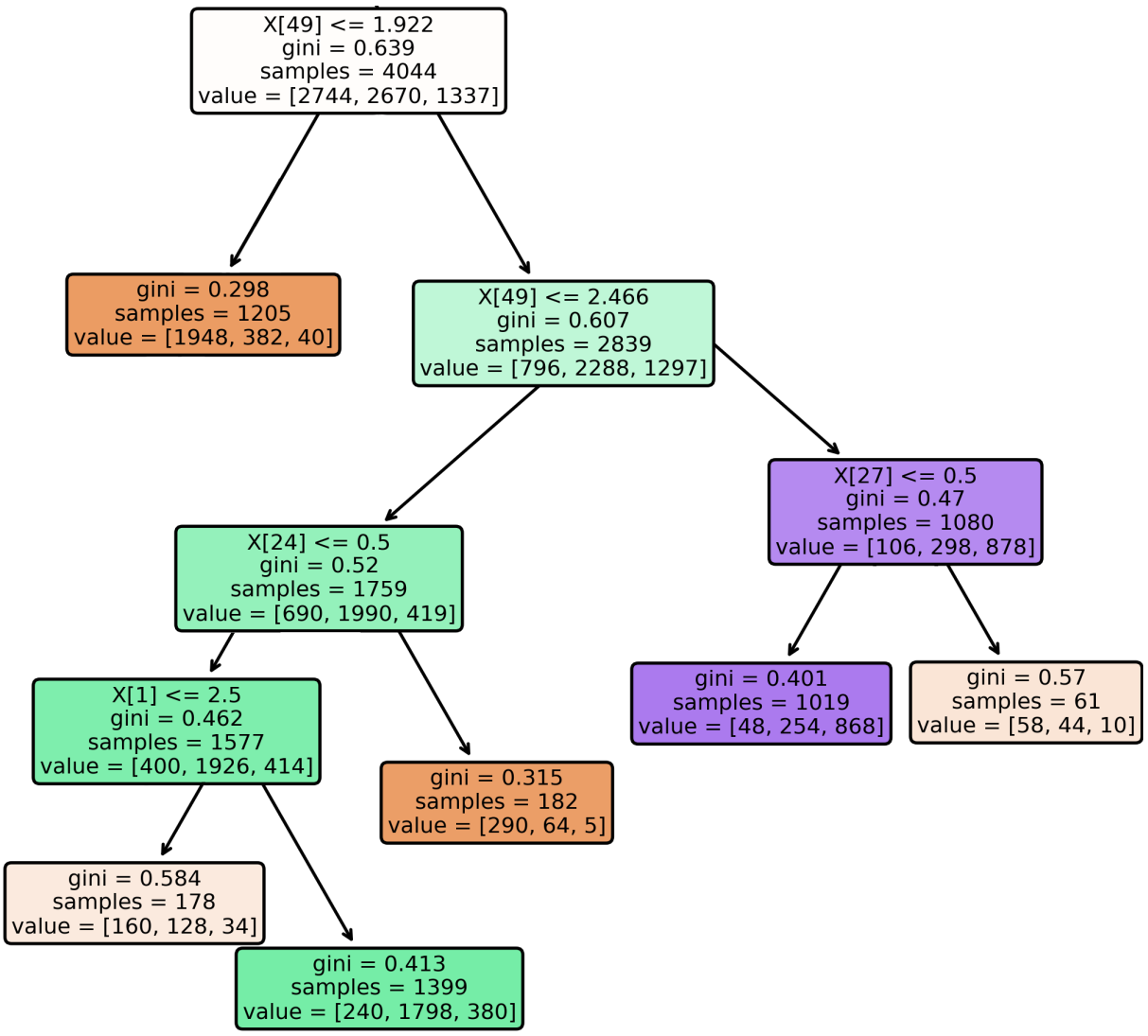
For this parameters we get the next results:



Class Name	precision	recall	f1-score	support
1	0.74	0.90	0.81	554
2	0.63	0.68	0.65	591
3	0.87	0.63	0.73	589

accuracy			0.73	1734
macro avg	0.75	0.73	0.73	1734
weighted avg	0.75	0.73	0.73	1734

The decision tree has chosen the more relevant columns, column 49 which corresponds to geo_level_3_id, column 27 ground_floor_type_v, column 24 roof_type_x and column 1 age.


