POLITECNICO
MILANO 1863

Data-Science in C.E. Report

Machine-learning models and architectures for predicting forest cover type.

Data Science in Chemical Engineering, A.Y. 2023-2024.

Andrea Somma: 10660082

Lorenzo Paggetta: 10618672

Pietro Marelli: 10680881

Supervisors:

Prof. Alessandro Stagni

Ing. Riccardo Caraccio

Contents

1 Introduction

The present work showcases different methods to develop a classifier for the Cover Type dataset, in order to achieve an accurate and balanced model for the cover forest type from the cartographic variables in the dataset.
The Cover Type dataset contains trees observation from four wilderness areas of the Roosevelt National forest in Colorado. The data is made of cartographic variables only, with no remotely sensed data. It is a rather large dataset, made of 7 forest cover types (Table 1), more than half a million instances and 54 features (Table 2), which include data such as elevation, aspect, slope, distance to hydrology, soil type and many others.

#	Forest cover type
1	Spruce/Fir
2	Lodgepole Pine
3	Ponderosa Pine
4	Cottonwood/Willow
5	Aspen
6	Douglas-fir
7	Krummholz

Table 1: Forest cover types in the covtype dataset

#	Label	Type
1	Elevation	Integer
2	Aspect	Integer
3	Slope	Integer
4	Horizontal Distance To Hydrology	Integer
5	Vertical Distance To Hydrology	Integer
6	Horizontal Distance To Roadways	Integer
7	Hillshade 9am	Integer
8	Hillshade Noon	Integer
9	Hillshade 3pm	Integer
10	Horizontal Distance To Fire Points	Integer
11-14	Wilderness Area	Binary
15-54	Soil Type	Binary

Table 2: Feature labels in the covtype dataset

The aim of the present work was to perform a classification of the data from the Cover Type Dataset using a machine learning (ML) and neural network (NN) approach, discussing also the differences between the ML-based and NN-based models. Moreover, in the latter section of the present work, a further improvement from the obtained results will be proposed (optimized ML and NN models), obtaining much lighter models with just a relatively small loss on accuracy.

Since this dataset is fairly large, preliminary computations were performed on only a small fraction of the total data, to save time. After having correctly assessed the proper classification model, the training was repeated including all the data and the final results were obtained. Beyond being large,

the Cover Type Dataset is characterized by the fact that is unbalanced: as shown in Figure 1, the first three cover type being much more prevalent than the other categories. This feature represent one of the main challenges in the creation of a classifier and will play a huge role.

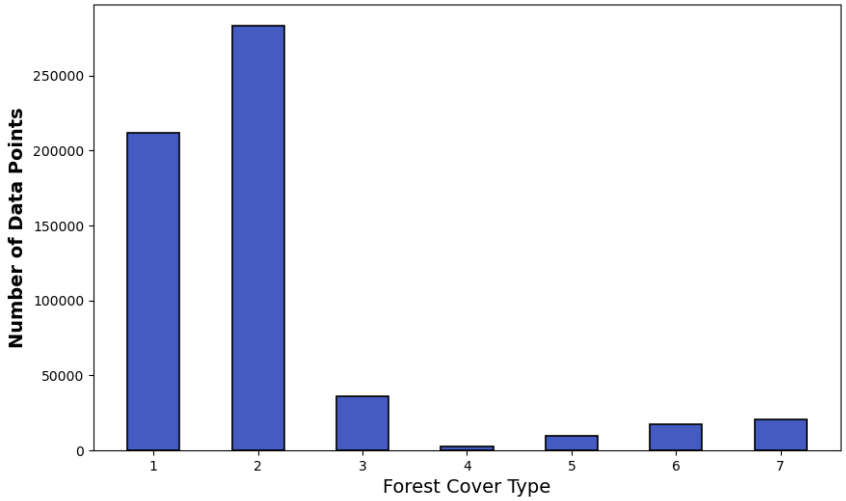

Figure 1: Number of data points for each cover type.

Another important characteristic of the Cover Type Dataset is the fact that the labels regarding the Wilderness area and Soil Type (Features 11-54) are represented as binary indicator (Table 2). This feature will pave the way for the proper development of a classifier, especially in the case of the NN-based model.

2 Machine Learning Based Model

2.1 Introduction on Machine Learning Methods

The first approach that was applied was machine learning, or ML for short. The reason for this was to develop a model for accurately predicting which class, meaning the cover forest type, that data belongs to. In fact, ML is an ideal method for data characterization, as it is based on pattern recognition and trained on already available data.

Since the data was already labelled, the best approach is to be found in supervised ML, thus avoiding unnecessary steps with clustering. In order to obtain the optimal model, both in terms of accuracy and efficiency, five different classification methods were explored, briefly described as follows:

- **Decision Tree Classifier**: relatively simple and easy to implement. It is one of the most common classifier as it operates with a hierarchical methodology, generally by binary split. This is because the model is built by "cutting" data in simple sectors so that the classification is carried out by successive if/else decisions.
- **Random Forest Classifier**: it lays on the foundations of the previous method. The training step is basically the same, but rather than one, it creates multiple decision trees on different randomly selected data. The data is then processed through all trees and the final result is the majority vote, thus adding robustness by reducing statistical variability.
- **Extra Trees Classifier**: it works almost identically to the random forest classifier. The main difference is that the previous method firstly splits the dataset in smaller subsets and then trains each model on its own subset, while this method generates multiple trees by using the entire dataset, further reducing the noise of the data.
- **AdaBoost Classifier**: it improves performance on "simpler" methods (also called weak learners), by operating them successively, each time augmenting the weight of wrongfully classified data for an improved model. Finally, it combines the output of the weak learners into a weighted sum for a boosted model.
- **Bagging Classifier**: it combines previous methods similarly to how random forest operates. It splits the dataset in subsets (Bootstrapping), then applies various classification methods to each of them, and finally combines them in an ensemble model (Bagging). The resulting output is the majority vote of all subsets.

2.2 The selection of the right classifier

A preliminary evaluation was performed on each model. An important detail to point out is that the fraction of the dataset used for training is comparatively small, being only 20%. The reason

for this is having the code running quicker due to the relatively high amount of data, especially because this step was made just for selecting the optimal classifier, not refining the final model. In addition, the data was rescaled using the *StandardScaler* utility class from the *sklearn* package, with particular care in keeping the testing data as independent from the training data as possible.

The models were then properly defined, trained, and tested for accuracy based on how well they predicted the correct allocation of the remaining data points. This procedure was then iterated to reduce statistical variability and the obtained results finally plotted in the following bar graph (Figure 2):

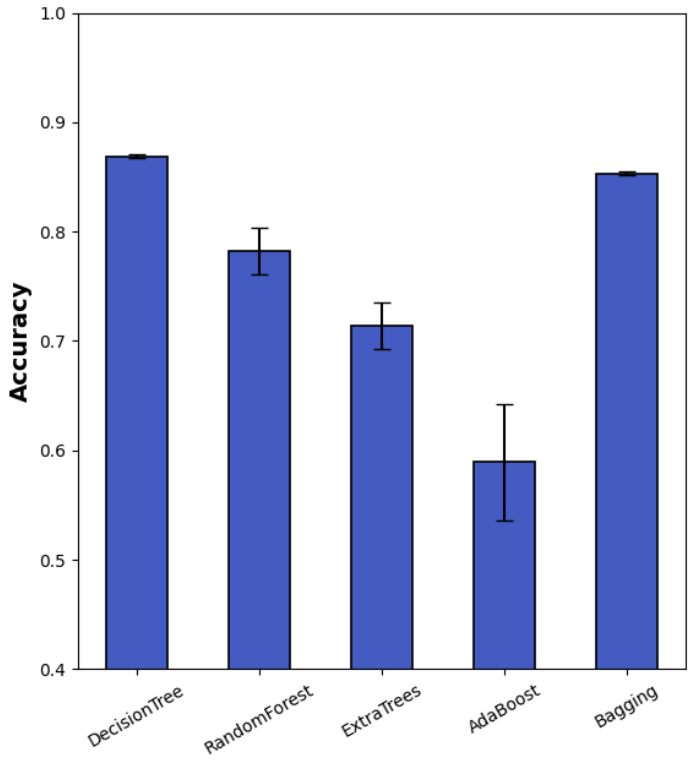

Figure 2: Accuracy plot with standard deviation depending on the type of classifier.

