Final project: Predicting students grades

APA

Leo Arriola
Zixuan Sun

January 2021

Universitat Politècnica de Catalunya

Facultat d'Informàtica de Barcelona

Contents

1 Introduction

In this project, we are going to study the correlation between demographic attributes and their relationship with academic performance. The dataset we've chosen is formed by attributes that a priori are interrelated with the academic performance of secondary school students. This includes the level of education of their parents, their parent's job and other similar attributes. The dataset allows us to treat it as a regression problem or through classification problem. In summary, the dataset has 33 attributes (some numerical and some categorical) and a total of 395 samples.

The dataset was extracted from the `UC Irvine Machine Learning Repository`, it is a collection of databases, domain theories, and data generators that are used by the machine learning community for the empirical analysis of machine learning algorithms. It was originated by *David Aha* and fellow graduates from `UC Irvine`.

The full dataset can be obtained from here, this includes the `.csv` files and other descriptive documents about the dataset. Regarding the source of this dataset, we know that it was collected through school reports and questionnaires from a Portuguese school (in *Cortez & Silva, 2008*).

2 Problem description

We decided to treat the problem as a regression problem, therefore we are going to try to predict the academic grades of the students.

The original dataset includes two datasets, one for the subject of Portuguese and another for the subject of Mathematics, as we are computer scientists we would like to work with the dataset for the subject of Mathematics, it can be found in the `student-mat.csv` file inside the original zipped file.

In order to predict the student's grades, our target variable is going to be '`G3`'. This attribute corresponds to the final grade of the course of Mathematics.

2.1 Related previous work

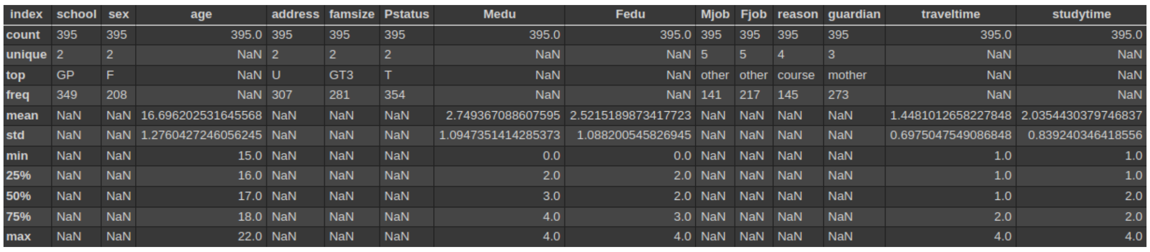
As it is specified in the `UC Irvine Machine Learning Repository` description, the dataset was extracted by *Cortez* and *Silva* [CS08]. The dataset was originally created purposely for this article, and that is why it has accumulated nearly 600 citations. There are a few more articles that set problems similarly.

Their study objective was to predict student's performance using 4 different models (i.e., *Decision Trees*, *Random Forest*, *Neural Networks* and *Support Vector Machines*) and tried including and excluding three attributes selections (`G1`, `G2` and `G3`). Regarding their results, they obtained high accuracies while including older performance attributes.

3 Data exploration process

In order to get a good notion of the data we were working with, we first checked the meaning of the attributes in the `student.txt` file, there we found a short description for each of the different attributes.

Our first step to take after loading the data is to call the method `describe(include='all')` of `pandas.Dataframe` class, this function gives some basic statistics, so we have a general sense of the characteristics of the dataset.



index	school	sex	age	address	famsize	Pstatus	Medu	Fedu	Mjob	Fjob	reason	guardian	traveltime	studytime
count	395	395	395.0	395	395	395	395.0	395.0	395	395	395	395	395.0	395.0
unique	2	2	NaN	2	2	2	NaN	NaN	5	5	4	3	NaN	NaN
top	GP	F	NaN	U	GT3	T	NaN	NaN	other	other	course	mother	NaN	NaN
freq	349	208	NaN	307	281	354	NaN	NaN	141	217	145	273	NaN	NaN
mean	NaN	NaN	16.696202531645568	NaN	NaN	NaN	2.749367088607595	2.5215189873417723	NaN	NaN	NaN	NaN	1.4481012658227848	2.0354430379746837
std	NaN	NaN	1.2760427246056245	NaN	NaN	NaN	1.0947351414285373	1.088200545826945	NaN	NaN	NaN	NaN	0.6975047549086848	0.839240346418556
min	NaN	NaN	15.0	NaN	NaN	NaN	0.0	0.0	NaN	NaN	NaN	NaN	1.0	1.0
25%	NaN	NaN	16.0	NaN	NaN	NaN	2.0	2.0	NaN	NaN	NaN	NaN	1.0	1.0
50%	NaN	NaN	17.0	NaN	NaN	NaN	3.0	2.0	NaN	NaN	NaN	NaN	1.0	2.0
75%	NaN	NaN	18.0	NaN	NaN	NaN	4.0	3.0	NaN	NaN	NaN	NaN	2.0	2.0
max	NaN	NaN	22.0	NaN	NaN	NaN	4.0	4.0	NaN	NaN	NaN	NaN	4.0	4.0

Figure 1: Output for `Data.describe(include="all")`

After seeing numerically the main characteristics of the dataset, we can start plotting some histograms to see data distributions for the attributes and to detect any irregularities. Before doing so, we have to separate the variables into numerical and categorical. To separate them, we used the following lines of code:

```
#identify all categorical variables
cat_columns = Data.select_dtypes(include='object').columns

#identify all numerical variables
num_columns = Data.select_dtypes(exclude='object').columns
```