X[49] <= 1.922
gini = 0.639
samples = 4044
value = [2744, 2670, 1337]
gini = 0.298
samples = 1205
value = [1948, 382, 40]
X[49] <= 2.466
gini = 0.607
samples = 2839
value = [796, 2288, 1297]
X[24] <= 0.5
gini = 0.52
samples = 1759
value = [690, 1990, 419]
X[27] <= 0.5
gini = 0.47
samples = 1080
value = [106, 298, 878]
X[1] <= 2.5
gini = 0.462
samples = 1577
value = [400, 1926, 414]
gini = 0.315
samples = 182
value = [290, 64, 5]
gini = 0.401
samples = 1019
value = [48, 254, 868]
gini = 0.57
samples = 61
value = [58, 44, 10]
gini = 0.584
samples = 178
value = [160, 128, 34]
gini = 0.413
samples = 1399
value = [240, 1798, 380]

Support Vector Machines (SVM)

In this method we executed 3 types of kernels on our dataset: Linear kernel, polynomial kernel and RBF kernel. This is the method where we had to wait the longest to finish the execution, as our dataset is very large (260,000 rows) and the algorithm has a cubic cost relative to the number of rows. So we had to reduce a lot of parameters to make it work a little faster. We reduced the dataset selecting 5000 or 10000 rows (depending on the kernel we used), we reduced the *cv* (cross validation) value down to 5 for the Grid search and we lowered the values of *C* to a reasonable number for each kernel. We decided to separate the kernels in 3 distinct notebooks so we could execute them in parallel and make all the changes we needed for each one. We tried to make multiple executions with different values trying to find a better accuracy. The execution time of each of the kernels and the grid search took around 1, 2 or even 6 hours.

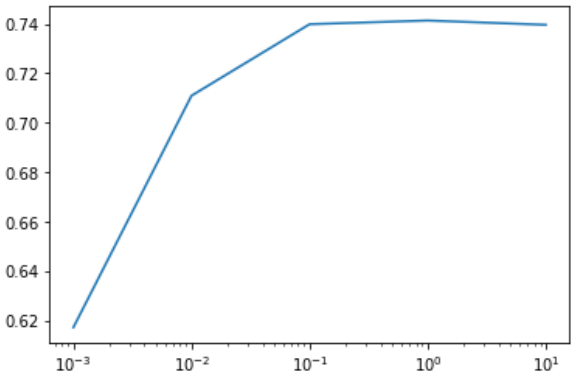
- **Linear kernel**

  As in all cases of this method we will perform the prediction with the balanced and unbalanced dataset. To perform the gridsearch of both, we have decided to use the following value `Cs = np.logspace(-3, 1, num=5, base=10.0)`.The results we obtain are quite good for our predictions. These results are as follows:

  - Balanced Dataset

Class Name	precision	recall	f1-score	support
1	0.79	0.81	0.80	554
2	0.62	0.68	0.65	591
3	0.82	0.72	0.77	589

accuracy			0.74	1734
macro avg	0.74	0.74	0.74	1734
weighted avg	0.74	0.74	0.74	1734

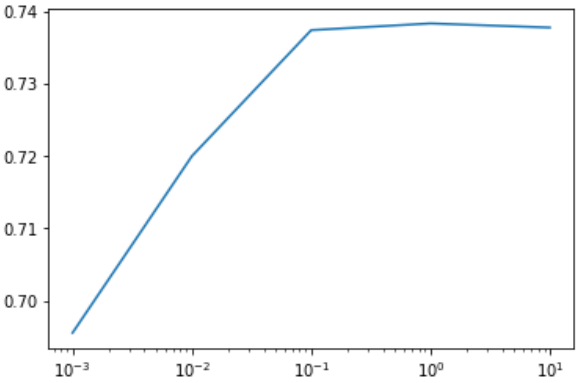


As we can see, when performing the gridsearch we obtain an accuracy of 74%, with a precision of around 80% in classes 1 and 2, so we can conclude that these are quite good results compared to other results obtained.

