1. **Presentation of the Dataset and Expected Objectives**

The dataset under study consists of 15 columns and 215 rows. The objective we worked on was to predict, based on the information present in this dataset, whether a student will be employed upon completing their studies or not. This is a supervised classification problem, binary in our case since there are only two possible outcomes. For each student, we have 15 features related to their education (results from previous years, specialization field, work experience, etc.). The method implemented through this project was the perceptron algorithm. However, we also used certain libraries to compare different methods with our perceptron, including DecisionTree and SVM.

Finally, we made an effort to use algorithms that are as generic as possible. Despite the relatively small size of our dataset, we aimed to use methods that work equally well for much larger datasets.

Our project is divided into four main parts:

- Data preprocessing to make it usable.

- Implementation of the perceptron and gradient descent.

- Hyperparameter tuning.

- Comparison with other existing methods.

1. **Preprocessing**

Dataset before preprocessing :

Une image contenant table

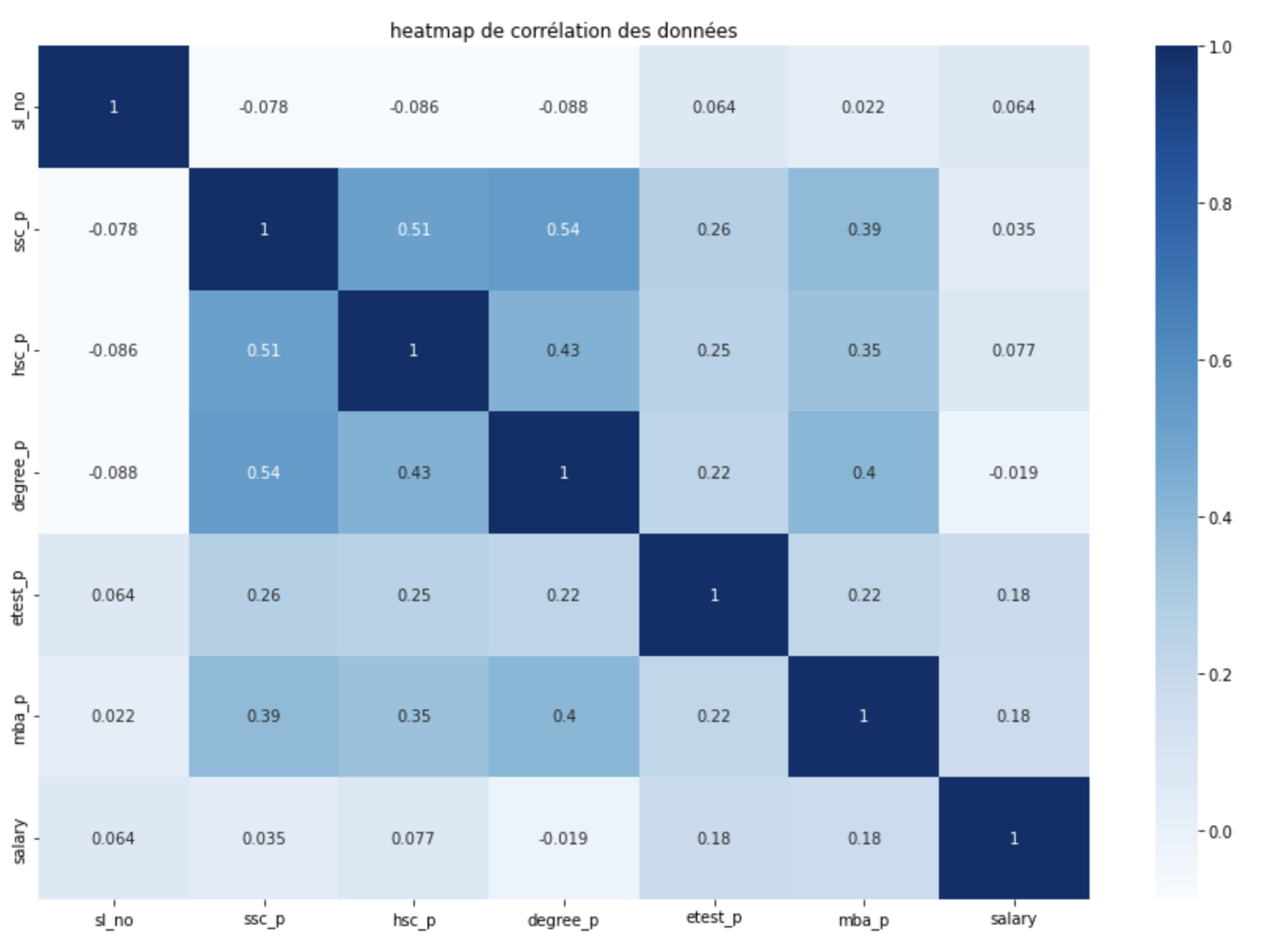
Description générée automatiquement

Firstly, we noticed that not all values in the dataset were numerical. Therefore, we had to perform various preprocessing steps on the dataset to create a new dataset that could be used for our algorithms. For features like gender, ssc\_b, hsc\_b, hsc\_p, degree, workex, specialization, and status, as they are categorical features, we encoded them into one-hot vectors. This means we transformed them into binary values. For example, we assigned 1 if the student is male and 0 if the student is female. This resulted in new columns for each attribute, but most of the created columns were redundant. For instance, for the gender attribute, we obtained columns for 'M' and 'W', but one of them was unnecessary because if the value in the 'M' column was 1, the value in the 'W' column would inevitably be 0. Therefore, we could safely discard one of the two columns without losing information.

Another issue we encountered was with the 'salary' column. Firstly, the values in this column were significantly larger than the others, which would have required centering and normalization to prevent the results from being heavily influenced by this column. Additionally, there were missing values in this column. Consequently, we decided to remove the 'salary' column from the dataset.

We also examined the frequencies of individuals who were placed (the variable we wanted to predict) versus those who were not placed. The purpose was to assess whether we needed to weigh the error costs differently for the gradient descent algorithm if one class was significantly more prevalent than the other. However, the values were sufficiently balanced (148 placed, 67 not placed), so we did not need to apply weighting.