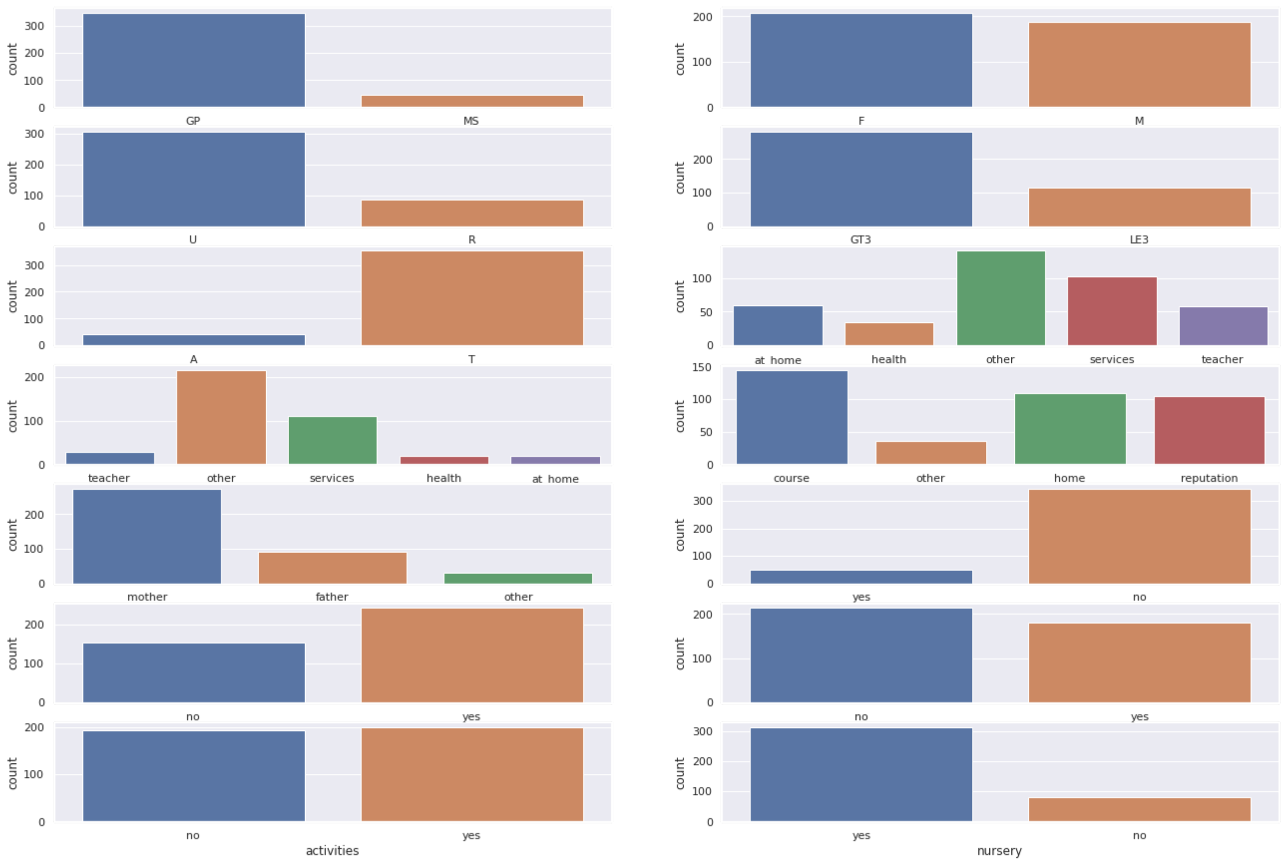


Figure 2: Histograms for categorical attributes

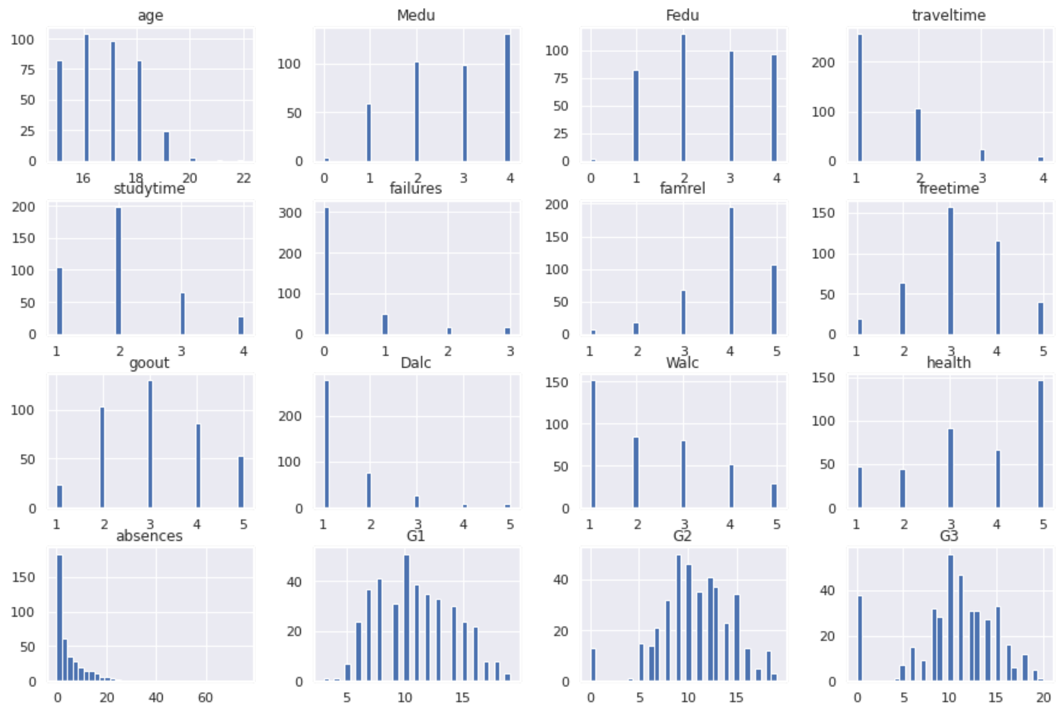


Figure 3: Histograms for numerical attributes

As we can see in the *Figure 3*, some numerical attributes look a bit off (in the sense of the general aspect of a numerical attribute). That is because, they are actually ‘categorical’ attributes, but they are already mapped into a numerical label (like the attribute `freetime` for example).