  - Unbalanced Dataset

Class Name	precision	recall	f1-score	support
1	0.64	0.42	0.51	567
2	0.73	0.86	0.79	3476
3	0.76	0.60	0.67	1957

accuracy			0.73	6000
macro avg	0.71	0.63	0.66	6000
weighted avg	0.73	0.73	0.73	6000



For the case of the unbalanced dataset it can be seen that there are no major differences in terms of accuracy. The majority class, in our case class 2, sees its accuracy increased, while the other two, which were previously the most accurate, are decreased by up to 15%, as is the case for class 1.

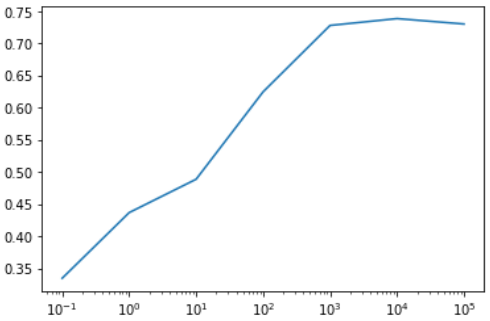
Linear Kernel Conclusion

The conclusion we came to is that we should use the balanced dataset, not only because the accuracy is a little better, and the computational time is smaller.

- **Polynomial kernel**
  In this polynomial kernel, just as in the previous kernel, we executed our code using an unbalanced and a balanced dataset. We tried both datasets using values for the grid search that were the most suitable in terms of execution time and accuracy. Both predictions, using unbalanced or balanced dataset, have been performed only with a quadratic polynomial kernel (*degree* = 2). Unfortunately, we could not get results from a cubic polynomial kernel (*degree* = 3), as it never finished the execution after more than 10 hours running, either by using an unbalanced or balanced dataset. Maybe if we had powerful tools we could have gotten these results but we had to stick to using Google Colab or Jupyter.

  - Balanced Dataset
    We used Grid search in order to find better results in terms of accuracy because they are way better than not using it to perform our prediction. In this case we used the following values of *C*: `Cs = np.logspace(-1, 5, num=7, base=10.0)` and the cross-validation value (*cv*) to 10.

Class Name	precision	recall	f1-score	support
1	0.83	0.81	0.82	578
2	0.60	0.69	0.64	578
3	0.79	0.70	0.74	578
accuracy			0.73	1734
macro avg	0.74	0.73	0.74	1734
weighted avg	0.74	0.73	0.74	1734

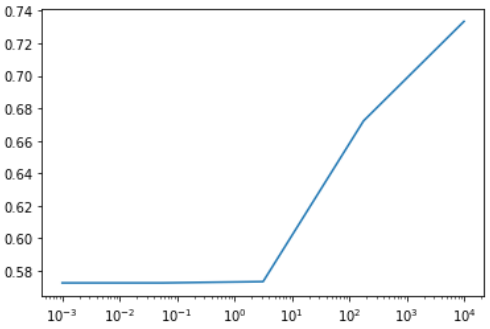


As we can see, we got similar results in comparison to the linear kernel. We got almost 74% of accuracy and the overall precision ranges between 60 to 80% depending on the class. These are way better results than not using a gridsearch

  - Unbalanced Dataset
    In this case we also used a Grid search so we could find good accuracy and precision results. Here, we used different values of *C*: `Cs = np.logspace(-3, 4, num=5, base=10.0)` because it was way faster to execute than using the previous values of *C*, where the execution could go

on for hours. We also used a cross-validation value (*cv*) of 10. Here are the results for this dataset:

Class Name	precision	recall	f1-score	support
1	0.69	0.31	0.43	578
2	0.72	0.88	0.79	3436
3	0.77	0.61	0.68	1986
accuracy			0.73	6000
macro avg	0.73	0.60	0.63	6000
weighted avg	0.73	0.73	0.72	6000



As we can see in the previous table the accuracy is very similar to the previous dataset (around 73-74%), although the graph representing the accuracy in respect to the values of *C* is quite different from the previous one using a balanced dataset. The precision, we can see ranges 69 to 77%.