Finally, we looked into whether certain columns were correlated, with the possibility of conducting Principal Component Analysis (PCA) or simply removing columns that did not provide useful information. For our project, this step wasn't particularly necessary given the small number of columns, but we wanted to follow a general approach that would be applicable to larger datasets, thus working with as much generality as possible.



We can observe that the columns 'hsc\_p' and 'ssc\_p' are somewhat correlated. Therefore, we may consider removing one of these two columns.

Here is an overview of the dataset we obtain after all the transformations.

**Une image contenant table

Description générée automatiquement**

1. **Implementation of the Perceptron and Gradient Descent**

The model we chose is the one we learned in class:

Une image contenant texte

Description générée automatiquement

It allows us to model a hyperplane defined by the weights, denoted as 'w = (w1, ..., wn)'. If the chosen point lies above the hyperplane, we return 1; otherwise, we return -1 based on the sign of the output value. The objective is to find the best values for the weights to separate the points representing students who were placed from those who were not. This implies that we need to define a metric to calculate the overall error of the model to determine its performance. For this purpose, we used the following cost function:

Une image contenant texte

Description générée automatiquement

This cost function works particularly well because, on one hand, it only deals with misclassified points, and on the other hand, it weights them by their distance from the plane. Therefore, the farther a misclassified point is from the plane, the more it contributes to increasing the cost function.

It's also worth noting that in our dataset, the column we want to predict takes values in {0,1}, whereas the perceptron predicts values in {-1,1}. So, we simply replaced the 0s with -1s.

To find the best values for the weights, we use the gradient descent algorithm. This algorithm updates the weights at each iteration by moving in the direction given by the gradient (which ensures that along this direction, the slope is steepest). Therefore, our goal is to minimize the cost function at each iteration in order to converge towards a minimum value. Mathematically, this can be expressed as follows:

Une image contenant horloge, montre

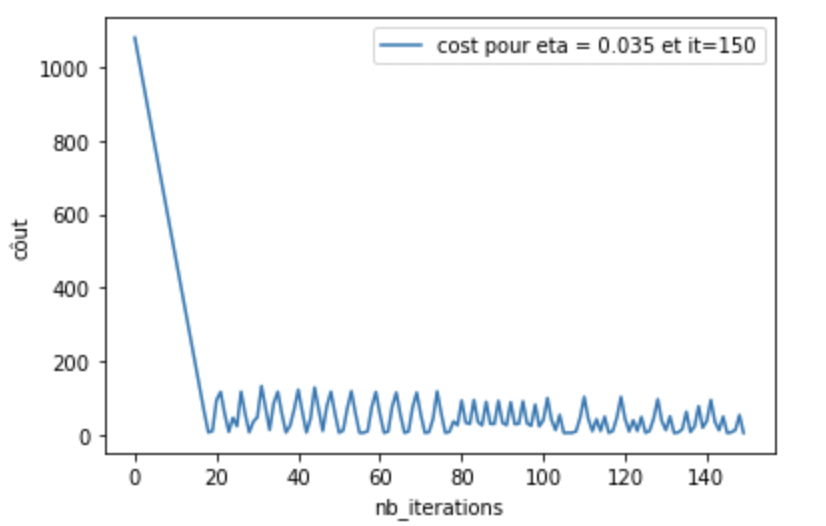
Description générée automatiquement

With Θ being the parameters we want to update (w in our case) and η being the learning rate, which is a parameter we will discuss later. In our case, the explicit formula with the calculated gradient is as follows:

Une image contenant texte, horloge

Description générée automatiquement

So, we implemented the perceptron and gradient descent algorithms from scratch, following these formulas and using numpy for matrix calculations. We also computed the cost to visualize the learning progress of our model.