3.1 Data preprocess

After briefly exploring the data, we started preprocessing the data so that it would ease the problem resolution and the process of modeling it.

3.1.1 Missing values

We started by finding and treating any lost and missing values.

```
Data[:].isnull().sum()
```

By executing the line of code above, we found that our dataset didn’t have any missing values. The `isnull()` function allows detecting missing categorical and numerical values. All attributes returned 0 missing values. We also checked the original dataset description just in case if a ‘missing’ was actually coded with a numerical value.

3.1.2 Outliers

To find any outliers, we separated the attributes in different categories. We did this considering the different value scales. We first searched for outliers in the numerical attributes that ranged from 0 to 5.

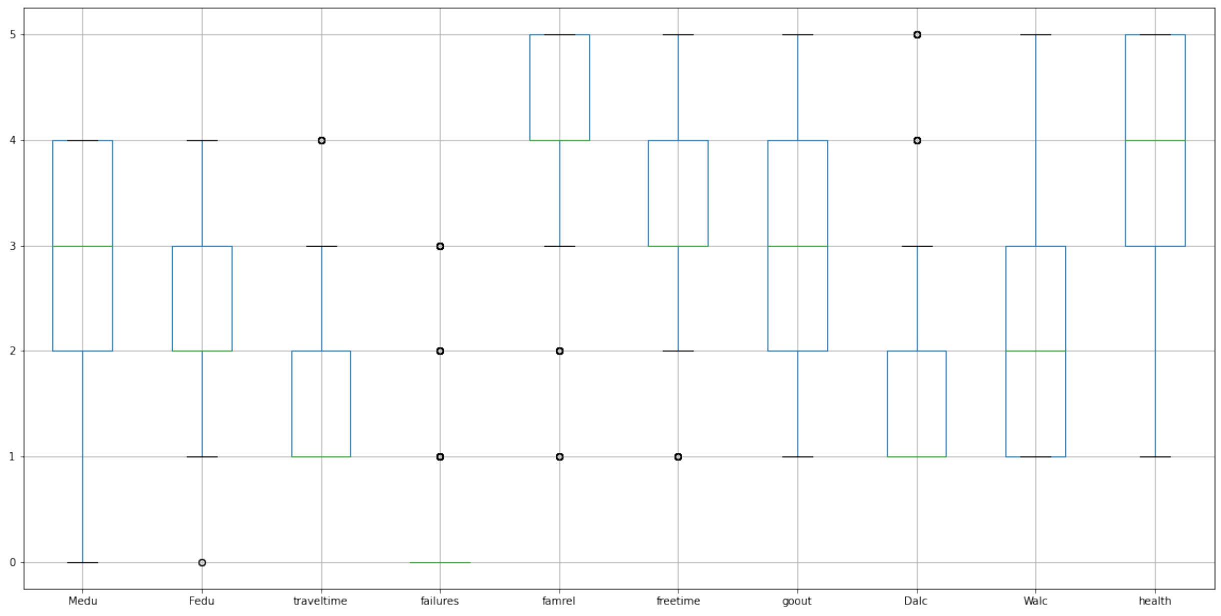


Figure 4: Box plots for numerical attributes of range `[0,5]`

Most of these attributes could be considered categorical, as they're only a numeric in order to indicate an element from a set of options. For example, the attribute `Medu` is described as indicating the mother's education, meaning (*numeric*: `0 - none`, `1 - primary education` (4th grade), `2 - 5th to 9th grade`, `3 - secondary education` or `4 - higher education`)

Taking that into consideration, we observe some values that could be considered outliers in the attributes '`failures`', '`famrel`' and some more. Regarding their treatment, we decide to maintain them, as modifying them would result in losing important information.

The rest of attributes we checked for outliers were the attributes that had bigger range values, these were the number of '`absences`' and the '`age`' of the students.

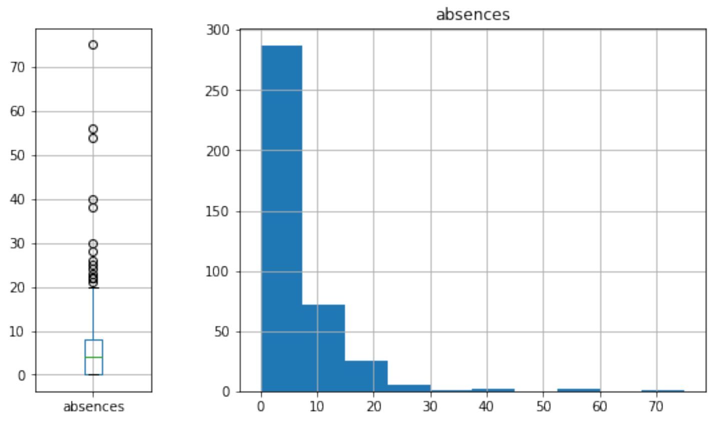


Figure 5: Box plot and histogram for '`absences`'

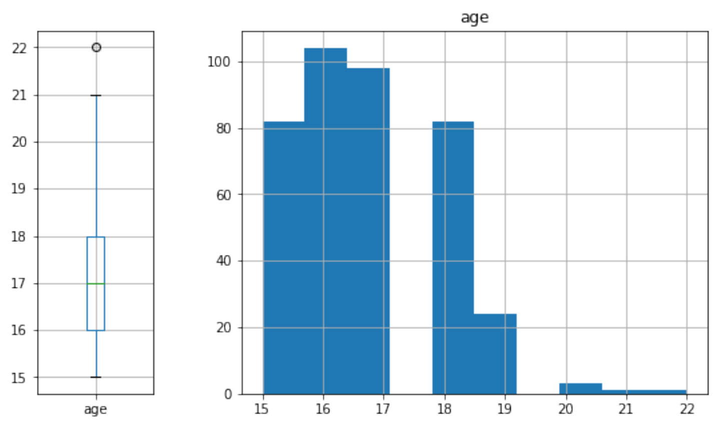


Figure 6: Box plot and histogram for 'age'

To detect outliers on these attributes, we used the **IQR Method**. This method decides which data points are outliers by looking at the distance between quartiles.

```
Q1age = Data['age'].quantile(0.25)
Q3age = Data['age'].quantile(0.75)
IQRage = Q3age - Q1age

small_outliers_age = Data['age'] < (Q1age - 1.5 * IQRage)
big_outliers_age = Data['age'] > (Q3age + 1.5 * IQRage)

sum(small_outliers_age), sum(big_outliers_age)
```

Through this method, we found 15 outliers in the attribute 'absence' and 1 outlier in the attribute 'age'. We didn't discard them or modify them because as explained before we would be loosing valuable information for the interpretability and the conclusions. For the moment, we just acknowledge their existence.