Polynomial Kernel Conclusion

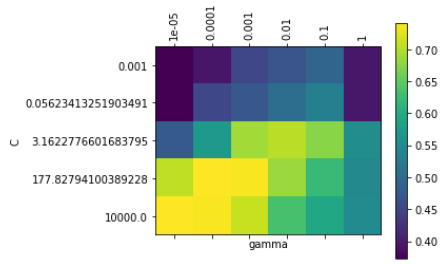
To sum up, in this kernel we saw that using a balanced dataset gives a little more accuracy than an unbalanced dataset, although it is not a big difference. One interesting thing that we noticed is that when doing the prediction using a balanced dataset, the precision of the second class lowers to around 60% while the other two classes have a precision around 80%. We have to keep in mind that the second class is the one that has the most values by far, so maybe that is why it has such a low precision.

Finally, another aspect that we would have liked to consider is seeing the results of executing the same datasets with the same values but using a cubic polynomial kernel (*degree* = 3) instead. Maybe we could have gotten better results, but if we got almost the same ones, we would have still chosen the quadratic polynomial kernel. The reason is because, as we saw during lab classes, we choose the kernel not only considering the performance, but a combination of performance and simplicity.

- **RBF kernel**

  For the RBF kernel we have decided to obtain the results with the prediction of the balanced dataset and the unbalanced dataset. Due to previous results we have come to the conclusion to perform this method simply with mean encoding as it is usually better or very similar to the results without mean encoding.

  - Balanced Dataset



Class Name	precision	recall	f1-score	support
1	0.84	0.79	0.81	578
2	0.60	0.71	0.65	578
3	0.80	0.71	0.75	578

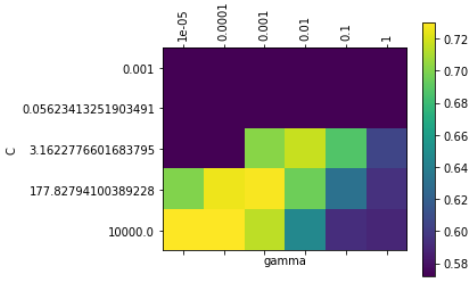
accuracy			0.73	1734
macro avg	0.75	0.73	0.74	1734
weighted avg	0.75	0.73	0.74	1734

```
Best combination of parameters found:  {'C': 177.82794100389228, 'gamma': 0.0001}

Acc. 4-fold cross on train data=  0.7413452027695351
```

In this prediction with the balanced dataset we can see how the accuracy reaches 73% as in other previous cases. Classes 1 and 3 reach a precision of over 80%, while the second class only reaches 60%, although in many of the results of this document it does not reach this figure.We can conclude that this is a good result and that it improves the polynomial kernel previously mentioned.
To perform the grid search we have used the following values Cs = np.logspace(-3, 1, num=5, base=10.0). As the image on the left shows, for a C value of 117.8 and a gamma of 0.0001 we get an accuracy of 74%.

- Unbalanced Dataset



Class Name	precision	recall	f1-score	support
1	0.66	0.37	0.47	287
2	0.73	0.87	0.79	1717
3	0.78	0.61	0.68	996

accuracy			0.74	3000
macro avg	0.72	0.62	0.65	3000
weighted avg	0.74	0.74	0.73	3000

```
Best combination of parameters found:  {'C': 10000.0, 'gamma': 1e-05}

Acc. 4-fold cross on train data=  0.7302857142857144
```

In the unbalanced, we can see that accuracy reaches 73.8% . Classes 2 and 3 in comparison to the rest of the methods, obtain good results above 73%. To perform the grid search we have used the following values Cs = np.logspace(-3, 1, num=5, base=10.0). As the image on the left shows, for a C value of 10000 and a gamma of 1e-05 we get an accuracy of 73%.

Conclusion RBF Kernel

In summary, we arrive at the conclusion that it is better to use the balanced dataset, even though the accuracy is lower, in our case infinitely lower, the recall in the case of unbalanced is much worse than in the case of the balanced dataset. In addition we can say that the execution time in the case of the unbalanced dataset is much higher to get a very little superior result.