Here is the evolution of the cost function that we obtain during the gradient descent algorithm.

It's evident that the cost function values decrease as the algorithm progresses. This clearly indicates that the weights are updating, and our model is becoming more and more accurate.

Given that our dataset is small, we start converging after around 20 iterations, and we quickly oscillate around the minimum. For larger datasets, it generally takes more iterations to converge.

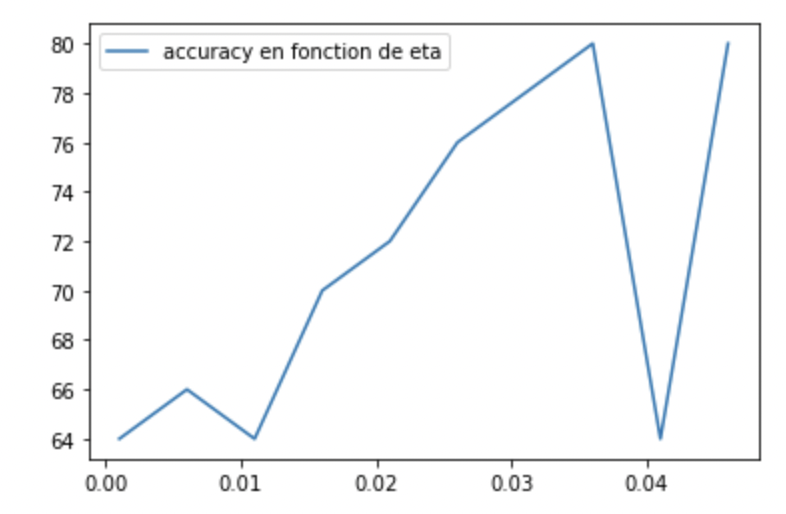
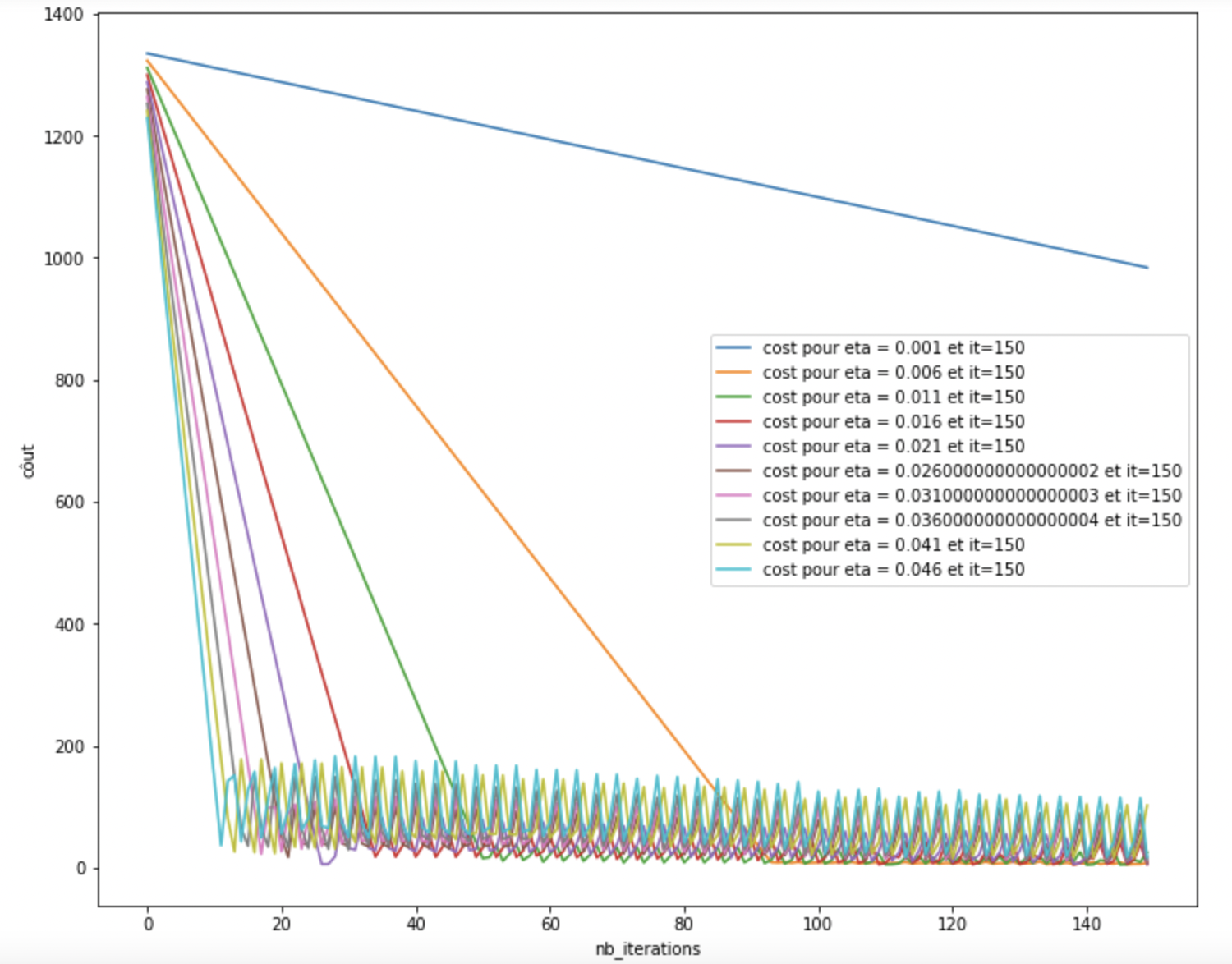
We achieve an accuracy of 78% on the test set and 85% on the training set, which are quite good results given the size of our dataset. Furthermore, the small difference between these two values indicates that our model is not overfitting significantly and generalizes well.