3.1.3 Coding of noncontinuous or non-ordered variables

As it is better to work with numerical values, we decided that we would transform all categorical values into numerical ones. In order to do that, we applied the following function.

```
    #identify all categorical variables
    cat_columns = Data.select_dtypes(['object']).columns

    #convert all categorical variables to numeric
    Data[cat_columns] = Data[cat_columns].apply(lambda x: pd.factorize(x)[0])
```

By applying the factorize function, we coded all categorical values into their numerical counterparts, for example, we code binary categorical attributes of 'yes' and 'no' to binary values of *one* and *zero*.

3.1.4 Feature extraction

In order to expect to improve the future models' performance, we thought that we can derive a new variable. We decided to include some information from the different performance metrics 'G1' and 'G2', but trying to reduce the correlation that this new attribute could cause. We thought of the next following attribute:

```
    Data['performanceFailure'] = (Data['G1']*0.4 + Data['G2']*0.6) /
        (Data['failures']+1)
```

The multiplicative constants of this new attribute 'performanceFailure' try to take into account how recent the performance metric has been taken, by giving more weight to the latter grade. We divide this grades' weighting by the number of failures plus *one* (as we want to avoid dividing by *zero*), so it can be punished by the number of classes failed previously.

3.1.5 Feature selection

In order to improve model performance, we want to see which attributes could be dropped and which ones we want to keep, therefore we plotted the correlation matrix to decided that.

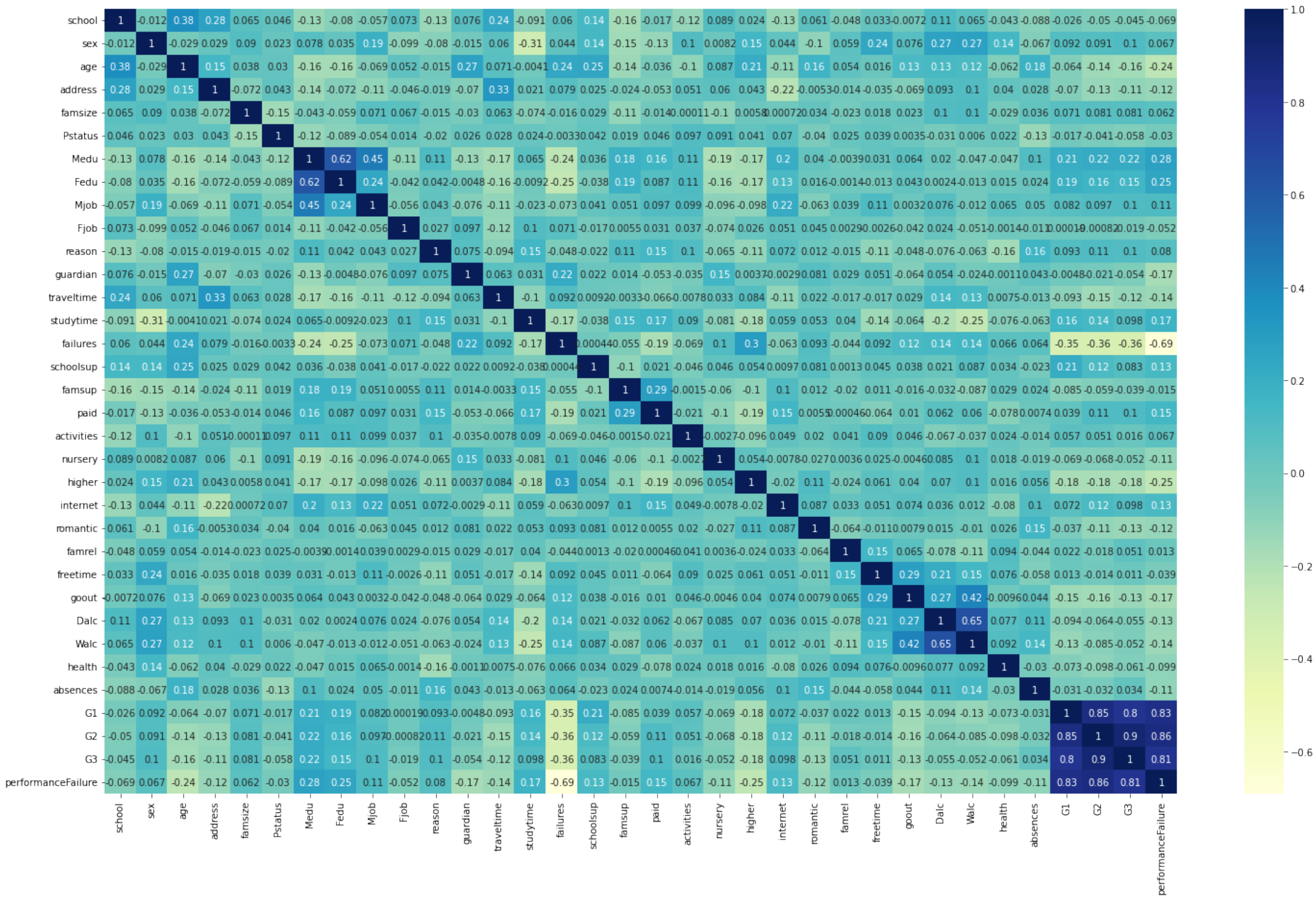


Figure 7: Correlation Matrix

We decided to drop G1' and 'G2' as they are very correlated to 'G3', which is our target variable to predict.