Meta-Learning algorithms

In this part we will do 5 different Meta-Learning methods: Voting Scheme, Bagging, Random Forest, Boosting and Feature Selection with forest of Trees. All the results are calculated in the notebook called “Meta Methods.ipynb”

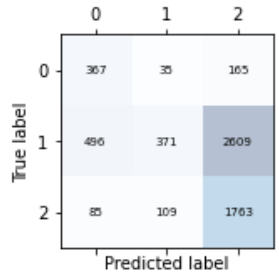
Voting Scheme

In this method we will use three different classifiers to vote for the best class for a point (Naïve Bayes, KNN and Decision Tree). We will do it twice, with no weighted voting and with a weighted voting.

First of all, we make a train for each method with each classifier and we obtain the following results:

NB

Confusion Matrix:

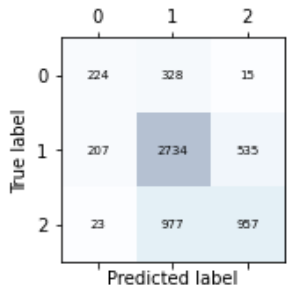


Class Name	precision	recall	f1-score	support
1	0.39	0.65	0.48	567
2	0.72	0.11	0.19	3476
3	0.39	0.90	0.54	1957

accuracy			0.42	6000
macro avg	0.50	0.55	0.40	6000
weighted avg	0.58	0.42	0.33	6000

KNN

Confusion Matrix:

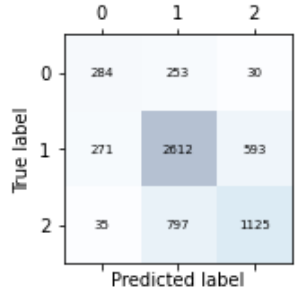


Class Name	precision	recall	f1-score	support
1	0.49	0.40	0.44	567
2	0.68	0.79	0.73	3476
3	0.64	0.49	0.55	1957

accuracy			0.65	6000
macro avg	0.60	0.56	0.57	6000
weighted avg	0.65	0.65	0.64	6000

Decision Tree

Confusion Matrix:



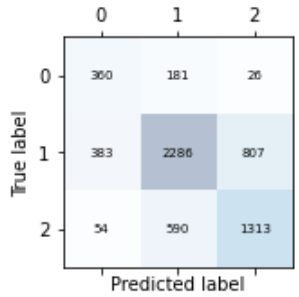
Class Name	precision	recall	f1-score	support
1	0.48	0.50	0.49	567
2	0.71	0.75	0.73	3476
3	0.64	0.57	0.61	1957

accuracy			0.67	6000
macro avg	0.61	0.61	0.61	6000
weighted avg	0.67	0.67	0.67	6000

The Decision Tree is the one we obtain better results and the Naïve Bayes is the worst classifier.

In the first voting (Majority Voting) we obtain not bad results but Naïve Bayes effects negatively in the results:

Confusion Matrix:

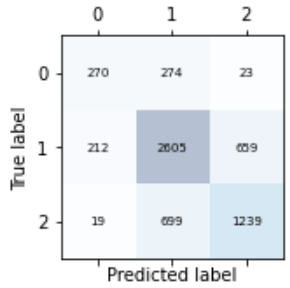


Class Name	precision	recall	f1-score	support
1	0.45	0.63	0.53	567
2	0.75	0.66	0.70	3476
3	0.61	0.67	0.64	1957

accuracy			0.66	6000
macro avg	0.60	0.65	0.62	6000
weighted avg	0.68	0.66	0.66	6000

We know that KNN and DT are better than NB so we will do a weighted voting and we will put more weight in these methods. So we obtain a significant improvement:

Confusion Matrix:



Class Name	precision	recall	f1-score	support
1	0.54	0.48	0.51	567
2	0.73	0.75	0.74	3476
3	0.64	0.63	0.64	1957

accuracy			0.69	6000
macro avg	0.64	0.62	0.63	6000
weighted avg	0.68	0.69	0.68	6000

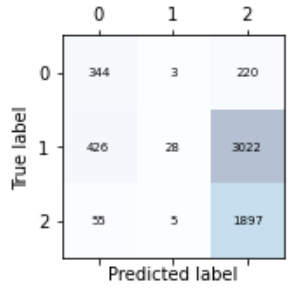
In all reports shown we can see that we predict better the predominant class

Bagging

The next step is to apply Bagging. We will do different baggins for each classifier (GNB, KNN, and DT) with different estimators. We are going to fix the parameter max_features to 0.35 and n_estimators in 200 in the two first methods and in 100 in Decision Tree.