In order to identify the best classifier from which to obtain a model, the optimization was performed only on the two methods that achieved the best accuracy: the *Decision Tree* and the *Bagging* classifier. Moreover, so as to acquire more information on how the model operates, the order of feature importance was also plotted:

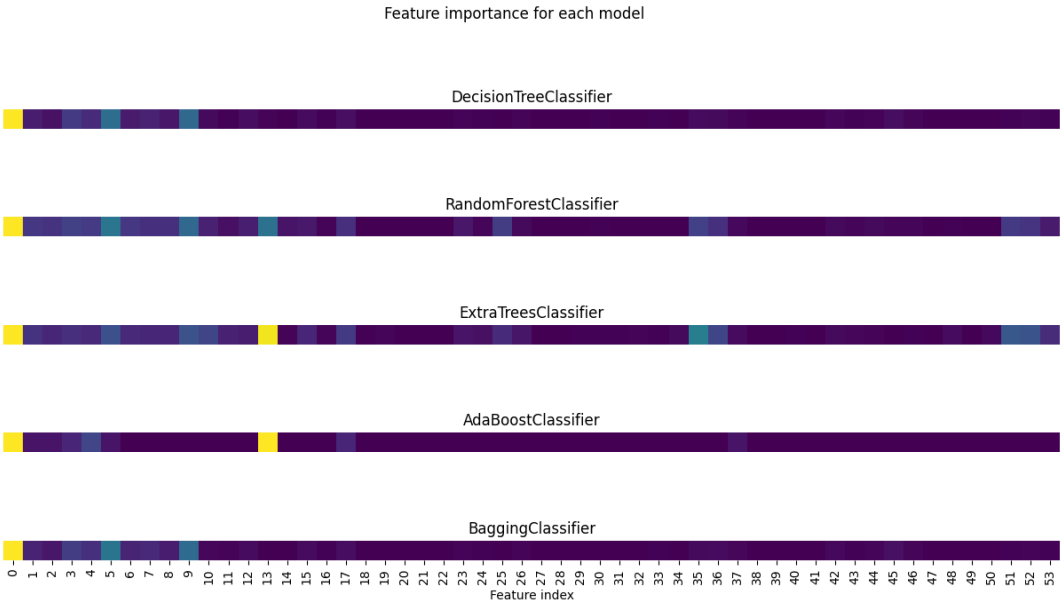

Figure 3: Order of feature importance for each classifier. The brighter the color, the more relevant the feature is for the model.

Figure 3 provides valuable information on which parameters influence the classification of cover types the most. The data in question does not have particularly high dimensionality, having 54 features for each data point, but it still could be useful to operate data mining to extract the most relevant features in order to make the model more interpretable, more efficient and avoid overfitting.

In particular, it is possible to observe how the first feature, representing the altitude at which a given forest cover is located, is also the most relevant for classification in basically all the models, as depicted in Figure 3 by the brightest color. It is also interesting to point out that the two models with the highest accuracy, the *Decision Tree* and the *Bagging* ones, also share many similarities in which are the most relevant features, the 6^{th} and 10^{th} in particular, both providing information about geographical disposition. On the other hand, the two models with the lowest precision, the *Extra Trees* and *AdaBoost* respectively, both put particular importance on the 14^{th} parameter, index of the type of wilderness area, which is of negligible importance for the other classifiers. The *Extra Trees* model in particular shows an increased importance in the parameters above the 14^{th}, all representing the soil type as binary information, while all the other models basically discard them. This could signify an overfitting of the training data on the model, which is why the accuracy drops significantly during the testing. Finally, the order of feature importance for the *Random Forest* appears to be kind of a middle ground between the *Decision Tree* and the *Extra Tree*, which also could explain its intermediate accuracy compared to the other models.

The final step, before proceeding with optimization, was to compare how modifying the number of estimators and the maximum depth of the *Random Forest* classifier could affect the accuracy of the model, this particular model is introduced as a test case given the similarity of the different ML classifiers. This was done to achieve some sensibility on how the number of parameters influence

the resulting model. The optimal solution would necessarily be a tradeoff of the highest accuracy with the lowest complexity of the model. In order to do so, the model was trained and tested iteratively, each time incrementing the number of estimators and the maximum depth to observe when the accuracy would reach a plateau; at that point, any added complexity would not result in a better model, instead it would be an indicator of overfitting, as it is possible to deduce from Figure 4.

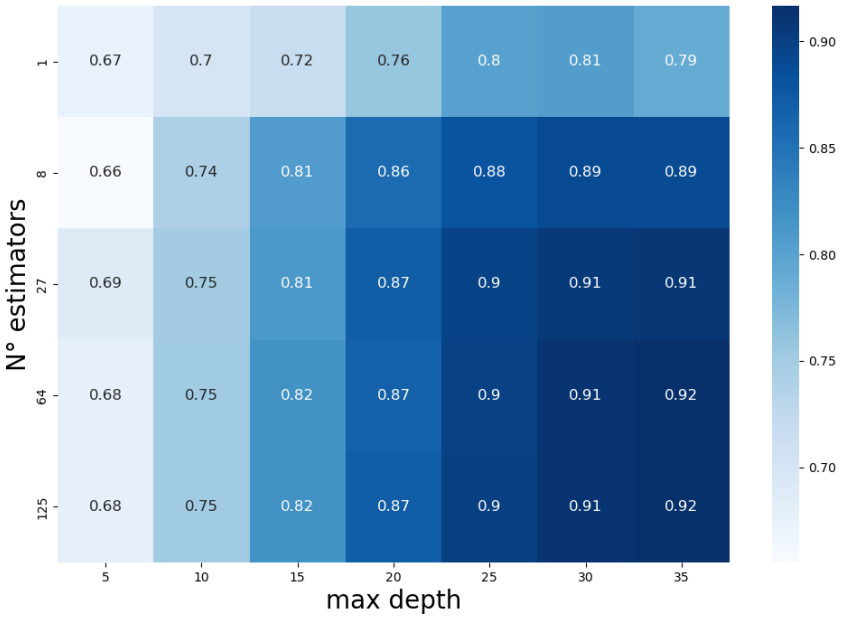

Figure 4: Accuracy matrix expressed as a function of the tuning parameters

2.3 Rescaling and Rebalancing techniques

Once the two classifiers with better performance were established, the next step was optimization. Before defining the models, some data manipulation was explored as a possibility in order to increase accuracy, specifically rescaling and rebalancing. The first one was applied in the same way as during the selection phase, to normalize the dataset and therefore making all the parameters comparable for the training. The latter instead was considered because, as it is possible to determine from Figure 1, the classes of the dataset are imbalanced in favor of the first two. Because of this, the model is relatively less efficient in classifying data belonging to the less dense classes, which risk of being less useful overall.

For the rebalancing, it was used the *Imblearn* library, which offers many possible options for handling imbalanced datasets. In particular four different samplers were tested, as explained in the following paragraph:

- **SMOTE**: This method employs oversampling, a technique that tries to rebalance the underrepresented classes by generating artificial points belonging to that class, in this case, specif-

ically by interpolating them from already existing data points. However, it could generate a model that falsely represents actual data by mislabeling them in the case of classes that do not form clearly defined clusters. A figure showing how this technique works is shown in Figure 5.

- **Random Under Sampler or RUS**: This technique operates in an opposite way from the previous one. It is an example of undersampling, meaning that the number of data points belonging to the other classes is reduced in favor to the minority one. As its name implies, it is operated by randomly discarding data for a smaller sample for training, but in turn, it may lead to "incomplete" models, unable to properly categorize the data of the higher classes.
- **SMOTEENN or SNN**: It is a combined method, meaning that it utilizes both over and undersampling. This one in particular aims at reducing the noisy data generated from the application of SMOTE by adding it with the Edited Nearest Neighbor technique, which removes the data points which are the closest to the decision boundaries.
- **SMOTETomek or SM**: Similarly to the previous one, it is a combined method. It attempts at clearing the noisy artificial data by removing Tomek links, which are defined as two data points from different classes that are their respective closest neighbors. The sampling strategy can be implemented by either removing the data point belonging to the majority class or both.