3.1.6 Normalization

The last step of our preprocess of the data is to normalize the data because some models are very sensitive to large values like outliers. To avoid this problem, we decided to do this step only to our attributes '`age`' and '`absences`' because those are the only real numeric attributes with outliers. We chose the *min-max scaling* transformation to normalize our data for those attributes.

```
    min_max_scaler = preprocessing.MinMaxScaler()
    newData[['Age_min_max', 'Absences_min_max']] =
        min_max_scaler.fit_transform(newData[['age', 'absences']])
```

3.2 Visualization

To visualize better our data, we used PCA (Principal component analysis) technique. This helps us to identify what is the best approach and which model is more suitable for the dataset.

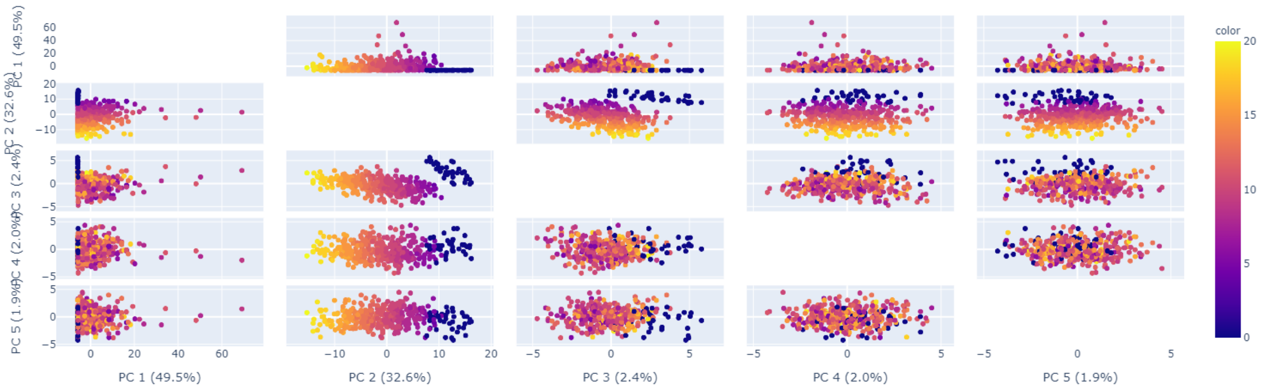


Figure 8: PCA visualization with 5 number of components (colored by '`G3`')

As we can see in the *Figure 8*, we can identify a few patterns and groups by setting the number of components to 5, although it is still hard to tell and extract some extra information.

Next, we plotted some graphs to see the relation between a specific attribute and the target variable '`G3`'. Here are some:

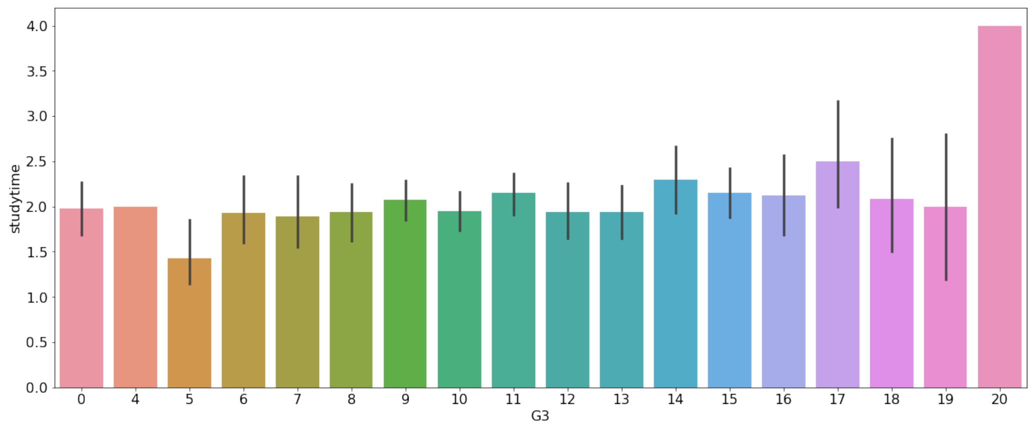


Figure 9: '`studytime`' a function of '`G3`'

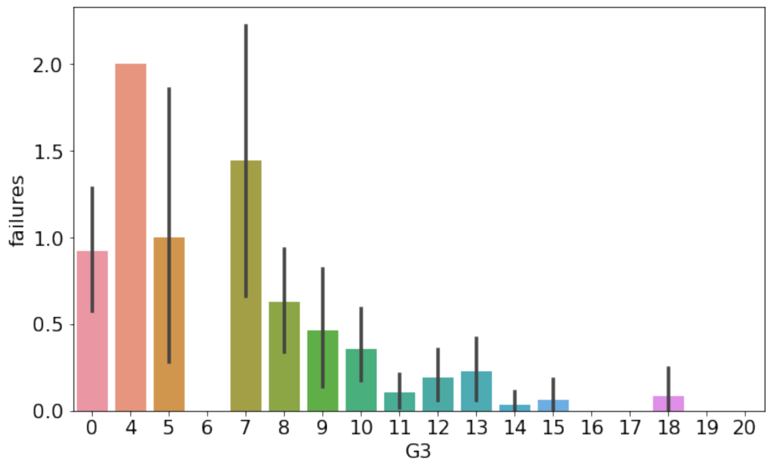


Figure 10: ‘`failures`’ a function of ‘`G3`’

Observing the *Figures 10*, we can immediately see a relation of ‘`failures`’ with the target variable ‘`G3`’. In this case, the students with the fewest class failures tend to have a higher final grade.