GNB

Confusion Matrix:

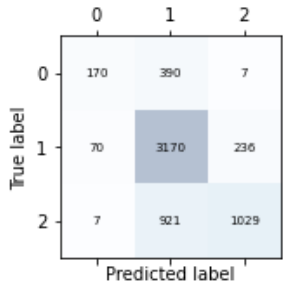


Class Name	precision	recall	f1-score	support
1	0.42	0.61	0.49	567
2	0.78	0.01	0.02	3476
3	0.37	0.97	0.53	1957

accuracy			0.38	6000
macro avg	0.52	0.53	0.35	6000
weighted avg	0.61	0.38	0.23	6000

KNN

Confusion Matrix:

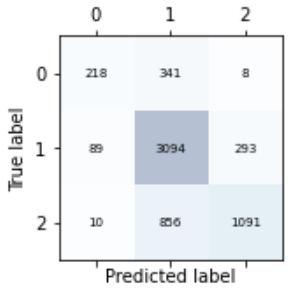


Class Name	precision	recall	f1-score	support
1	0.69	0.30	0.42	567
2	0.71	0.91	0.80	3476
3	0.81	0.53	0.64	1957

accuracy			0.73	6000
macro avg	0.73	0.58	0.62	6000
weighted avg	0.74	0.73	0.71	6000

DT

Confusion Matrix:



Class Name	precision	recall	f1-score	support
1	0.69	0.38	0.49	567
2	0.72	0.89	0.80	3476
3	0.78	0.56	0.65	1957

accuracy			0.73	6000
macro avg	0.73	0.61	0.65	6000
weighted avg	0.74	0.73	0.72	6000

The results of GNB are bad (we could know it by observing previous results). The results between the KNN and the DT are quite similar so we can determine that the bagging KNN and DT are so good, with an accuracy of 73%.

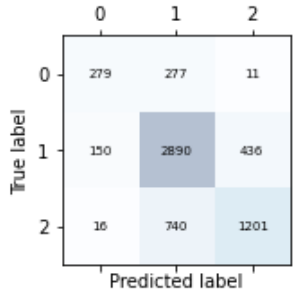
Also we observed that the classifier predicts better the majority class.

Random Forest

The next method is the Random Forest. Random Forest fits a number of decision tree classifiers (n_estimators) and the output is the average. We decided to train two random forest, fixing and no fixing the paràmeter of max_features and modifying the n_estimators

n_estimators = 100

Confusion Matrix:

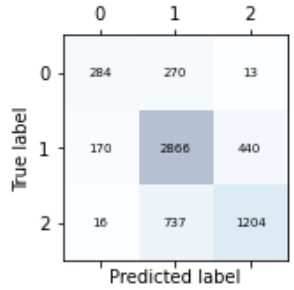


Class Name	precision	recall	f1-score	support
1	0.63	0.49	0.55	567
2	0.74	0.83	0.78	3476
3	0.73	0.61	0.67	1957

accuracy			0.73	6000
macro avg	0.70	0.65	0.67	6000
weighted avg	0.73	0.73	0.72	6000

n_estimators = 100 and max_features = 0.35

Confusion Matrix:



Class Name	precision	recall	f1-score	support
1	0.60	0.50	0.55	567
2	0.74	0.82	0.78	3476
3	0.73	0.62	0.67	1957

accuracy			0.73	6000
macro avg	0.69	0.65	0.66	6000
weighted avg	0.72	0.73	0.72	6000

We see a not bad accuracy and we see that the predominant class doesn't define the whole classificator. Random Forest fits the data well and the results are similar to the results of the

bagging with decision trees and knn. But also we have to consider that the classifier predicts better the majority class than others.

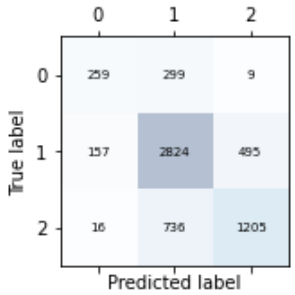
We also have made a test for an ExtraTreeClassifier but we obtained worse results , so we decided to not show the results.