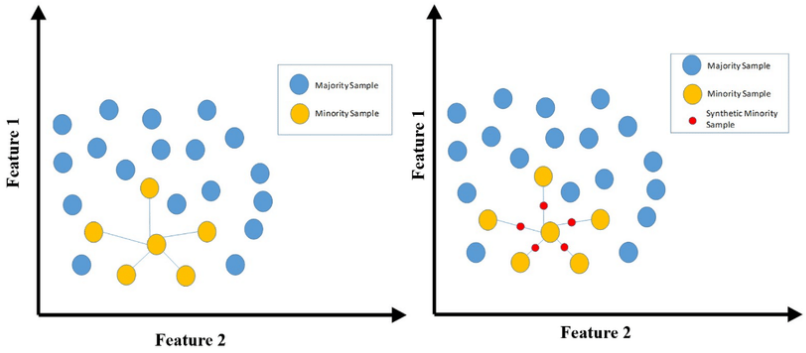

Figure 5: SMOTE technique

In order to make an informed choice of which method to implement, for each option the code was run multiple times to reduce statistical variability. The tuning parameters chosen for the different classifiers were setting the number of estimators to 40 for the *Bagging* and the maximum depth to 30 for the *Decision Tree*. For the latter in particular, some adjustment were made in order to obtain better results: first off, the minimum impurity decrease was set to 10^{-5}, meaning that a node will split only if that can decrease the likelihood of wrongful classification as calculated by the Gini Index (which regards the entropy of the system). In addition, the minimum sample split was set to 2, which, as the name implies, impose a minimum in the number of samples required to split a node. Similarly, the minimum leaf sample was set to 1, meaning that, for a split to operate, it requires at least one data point during training to be classified in both directions.

After defining the models as such, each one was trained and tested for predicting data. The functions *accuracy_score* and *classification_report* were utilized to extrapolate data on the performances of each model.

2.4 Model evaluation

Since the classes are heavily imbalanced, relying only on the accuracy alone for the evaluation of the model is reductive, as it could just signify that the majority classes absorb most of the wrongly classified data. For that reason, the Macro averaged precision and recall were also plotted. The distinction between these indicators is that the accuracy is defined as the total number of correct predictions of the model over the total number of data points, while the other two are a little more complex. Precision and recall are class metrics which can be averaged over all the classes to make them more interpretable. Precision is defined as the number of samples correctly labelled with a given class over all the samples that were identified with that class. Meanwhile, the recall measures how many of the data points in a given class were correctly identified, meaning that it is defined as the number of samples correctly labelled with a given class over all the samples that truly belonged to that class. This distinction is exemplified in following formulas:

$$Precision = \frac{TP}{TP + FP} \tag{1}$$

$$Recall = \frac{TP}{TP + FN} \tag{2}$$

Where TP stands for true positives, FP false positives and FN false negatives.
In the following picture, a visual scheme for a better understanding of the difference between accuracy, precision and recall is provided.

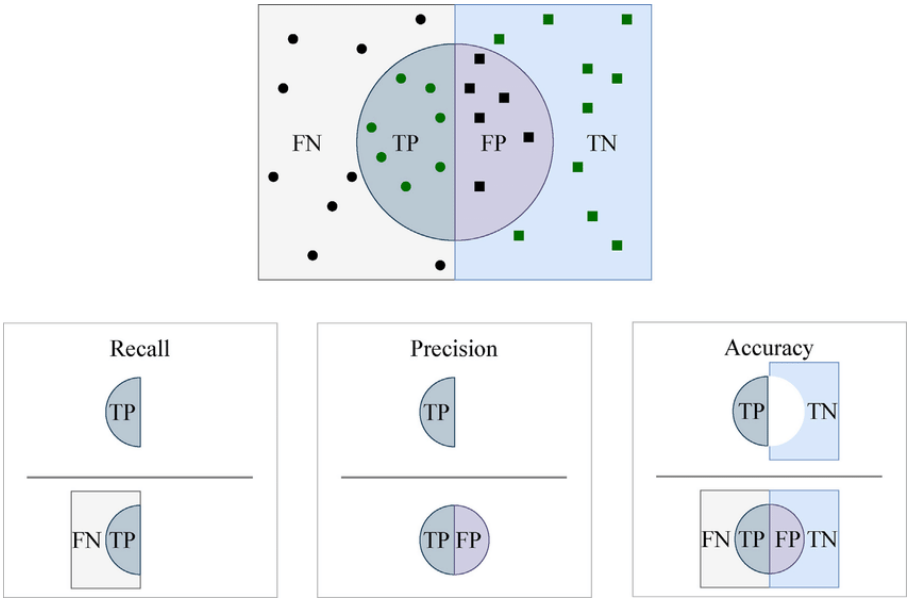

Figure 6: Performance metrics visualized

Lastly, the time required for training was also taken into account for considering the complexity of the model. It is important to note that the time required for executing any program can vary wildly depending on hardware, which is why all simulation were carried out by the same machine with an Intel i7-12700H processor and 16 GB of RAM as benchmark. As a simplifying consideration it is more useful to consider the times in relative terms. The final outcomes were plotted in the following figures for an informed discussion on the results.