4 Resampling protocol

After experimenting with different `train_test_split()` proportions, we decided to split the data with *one-third* as test data and the rest as training data.

```
X = newData.loc[:,newData.columns != 'G3']
y = newData['G3']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33,
    random_state=42)
```

Regarding the resampling protocol, we will be using *k-fold cross validation* for all the regression models of this project. Since we have a dataset that is rather small, the value of k we picked is 5 in the cross validation partitions.

5 Regression models

In this section, for the modeling of the data of this project, we mainly used the library `scikit-learn` and other libraries like `Matplotlib` for plotting, `Numpy`, `Pandas`, etc. To test which models achieved better performances, at first we tested all of them with default hyper-parameters. Then, we evaluated them using the R^2 metric as the performance indicator.

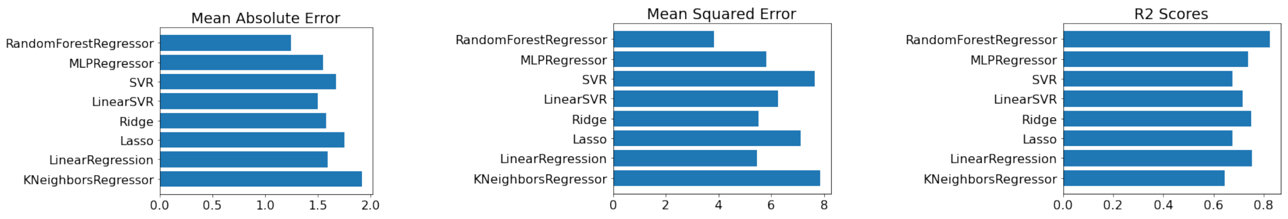


Figure 11: Performance metrics

5.1 Linear models

After looking at the results of the performance evaluation in *Figure* ??, we decided to select the `Ridge Regression`, the `k-nearest-neighbours` and lastly, the `Linear Regression` model. For each of them, we will be applying the `GridSearchCV()` method to find the best performing hyperparameters.

5.1.1 Linear Regression

This regression model is the simplest one that we will be using in this project. It attempts to fit a linear model with coefficients $w = (w_1, ..., w_p)$ to minimize the residual sum of squares between the observed targets in the data set, and the targets predicted by the linear approximation.

For this regression model we used `sklearn.linear_model.LinearRegression()`. It has a few hyperparameters, but in our case we will be working only with: `fit_intercept` which indicates whether to calculate the intercept for this model or not, and also `positive` which forces coefficients to be positive.

```
linRegParam = {   'positive' : ["True","False"],
                  'fit_intercept' : ["True","False"]}
```

Using `GridSearchCV()` we found out that the best hyperparameters for this model are:

fit_intercept	**positive**
True	True

After that, we set the model with these parameters found with the `GridSearchCV()`, trained it again, and performed an evaluation of the model using the R^2 score and using the test data.

Mean Absolute Error	**Explained Variance**	**R2 train**	**R2 test**
1.482	0.763	0.702	0.762

As we can see, our model performs better in the test data (with a score of 0.76) than the training data (with a score of 0.70). That means that the model has generalized pretty decently. Also, we have an explained variance of 0.76, which means that 76% of the proportion of the target variability of the dataset is explained by the model.

5.1.2 Ridge Regression

The `Ridge` model is a regression model where the loss function is the linear least squares function and the regularization is given by the i2-norm. To use this model we are going to be training the `sklearn.linear_model.Ridge()` model implemented in the `sklearn` module.

The parameters we are going to be exploring with the `GridSearchCV()` method are `alpha`, which corresponds to the regularization strength. We will also be tuning the `fit_intercept`, which is explained on the `LinearRegression` section. The `tol` parameter indicates the precision of the solution and finally the `solver` is the method to use in the computational routines.

```
ridgeParam = { 'alpha': [1, 0.1, 0.01, 0.001, 0.0001, 0],
               'fit_intercept' : ["True","False"],
               'tol' : [10**-5, 10**-4, 10**-3, 10**-2],
               'solver'  : ['auto', 'svd', 'cholesky', 'lsqr', 'sparse_cg', 'sag',
                   'saga']}
```

Using `GridSearchCV()` we found out that the best hyperparameters for this model are:

alpha	fit_intercept	solver	tol
1	True	saga	0.01

We find that the selected solver is `SAGA`, this solver is similar to `SAG` and other incremental gradient algorithms although `SAGA` improves with faster convergence rates and supports a wider variety of problems.

Mean Absolute Error	Explained Variance	R2 train	R2 test
1.581	0.744	0.691	0.744

As it was the case for the `LinearRegression` the performance is better with the test data. The train R^2 score is 0.6912 while with the test data is 0.7436.

5.1.3 K-Nearest Neighbors

The last linear model we will be using is k-nearest neighbors. This method is used in both classification and regression problems. In both cases, the input consists of the k closest training examples in a data set. And particularly in k-NN regression, the output is the property value for the object. This value is the average of the values of k nearest neighbors.

For this project, the library *sklearn* offers us a method that implements this algorithm called `sklearn.neighbors.KNeighborsRegressor()`. The target is predicted by local interpolation of the targets associated of the nearest neighbors in the training set.

As for the hyperparameters, we tune up the `algorithm` that will be used to compute the nearest neighbors, the distance `metric` that will be used for the tree, `n_neighbors` which indicates the number of neighbors used for the queries, and lastly, the `weights` or weight function used in the prediction.

```
KNNParam = {  'n_neighbors' : range(3,52,3),
              'algorithm' : ['auto', 'ball_tree', 'kd_tree', 'brute'],
              'weights' : ["uniform","distance"],
              'metric' : ['euclidean','manhattan','minkowski']}
```

After running the `GridSearchCV()`, we found out that the best hyperparameters for this model are: the best algorithm to compute the nearest neighbors is `ball_tree`, the distance metric that will be used is the `euclidean`, the best number of neighbors is 9, and the weight function is `uniform` which means that all points in each neighborhood are weighted equally.

algorithm	metric	n_neighbor	weights
ball_tree	euclidean	9	uniform

In order to evaluate the model, we set up this model with the best hyperparameters and run the prediction with the test data. As you can see in the table below, the R^2 score for the training is lower than the R^2 score of the test, which means that the algorithm has generalized and works decently feeding it with new data.