Note: In the Jupyter notebook, we used an initial weight value of 12.22, but this is arbitrary, as it was chosen randomly.

1. **Hyper parametres**

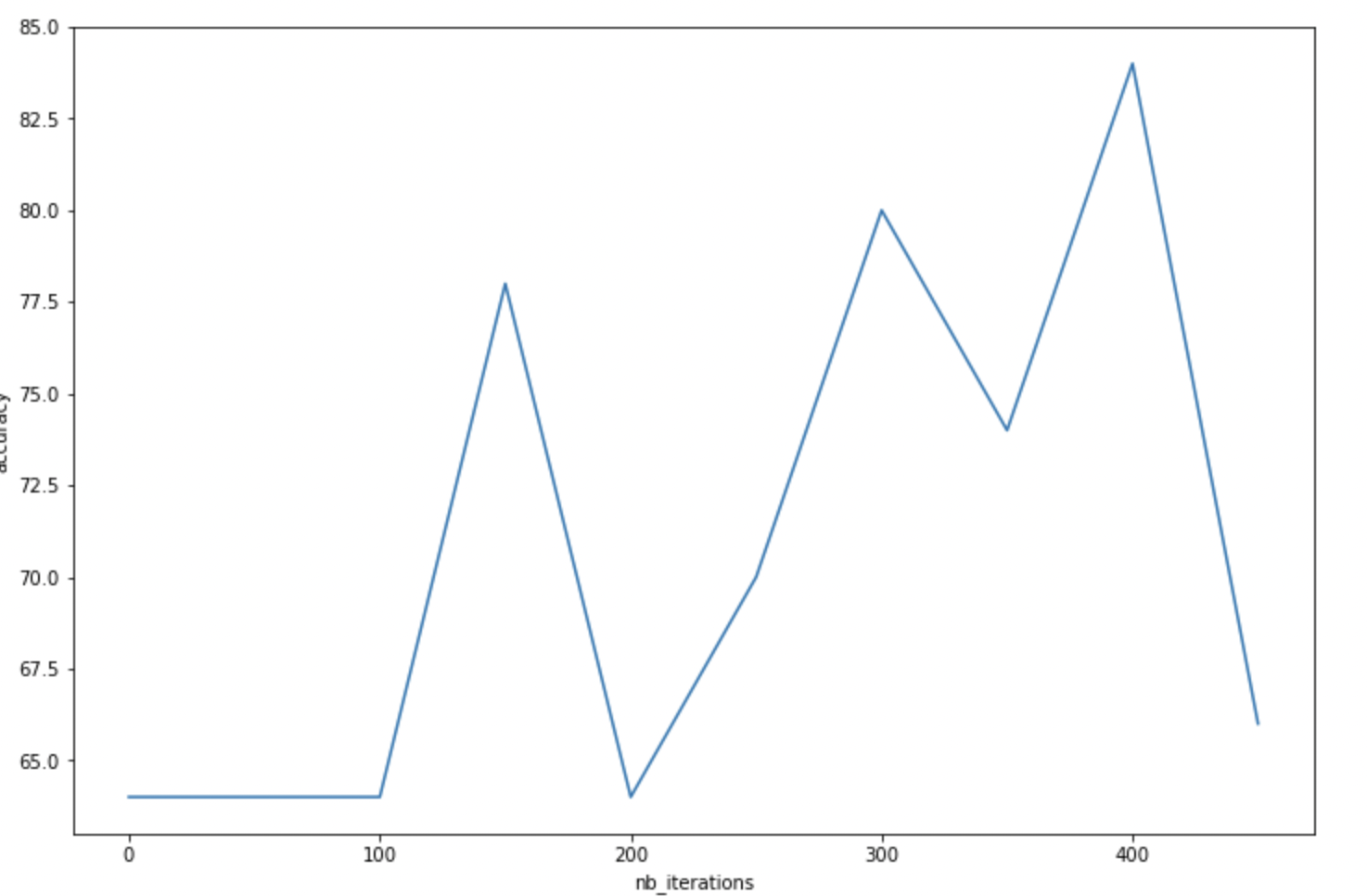
Now, let's focus on the choice of hyperparameters and how to make our model even more efficient.