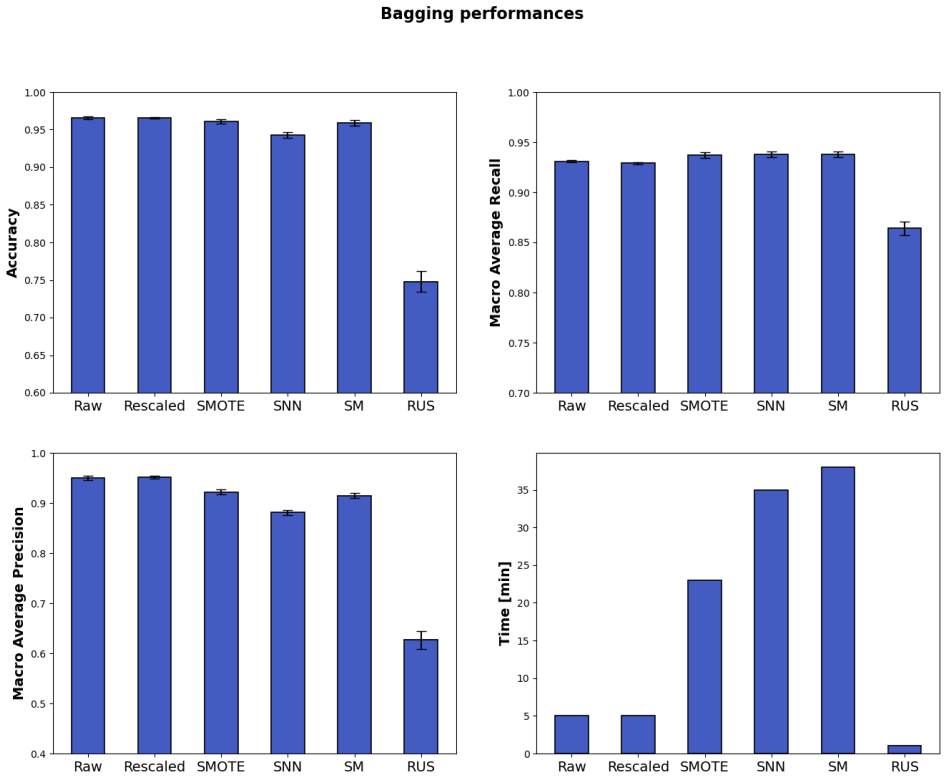

Figure 7: Bagging-based model performance summary

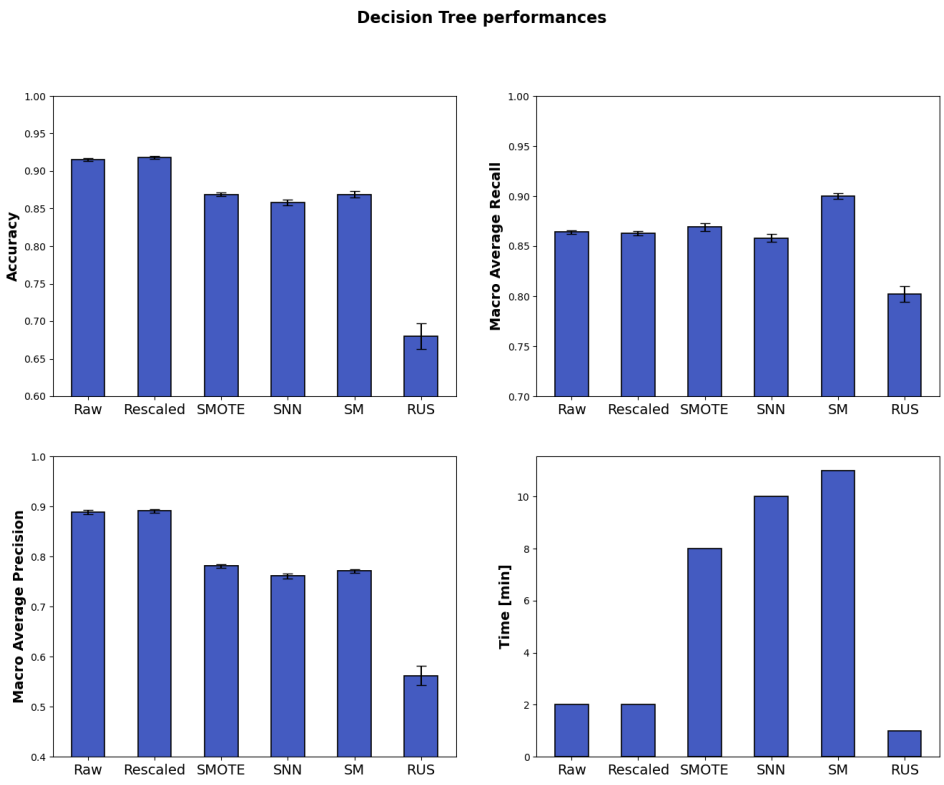

Figure 8: Decision Tree-based model performance summary

In this way, it is possible to visualize the results for each classifiers in all their trials. As it results from Figures 7 and 8, in both cases the rescaling triggers an ever so slightly increase in accuracy, without visibly incrementing the time required for executing the program. For this reason, the rescaling can be consider a viable option for the preprocessing of data.

However the same does not hold true for any of the rebalancing methods. Regarding the *Bagging* classifier (Figure 7), the oversampling operated by the SMOTE and the other two combined samplers does increase the Macro averaged recall compared to the raw data, but at the cost of lowering the precision and therefore obtaining a model with lower accuracy. In addition, when looking at the time required for training the model, this is five times longer with the SMOTE sampler and more than seven times longer when using either of the two combined methods. In contrast, the RUS sampler leads to a much quicker training, even significantly faster than raw data, however it pays it in greatly reducing both precision and recall for all the classes involved, thus heavily decreasing accuracy and rendering the final model almost useless.

As for the *Decision Tree* classifier, the profiles show a similarity in how the various techniques affect the indicators compared to the previous case, albeit the differences between one another are much more pronounced. As shown in Figure 8, the application of rebalancing in any trial greatly reduced the precision, for an almost negligible gain in recall, thus affecting the overall accuracy of the model. Finally, just as before, the undersampling was quicker in training compared to any other case, but because of this it results in a ultimately fallacious model.

As a final observation, it is possible to state that the *Bagging* classifier resulted in overall better models compared to the *Decision Tree* classifier. Their peaks in accuracy were both reached after the data was rescaled and they were about 97% and 92% respectively. The significant increase in accuracy for the *Bagging* is mainly due to the fact that its model is much more complex compared to the *Decision Tree* one, however this result is not without its drawbacks: training time was more than twice the one required for the "simpler" model and their difference in complexity can be seen by the fact that the dimensions of the saved model for the *Bagging* are about one order of magnitude heavier (24 MB compared to 3 MB), effectively meaning that it requires a much higher number of parameters in order to accurately predict the results.

3 Optimized Machine Learning Based Model

3.1 Improvements on the previous model

Even though a viable solution was already established, as a possible improvement the possibility of having a smaller model was explored, specifically one based on the *Decision Tree* classifier. In accordance with what was obtained from the previous step, the model was developed by firstly scaling the data and then fitting them in the classifier, as the rebalancing step would only lead to a heavier and less accurate model.

The tuning parameters were changed in order to get a lighter model, with the understanding that a more lax model would inevitably cause a loss in accuracy. First off, the minimum impurity decrease was set to $2.15 \cdot 10^{-5}$, meaning that a given node would split only for a more significant decrease in the entropy of the system, compared to the previous case.

In addition, the minimum sample split and the minimum leaf sample were both set to 10, further reducing the complexity of the system. The reason for this, is that now the minimum number of data points required for a node split has increased, thus avoiding excessive branching on the decision tree while further reducing the risk of overfitting and redundant parameters.

However, in order to obtain a sufficient accuracy, the maximum depth tolerable was changed to an arbitrarily high number. In this case 75 was selected, even though in practice each model created like this just barely overcomes 30 nodes split before stopping the training. In this way, it was possible to achieve a model with 90% accuracy while greatly reducing its complexity to a measly 723 kB, almost two orders of magnitude lighter compared to the *Bagging*-based model of the previous step.

3.2 Results

The results obtained from the optimized model can be better visualized in the confusion matrix in Figure 9 to better understand how it operates.

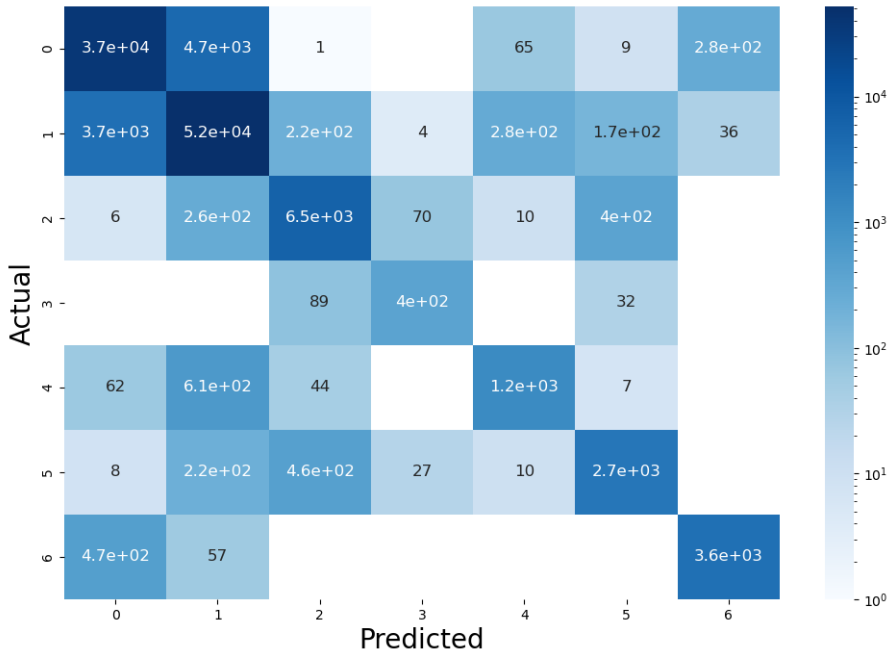

Figure 9: Confusion matrix for the ML-based model

By comparing the most relevant features for the various models which were developed, the improvements made result much more evident as it is shown in Table 3.

Model	Accuracy [%]	Parameters	Size [MB]	Training time [min]
Bagging-based	97	3.9M	24	5
DecisionTree-based	92	6k	3	2
DecisionTree-based opt	90	3k	0.72	∼20s

Table 3: ML-based models feature comparison

An important distinction to make is that for *Decision Tree* classifiers the number of parameters memorized is identified as the number of leaves of the final model, while for the *Bagging* ones, the number of parameters is defined as the number of nodes times two added to the number of leaves as in accordance with *scikit-learn* documentation

In conclusion, the development of light models shows the potential of having a better scope of applications in a world where processing power and memory usage are becoming the limiting factors.