Mean Absolute Error	Explained Variance	R2 train	R2 test
1.746	0.692	0.647	0.691

5.2 Non-linear models

In this case, we chose `MLP Regression` and `RandomForest Regression` models. These showed better metrics results in the multiple model default test done initially at the section 5.

5.2.1 MLP Regressor

The `MPL Regressor` is based on a multiple Preceptor regression model, which is to say it's a neural network. It optimizes the squared error using *LBFGS* or the stochastic gradient descent. We will be using the `sklearn.neural_network.MLPRegressor()` which allows us to tune a wide range of hyperparameters.

We will be exploring with `GridSearchCV()` the following hyperparameters. First, the `hidden_layer_sizes` indicates the number of neurons that compose the ith hidden layer. The `activation` parameter refers to the activation function for the hidden layer, the `solver` indicates the method to use for weight optimization. The `alpha` is the L2 penalty or regularization term, and finally, the `learning_reate` refers to the learning rate schedule for weight updates.

```
MLPParam = {        'hidden_layer_sizes': [(10,30,10),(20,)],
                    'activation': ['tanh', 'relu'],
                    'solver': ['sgd', 'adam'],
                    'alpha': [0.0001, 0.05],
                    'learning_rate': ['constant','adaptive']}
```

Using `GridSearchCV()` we found out that the best hyperparameters for this model are:

activation	**alpha**	**hidden_layer_sizes**	**learning_rate**	**solver**
relu	0.05	(20,)	constant	sgd

With the tuned hyperparameters, we obtain the following metrics.

Mean Absolute Error	**Explained Variance**	**R2 Train**	**R2 Test**
1.601	0.735	0.696	0.735

Measuring the performance with the test data also improves the R^2 score. We obtain 0.7351 for the test data and 0.6956 for the training data.

5.2.2 RandomForest Regressor

The last non-linear model that we will be experimenting with, is the *Random Forest Regressor*. A random forest is a meta estimator that fits a number of classifying decision trees on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting.

There is a method called `sklearn.ensemble.RandomForestRegressor()` in the `sklearn` library that implements this algorithm and has many parameters. For this project, we will be tweaking the hyperparameters: `n_estimators` which indicates the number of tress in the forest, `criterion` which is the function to measure the quality of a split, and lastly, `max_depth` which indicates the maximum depth of the tree.

```
RForestParam = {        'n_estimators': [100, 150, 200, 250, 300],
                        'criterion'  : ['squared_error', 'absolute_error', 'poisson'],
                        'max_depth': [3,5,9,11,15]}
```

After running the `GridSearchCV()` to tune up the model to its best performing hyperparameters, we found that the best are:

n_estimators	criterion	max_depth
100	`absolute_error`	9

After setting the tuned model with the best hyperparameters, we used the test data to predict the results and obtained the following metrics:

Mean Absolute Error	Explained Variance	R2 train	R2 test
1.228	0.835	0.812	0.834

As we can see in the table above, the obtained a R^2 score of 0.81 for the training data and an even better R^2 score of 0.83 for the test data, which means that the model has a good generalization and works well with new data.

6 Final model

From all the different models, we chose the `RandomForest` model. It performed considerably better than the other models.

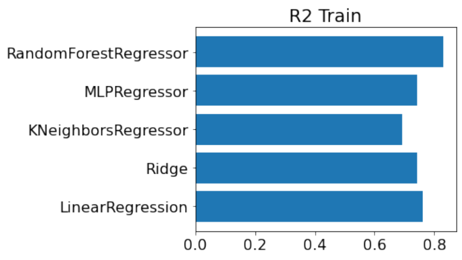


Figure 12: R2 Train

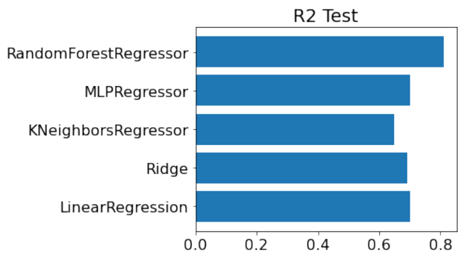


Figure 13: R2 Test

As we can see in *Figure 12* and *Figure 13* the data fits well with the `RandomForest` model. The **R^2 Squared** metric determines the proportion of variance in the dependent variable that can be explained by the independent variable. In other words, R^2 *Squared* shows how well the data fit the regression model (the goodness of fit).

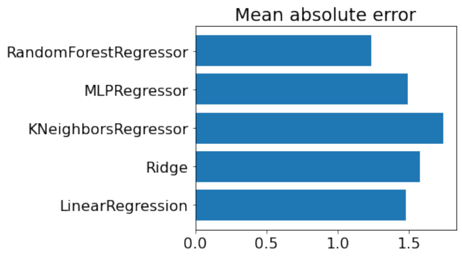


Figure 14: Mean Absolute Error (MAE)

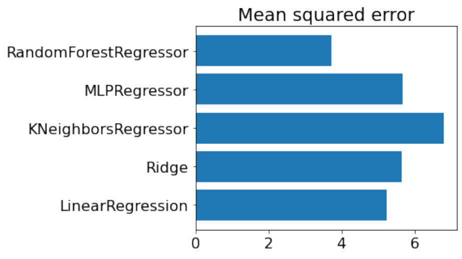


Figure 15: Mean Squared Error (MSE)

We also observe that `RandomForest` has a lower **Mean Absolute Error** (MAE) and **Mean Squared Error** (MSE) in comparison with other models (in *Figures 14* and *15*). The *Mean Absolute Error* reflects the difference between the predicted results and the ones found in test data, so it shows us how well the predictions of our model are. Similarly, the *Mean Squared Error* measures the squared average of the differences between predicted values and actual values (test data).