Boosting

We will use two methods of boosting: AdaBoostClassifier and GradientBoostingClassifier.

AdaBoostClassifier

Confusion Matrix:

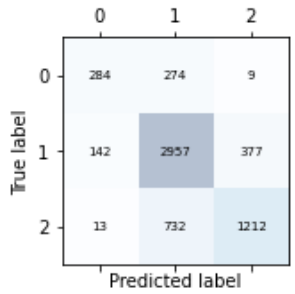


Class Name	precision	recall	f1-score	support
1	0.60	0.46	0.52	567
2	0.73	0.81	0.77	3476
3	0.71	0.62	0.66	1957

accuracy			0.71	6000
macro avg	0.68	0.63	0.65	6000
weighted avg	0.71	0.71	0.71	6000

Gradient Boosting Classifier

Confusion Matrix:



Class Name	precision	recall	f1-score	support
1	0.65	0.50	0.56	567
2	0.75	0.85	0.79	3476
3	0.76	0.62	0.68	1957

accuracy			0.74	6000
macro avg	0.72	0.66	0.68	6000
weighted avg	0.74	0.74	0.74	6000

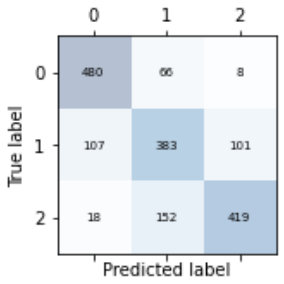
WIth the Gradient Boosting Classifier we achieve the better results we had seen but also we had a high precision in the majority class and no soo good in the others (but also not bad).

Conclusions Meta-Learning Algorithms

We achieve good results. The better results are obtained with the boosting algorithm. Is the last result we obtained (74%). These results are quite similar to bagging with a decision tree and random forest.

In the notebook we do all previous calculations with the balanced data. The global accuracy is quite similar. The results with the gradient boosting classifier with balanced data are the following:

Confusion Matrix:



Class Name	precision	recall	f1-score	support
1	0.79	0.87	0.83	554
2	0.64	0.65	0.64	591
3	0.79	0.71	0.75	589
accuracy			0.74	1734
macro avg	0.74	0.74	0.74	1734
weighted avg	0.74	0.74	0.74	1734

The global accuracy is quite similar, 74%. But if we observe the classification report and the confusion matrix we can see that this classifier doesn't classify the class ‘2’ better than the minority classes, as we can see in the unbalanced data set. If we would have a higher precision for the majority class we would use a classifier trained with the original unbalanced dataset, but if we would have better results classifying classes one and three we would use the last classifier. This one maybe is better because there are not a lot of differences between the precision and recall, the results are balanced.

Comparison and conclusions

The preprocessing step has been very important to reduce the size of the data set, because originally we had 260.000 (two hundred sixty thousand) rows with 39 columns. Also we had the problem that the data was unbalanced, there was a predominant class (‘2’). We decided to work with two data sets, an unbalanced dataset reduced to 20.000 (twenty thousand) rows and a balanced dataset with 8.779 (eight thousand seven hundred seventy-nine) rows.

In the preprocessing also we have done a transformation of some values that have notably increased the results of many of the methods applied, for example, the mean encoding in the variable that were about the geographical ids. Also we removed a lot of variables that didn’t provide relevant information.

Of all methods applied, we can decide that Naïve Bayes didn’t fit our data well, we can’t improve the results with an accuracy over 55%.

For the other methods, we are quite happy with the results achieved. At first, we obtained results between 50 and 60%, with a clear overrepresentation of the majority class. But, with the steps of the preprocessing and finding better parameters for each model, we cracked the 70% barrier for several of them.

One of the models that have yielded better results is the RBF kernel with the unbalanced dataset and the Gradient Boosting Classifier each with 74% accuracy.

For this project, we have once again discovered the importance of the preprocessing step. The large improvements that we have accomplished in the accuracy of the models has been, due to iterating in our preprocessing so as to better fit our models.

Finally, we obtain quite similar accuracy choosing either the balanced or the unbalanced dataset. However, in the unbalanced, the classifier predicts the majority class better than the balanced one. We cannot decide if it's more important to predict one class with respect to another.