4 Neural network based model

4.1 Introduction on Neural Networks

One of the classifying methods used in the present work consisted in the developing a neural network-based model (NN-based model) that performed the classification of the data in the Cover Type Dataset. A neural network consists of several units (perceptrons or neurons) that process information through connected nodes; they are able to perform parallel processing and pattern recognition, which is the aspect that this work is interested in. An example of a NN structure of is shown in Figure 10: it is made of an input layer, hidden layers and an output layer.

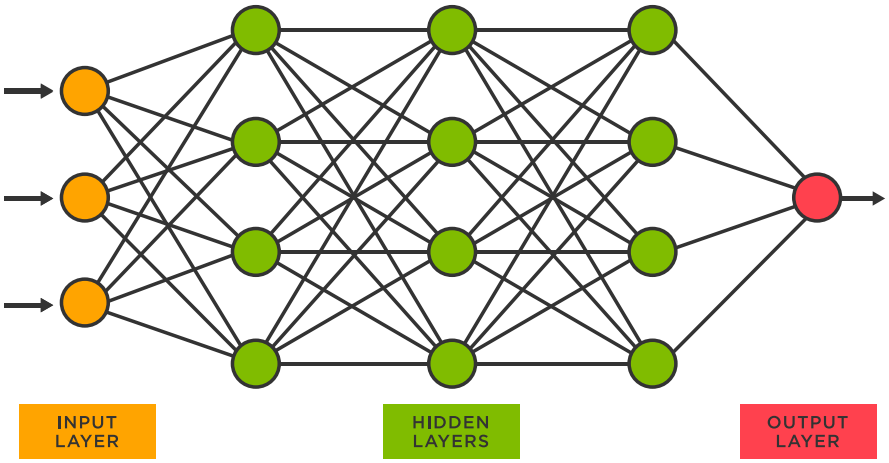

Figure 10: NN general layout.

The structure in figure 10 is just a general example; in fact, it is possible to have a single or multiple hidden layers, each one with a different number of neurons. In a classification problem such as the one in this work, the number of input nodes is determined by the number of features of the dataset, while the number of output nodes depend on the number of classes in which the dataset is organized. Neural networks optimize a compositional function (Equation 3), usually by exploiting gradient descend and back propagation algorithms. The objective function is the following:

$$\underset{A_j}{argmin}(f_M(\mathbf{A}_M, ..., f_2(\mathbf{A}_2, f_1(\mathbf{A}_M, \mathbf{x}))) + \lambda g(\mathbf{A}_j)) \tag{3}$$

where **x** indicates the input values and **y** the output results, here each matrix $\mathbf{A}_j$ denotes the weights connecting the neural network from a certain layer to the next one. Since it is a severely high dimensional problem, the presence of the regulator function $g(\mathbf{A}_j)$ plays a crucial role in avoiding overfitted results. In order for the neural network to map complex patterns, non-linear activation function can be applied to the neurons: these can be of many different kinds such as the most popular *ReLU* (rectified linear unit) or hyperbolic tangent. After determining the number and type of layers in the neural networks together with their activation functions and number of neurons in each layer, it is possible to start the training of the model.

The iterations during which the mapping of the data is performed are called epochs. In a well designed neural network, increasing the number of epochs leads to a better accuracy in the results, while still reaching a plateau value after a certain number of iterations.

Like ML-based models, also NN-based models are not to be considered trustworthy if not properly trained and validated: there should be a significant split between train data, which are used to develop the model, and test data which need to be unknown to the model and represent a crucial exam to its performances.

4.2 The Structure of the Neural Network

Similarly to what has been shown in the ML-based model case, the first issue when starting to develop the NN-based model was the huge size of the dataset: this problem was tackled by performing the preliminary steps of the model creation only on a small fraction of the dataset, and then performing the mapping on the complete dataset only after having finalized the structure of the NN.

The overall structure of the neural network structure is shown in the Figure 11 and Table 4.

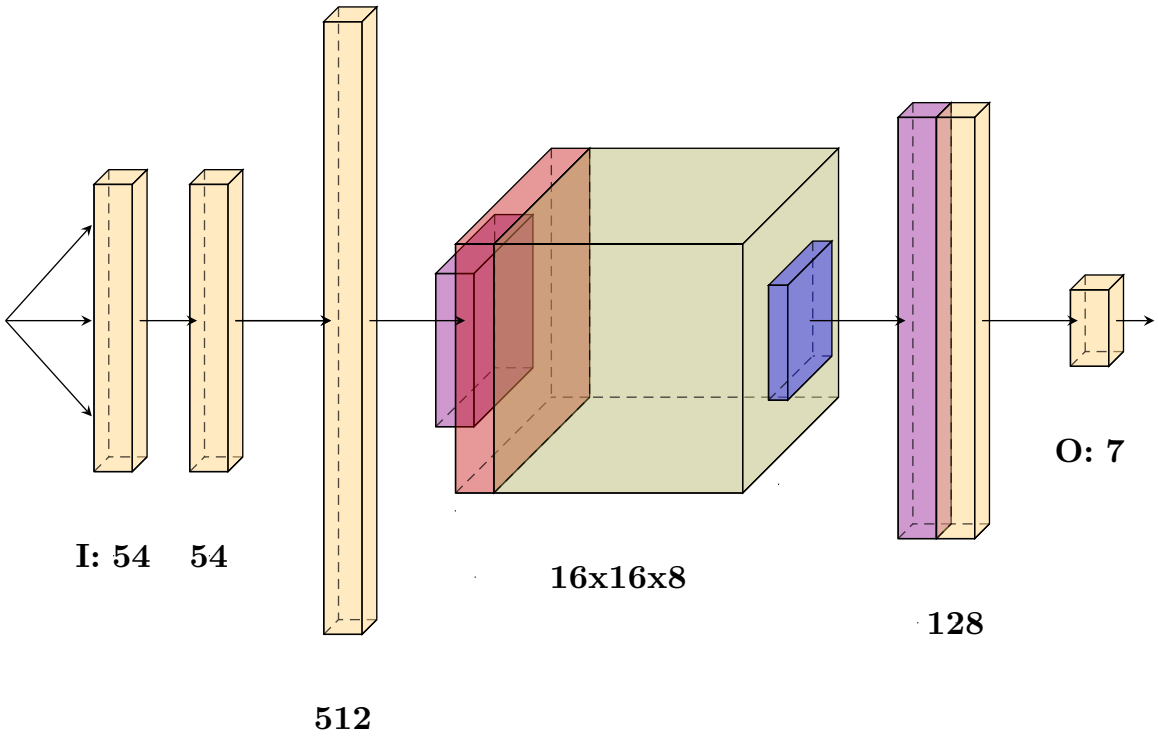

Figure 11: NN structure. Dense layers in yellow, reshaping l. in purple, upSampling l. in red, MaxPooling l. in blue.

Layer	Number of nodes/shape	Layer type	Activation function
I	54	Dense Input (I)	-
1	54	Dense	relu
2	512	Dense	hard-silu
3	(8, 8, 8)	Reshape	-
4	(16, 16, 8)	UpSampling2D	-
5	(15, 15, 32)	Convolutional	relu
6	(7, 7, 32)	MaxPooling2D	-
7	1568	Flatten	-
8	128	Dense	relu
O	7	Dense Output (O)	softmax

Table 4: Summary of NN layers

The first features to highlight are the number of neurons for the input and output layers, which are respectively 54, the number of feature labels in the dataset, and 7, the number of forest cover types. Both the input and the output layers are *dense* layers: this kind of layer is the most basic and commonly used, it represents the fundamental building block of neural networks, connecting every neuron in the previous layer to every neuron in the current one. After the input layer and two preliminary *dense* layers, the data is fed to the *reshape* layer, which has the simple function of converting the flat array into a three-dimensional structure. This operation is needed for the next steps, in which the data is fed to the *upsampling2D* layer, which artificially increases the size of the first two dimensions of the input data. In this case, the operation is carried out by *nearest* interpolation, which simply repeats rows and columns of the data. Since the data has been reshaped with increased dimensions, it is effectively a sort of 3D image, in which the pixels representing features 11-54 can be represented very effectively since they can only assume values of 0 or 1. This kind of structure is extremely suitable to be fed to the *convolutional* layer,which represents the core of this neural network.