Firstly, concerning the learning rate, we modeled the evolution of the cost function during gradient descent when varying the learning rate.



We observe that for learning rates (η) below 0.01, the model converges too slowly, and the accuracy is not very good. Therefore, we chose η = 0.035. With this value, the accuracy is maximized, and the model converges reasonably quickly.

Now, let's see what happens when we increase the number of iterations.



The accuracy fluctuates significantly depending on the number of iterations. We chose 150 iterations because the performance is quite good (78%), and the model is computationally less expensive compared to selecting 400 iterations.

Finally, we also wondered if we could remove certain columns from the dataset to make it lighter and potentially improve performance. Since we are using a perceptron, there's a chance it might work better with fewer binary variables. To explore this, we looked at which binary columns provided less information and removed them (columns 'Commerce' and 'Science'). However, it turns out that the model's performance is worse without these columns (64% without them compared to 78% with them). So, it's better to keep these columns.

1. **Comparison with other models**

Our main goal was to create a functional and efficient perceptron. Therefore, we did not delve deeply into the workings and optimization of other models. We used the sklearn library to implement them.

Note: Model performances may vary slightly when running the notebook, but overall they remain close.

The first two models we used are decision trees. The difference lies in the criterion chosen to assess leaf purity. For the first one, we used the entropy criterion and obtained an accuracy score of 84%. For the second one, we used the Gini criterion and obtained a score of 80%. Thus, the entropy criterion performs better for our dataset. This is understandable because it is more qualitative but slower to calculate due to its formula involving logarithms. In contrast, the Gini criterion is faster to compute but slightly less effective in terms of results. Since our dataset is small, we do not have any computational time issues, so it's better to use the entropy criterion.

Une image contenant texte, montre

Description générée automatiquement

However, when we try these models on the training set, we achieve 100% accuracy, which means the model is completely fitted to the dataset and overfits. At the same time, given that we have only 215 data points, we obtain pure leaves in just a few iterations.

Next, we used a Random Forest tree, which achieved an accuracy score of 86%, slightly better than the Decision Tree.

Finally, we used an SVM, which achieved a score of 82%, a bit higher than our perceptron. This result is not surprising because, like the perceptron, an SVM is a linear classifier. Therefore, it's quite logical to obtain relatively similar results.

**Conclusion and critics**

The method that achieved the best scores is the Decision Tree. This is because this algorithm works very well when there are a large number of binary variables (which is the case with our dataset). However, achieving 78% accuracy with the perceptron is still quite good, especially on a dataset of this size.

Despite the dataset's small size, we aimed to keep the algorithms as general as possible. Thus, the algorithms we built would work just as well on much larger datasets. In fact, they would perform even better with more data points, and accuracy would be less sensitive to the choice of hyperparameters.

Finally, the primary challenge with our dataset, regardless of the method chosen, is overfitting. Due to the dataset's size, after a few tens/hundreds of iterations, we end up with a model that fits the data too closely.

Summary

|  |  |
| --- | --- |
| Model | accuracy |
| perceptron | 78% |
| SVM | 82% |
| Decision tree (gini) | 80% |
| Decision tree (entropie) | 84% |
| RandomForestTree | 86% |