This kind of layers extract features from the input data by applying convolution operations with kernels. Convolutions process fractions of the image, sliding through it and producing an output after the kernel operation (Figure 12).

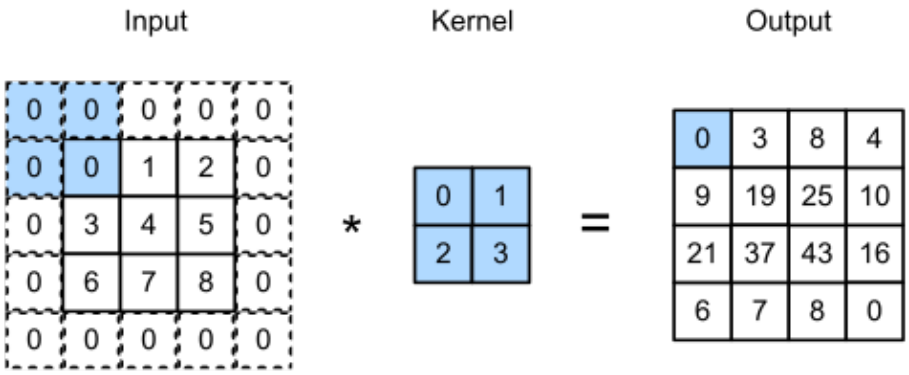

Figure 12: Convolution operation.

After the *convolutional* layer, the data is fed to a *MaxPooling2D* layer, which on the contrary to what

was done by the *upsampling2D* layer, downsamples the size of the data to reduce the total number of parameters. This kind of operation is very common when using convolutional neural networks and its function is to lower the computation cost of the network while also avoiding overfitting. Downstream to the pooling, compressed information is obtained and the data is passed to the *flatten* layer, which simply reconverts the input data to a flat array and to one final *dense* layer before the output layer.

The activation function used in the first *dense* layer is the *hard-silu*, which is a sigmoid-shaped function that combines linear a non-linear features in a simple form. The activation function used for the *convolutional* layer and the last *dense* layer before the output one is the *relu* function, which is one of the most commonly used ones in the development of neural networks, as it introduces non-linearity by thresholding input values at 0. Finally, the output layer was set with the *softmax* activation function, which is commonly used for the output when classification is needed. This function converts a vector of n numbers into a probability distribution of n possible outcomes, increasing the disparity between the values exponentially, this resembles a smoothed version of the *max value* function.

Other than a valid structure of the NN, it is important to preprocess the data to have better performances and predictability: this is achieved through the *Standardscaler*, which transforms the input data into a normal distribution of data: this kind of procedure is quite common in the development of ML and NN-based models, this is because bringing features to a similar scale ensures similarity between features and avoids numerical issues during training while also achieving faster convergence. In addition, another pre-processing procedure paramount for the development of the NN-based model is to convert the cover type categories in a one hot encoded vector. This procedure allows to represent categorical values as numerical values: each categorical value is converted to a categorical column and assigned a binary value of 0 or 1. A simple example of this procedure is reported in Figure 13.

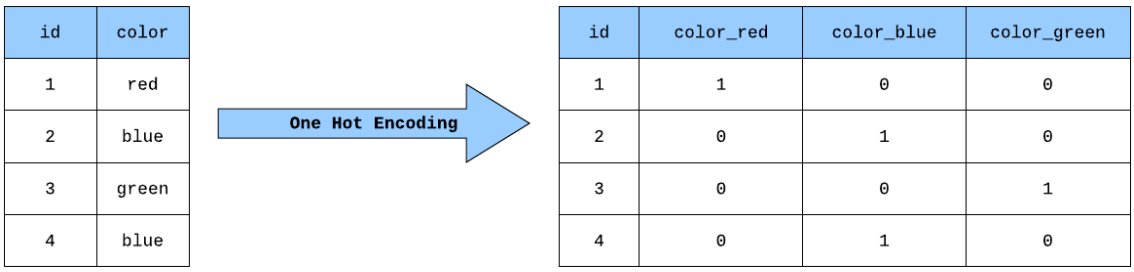

Figure 13: One hot encoding example.

Using one hot encoding is particularly useful when data categories have no significant relationship between one another nor any order. It allows the training data to be more usable and readable, other than making it possible to reach better computational performances. Focusing on the example in Figure 13, when trying to identify, as an example, the color blue from the categorical table, the obtained identification would be [2,4]. After switching to one hot encoding, every color is defined

as a vector of N-1 elements equal to 0 and one element equal to 1, like in the example where blue is identified by the vector [0, 1, 0]. This procedure allows to have a ordered representation of data, with every category being identified by a simple and unique vector, which not only provides the information about that particular category, but also intrinsically communicates the absence of all the other categories.

Another important feature to be defined in the development of a NN-based model is the definition of the loss function, which compares the target with the predicted output value and measures how well the the model is performing a certain task, in this case how the classification is being performed. The objective of the neural network training is to minimize the loss function, which can be defined in different ways. In the present work, the *categorical crossentropy* loss function was chosen, as this is a popular loss function particularly used when dealing with classification problems. One major aspect of this kind of loss function is that, similarly to many other ones, it expects to be provided with labels in a one hot encoded representation.

4.3 Reults and Further Improvements

The training of the model on the reduced size dataset granted satisfactory results, allowing to proceed in the evaluation of the model performances on the complete database. The obtained confusion matrix is shown in Figure 14.

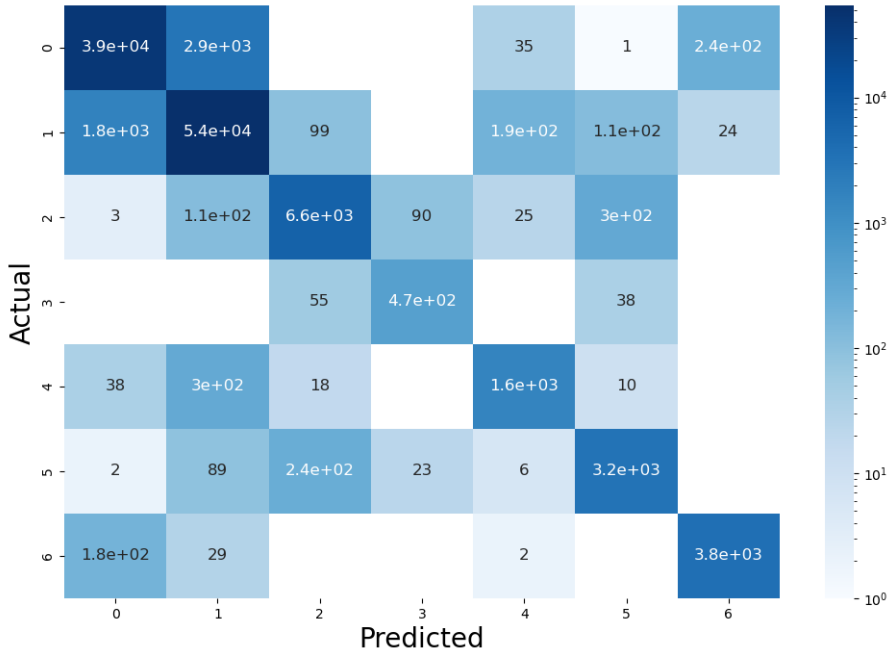

Figure 14: Confusion matrix - NN-based model, rescale only.

It is clear that the model correctly identifies most of the entries, with the majority of the misclassified data being from one of the first two cover types. This is clearly caused by the fact that there is much more data belonging to one of this two classes than any other.

Before discussing in quantitative terms about the model performances, a further analysis about a possible procedure to obtain more reliable classification is needed. As stated in the introduction section, this dataset is strongly unbalanced, with the first two cover types making more than the 90% of all data points. The problem related to this kind of datasets is that the classifier could have poor performances on the less dominant classes. Since those are not less important than the most dominant ones, some additional steps were explored as a solution to have a more robust model. This consisted in applying the same over/undersampling techniques explained in the previous section of this work for the ML-based model. It is important to note that both the procedures for rescaling and rebalancing must be performed after the preliminary split of the data in order to avoid data leakage.

Figure 15 summarizes the performances of the NN-based model obtained by using the following procedures: no rescaling (raw), rescaling, *SMOTE*, *SMOTEEN* (SNN), *SMOTETomek* (SM) and *RandomUnderSampler* (RUS)

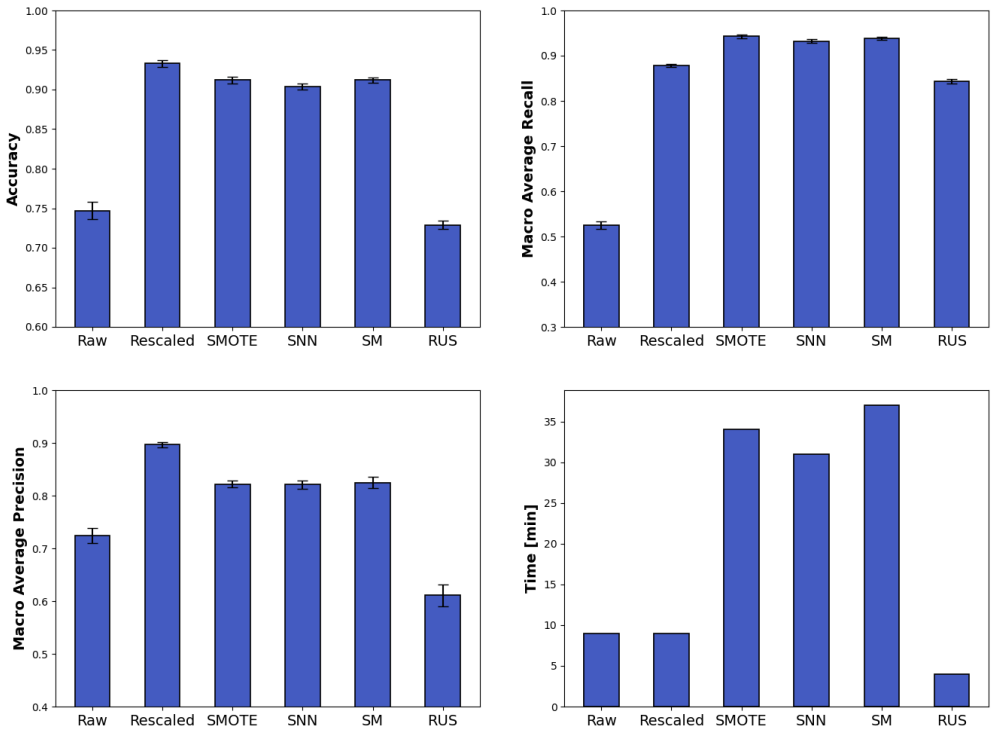

Figure 15: NN-based model performance summary, execution time is referred to GeForce RTX 3060 Mobile as benchmark.

The first metric to be analyzed is accuracy: excluding the raw and RUS case, the accuracy reached is quite high for all the other techniques. Both the recall and precision results confirm the general trend seen from the accuracy results, with the raw and RUS case being noticeably worse than the other ones, while the SMOTE, SNS, SM and rescale case show similar results.

Lastly, the training time has also been evaluated. It shows a much lower value for the raw and rescaled case, which are substantially identical. This is due to the fact that both models are fed the same amount of data, while the techniques which perform data augmentation predictably require significantly longer training time. The RUS technique, on the other hand, shows a much lower training time, which is a direct consequence of its operations of undersampling. In conclusion, a NN-based model for the classification of the data from the forest cover type dataset has been successfully developed. In particular, the rescaled solution with no rebalancing appears to offer the best compromise between training time and accuracy for the obtained results. This model reaches an accuracy of more than 93% on the test set with a training time being about a third of the *SMOTE* based models. All results were obtained by training the model with a train/test split of 80/20 and 15 epochs. Slightly better results (around 94% accuracy) could be obtained by increasing this number, however, by analysing the trend of the accuracy as a function of the number of epochs (Figure 16), it is safe to say that a plateau value was almost reached. Furthermore, no signs of overfitting were detected, with a very good matching of train accuracy to test accuracy, in general less than 0.5% difference. In some cases even higher values of test accuracy with respect to the training one have been noted.

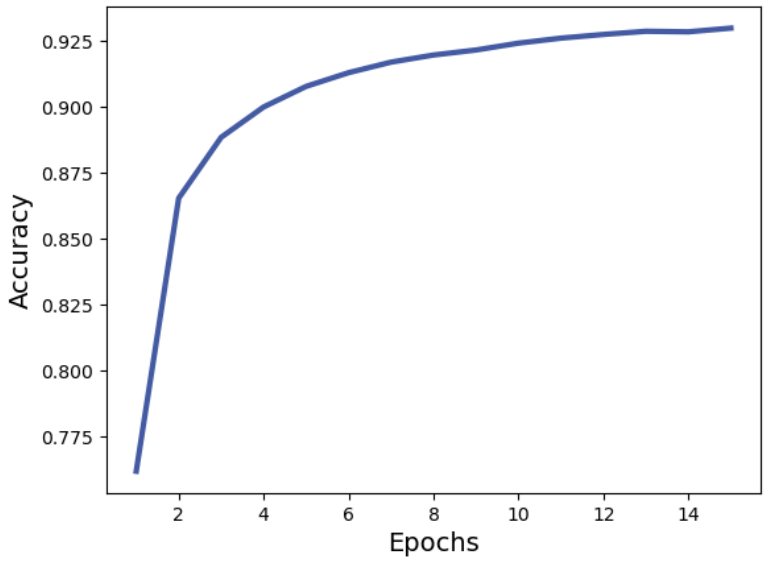

Figure 16: Example of accuracy trend by the NN-based model vs the number of epochs - Rescale only.

5 Optimized Neural Network Based Model

5.1 Structure of the Optimized Neural Network

After successfully developing a NN-based model with more than acceptable results (Figure 15), a further development of this model was proposed, focusing on obtaining a model as simple and light as possible without an excessive loss of accuracy. As shown in Figure 17 and Table 5, this new neural network has a structure similar to the other one developed in this work and shown in Figure 11.

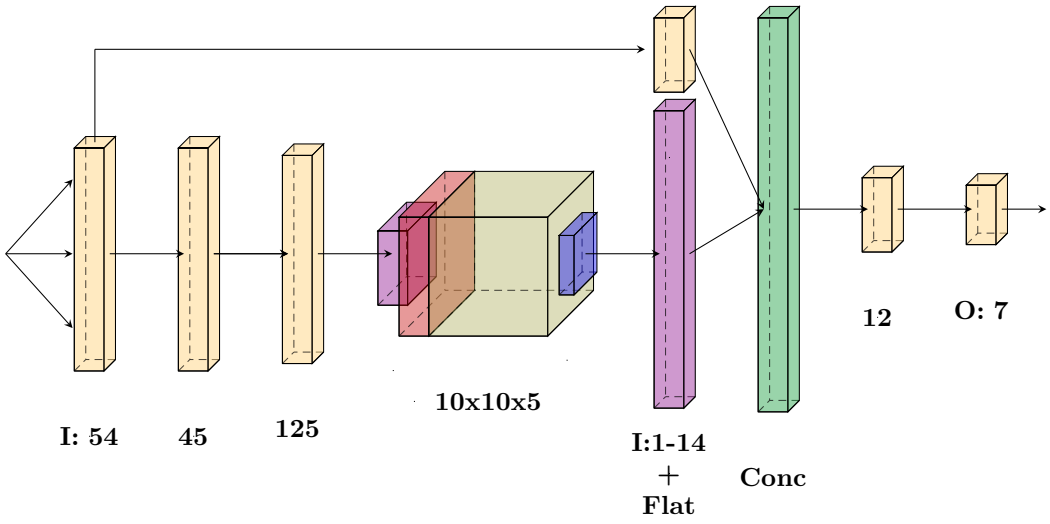

Figure 17: NN structure - Optimized case. Dense layers in yellow, reshaping l. in purple, upSampling l. in red, MaxPooling l. in blue, concatenation l. in green.

Layer	Number of nodes/shape	Layer type	Activation function
I	54	Dense Input (I)	-
1	45	Dense	hard-silu
2	125	Dense	relu
3	(5, 5, 5)	Reshape	-
4	(10, 10, 5)	UpSampling2D	-
5	(9, 9, 10)	Convolutional	relu6
6	(4, 4, 10)	MaxPooling2D	-
7a	160	Flatten	-
7b	14	Delayed Input	-
8	174	Concatenate	-
9	12	Dense	relu
O	7	Dense Output (O)	softmax

Table 5: Summary of NNO layers

Similarly to the other NN, an upsampling layer was introduced before feeding the data to the *Convolutional* layer, which is followed by the usual *MaxPooling2D* layer. Though having a similar structure, this NN is characterized by having a much smaller number of nodes compared to the previous

case. This is done with the aim of obtaining the lightest model possible. However, this important reduction of the size of the layer can lead to a significant loss of features, especially after the *Max-pooling2D* layer. In order to tackle this issue, the first and most important 14 features (as inspected in the machine learning model section) are fed again after the pooling. This makes it possible to keep a smaller size of the model while also preserving important information.

5.2 Results and Further Improvements

The obtained confusion matrix from for this new model is shown in Figure 18.

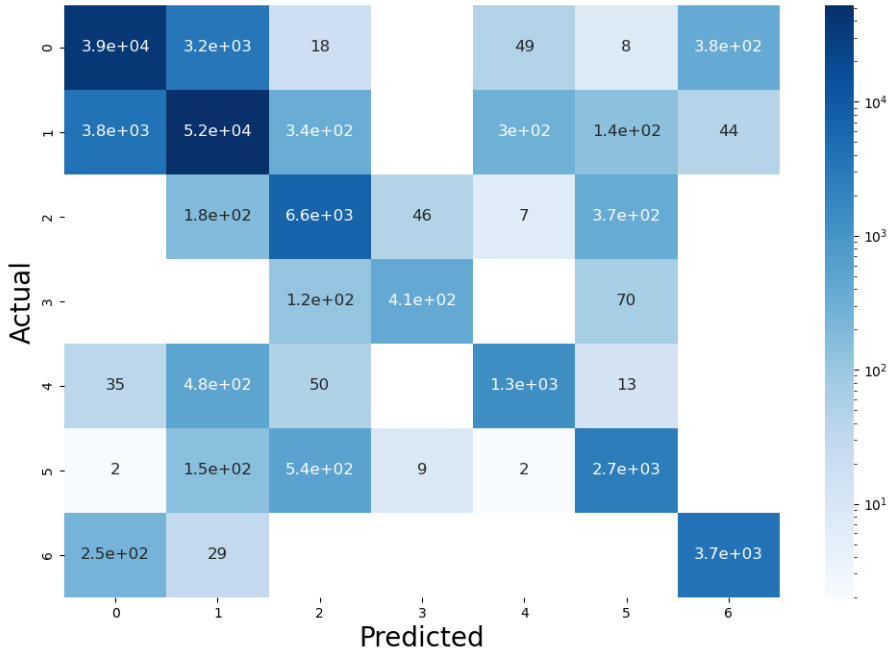

Figure 18: Confusion matrix - NNO

Another useful operation that can be performed after the training to further reduce the size of the model is quantization (Figure 19), a technique that allows to reduce the computational cost and memory usage by representing the weights and activation function of the neural network with lower precision data types, replacing the usual 32-bit floating point (float32). This procedure could seem futile, but nowadays the major bottleneck of GPUs is virtual memory usage (VRAM), and parameters quantization can drastically shrink the size of the model and potentially speed up calculations, especially for really large models or consumer-grade hardware. In this work, a converter from the library *TensorFlow lite* was implemented, relying on an experimental setting that converts weight into 16-bit precision and activation functions to 8-bit.

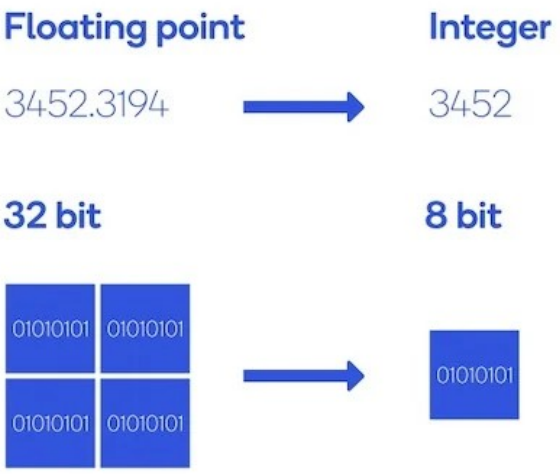

Figure 19: Quantization.

It is finally possible to compare the results obtained in the previous section and the ones just shown, with and without quantization.

Model	Accuracy [%]	Parameters	Size [kB]	Training time [min]
NN - R	93.3	233.9k	2850	9
NNO - Nquant - R	90.3	10.6k	172	4
NNO - quant - R	90	10.6k	19.5	4

Table 6: Results summary

where R indicates that the dataset has been rescaled before starting with the training. This table shows that by using the optimized neural network (NNO) it is possible to obtain a much simpler and lighter model with just a minimal loss of accuracy, while also requiring a lower training time. The significant reduction of size is caused by the extreme decrease in the number of parameters, by more than an order of magnitude. Another important feature to highlight is the fact that the NNO models have the same number of parameters both in the non quantized and quantized case: this is due to the fact that the quantization is done after the training, so the size of the model is reduced without changing the number of parameters but only their precision.

In conclusion, it is safe to say that these extremely light models could have a wider range of applications than the ordinary NN-based models because of their reduced size: this could happen in scenarios where devices with limited computation memory or power are employed.

6 Conclusions

As a final remark, the reason for the development of the optimized models was because of the ever growing demand in faster and more efficient algorithms. A light model ideally is capable of accomplishing both of those tasks. Nowadays, one of the greatest limitations in IT is having programs capable of being run locally due to their need of computational power and RAM availability. Moreover, running them on the cloud or on an external cluster could pose a serious threat to privacy or data security. A decisive advantage of running complex programs locally is the increased ability to customize and adjust models and parameters, without depending on external resources.

For this reasons, both the ML-based and the NN-based models were reduced in dimensions to obtain optimized models. However, the reduced model complexity and parameters quantization leads inevitably to a drop in accuracy. In a way, each model, both the complex and the lighter ones, are thought to be the best possible ones, since optimization is a trade-off between the number of parameters and the accuracy. Theoretically, by gradually increasing the complexity of the system it could be possible to reconstruct the Pareto curve. In any case it was observed that, for this specific classification, the rebalancing lead to almost no advantage, while the rescaling only increased accuracy. This was achieved without significantly increasing the size of the model or slowing down calculation time, which is why it was adopted as the optimal solution for every one of the models evaluated.

For what concerns mere accuracy, the ML models showed a peak performance of 97% with the implementation of a *Bagging* classifier, on the other hand NN models a less promising 93% but with a much lower model weight. In general NN are much more complex to handle especially when the number of layer/nodes is rather high. Their flexibility allows them to fit a wide array of data patterns and solve diverse problems, but it comes at a price, since it also introduces challenges in interpretability and consequent parameters tuning. On the other hand ML models are usually not suited for the recognition of patterns of high dimensionality and non linearity, but offer a more straightforward comprehension of the logic of the model. In this study, the dataset, comprising a relatively modest 54 features, allowed machine learning models to achieve impressive performance, particularly evident when the primary criterion was raw performance. However, neural networks demonstrated their superiority in terms of flexibility and model efficiency when it comes to reducing the number of parameters to the bare bones, with a astonishing 19.5kB model.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`. Software available from tensorflow.org.

[2] J. Blackard. Covertype. UCI Machine Learning Repository, 1998. DOI: https://doi.org/10.24432/C50K5N.

[3] S. L. Brunton and J. N. Kutz. *Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control*. Cambridge University Press, 2 edition, 2022.

[4] G. Lemaître, F. Nogueira, and C. K. Aridas. Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning. *Journal of Machine Learning Research*, 18 (17):1–5, 2017. URL `http://jmlr.org/papers/v18/16-365.html`.

[5] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[6] A. Stagni. Slides of the course Data Science in Chemical Engineering. MSc in Chemical Engineering, Politecnico di Milano, 2023-2024.

List of Figures

List of Tables