RandomForest is based on ensemble learning, which means multiple ML algorithms are generated and combined to solve the specified problem. RandomForest uses the bagging technique, which trains the different ML algorithms with random subsets of the train data. It then applies bootstrapping, which parallelizes all calculations and basically boils down to incrementally train instances with results of other models executions. After that, results are aggregated and combined to form the resulting ensemble model.

The name "*Random Forest*" comes from the Bagging idea of data randomization (Random) and building multiple Decision Trees (Forest). The resulting decision tree after fitting the model is the following:

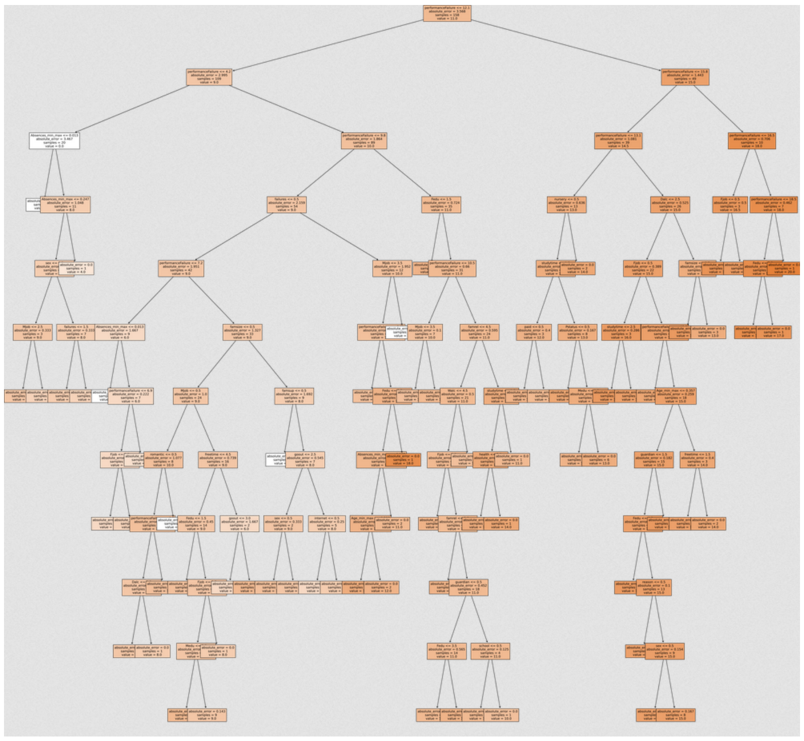


Figure 16: Decision Tree of the model

To have a closer look at the hyperparameter influence on the models' performance, we plotted the search results for the `GridSearchCV()`.

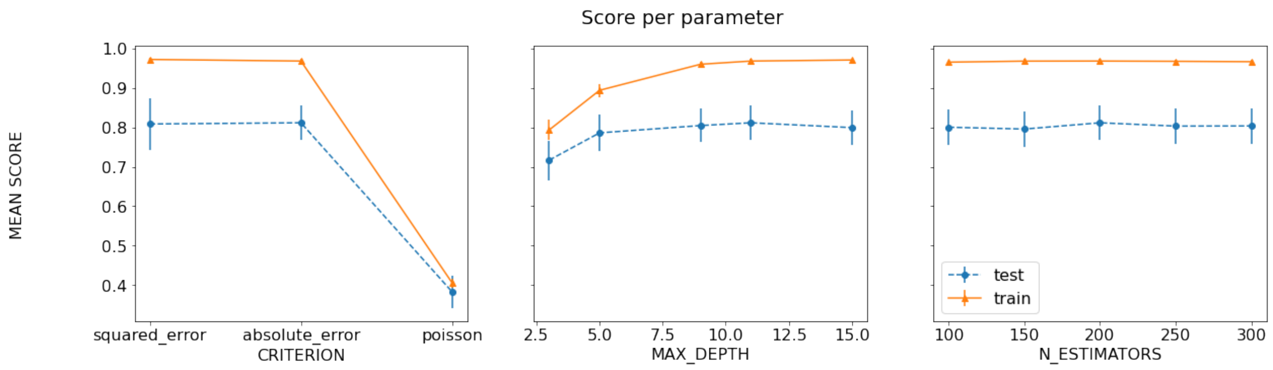


Figure 17: `GridSearchCV()` score per parameter

Lastly, we plotted the importance of the features in the model's performance according to `GridSearchCV()`.

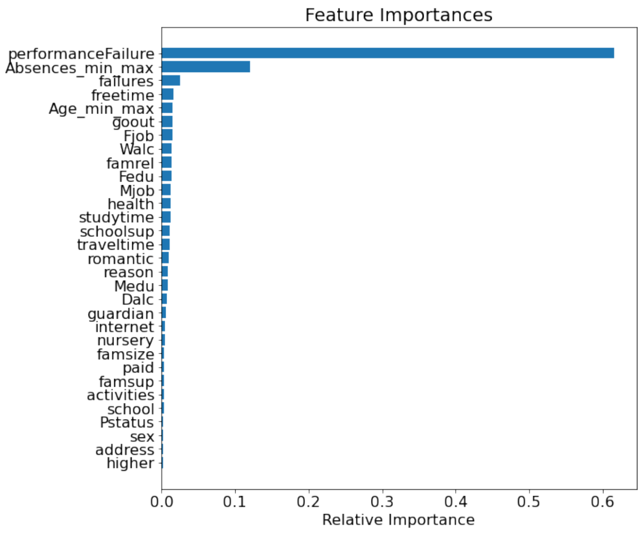


Figure 18: Features importance according to `GridSearch()`

We find that the attribute we introduced, `performaceFailure` has a really high importance, this is due to the fact that it's a mixture of the old performance metrics `G1` & `G2` and as these were really correlated to `G3`, we can still observe their influence on the added attribute.

One last inspection is to take a look at the coefficients (weights) of our model. Random forest is an ensemble of decision trees, it is not a linear model. Therefore, the sum of all coefficients adds up to 1.

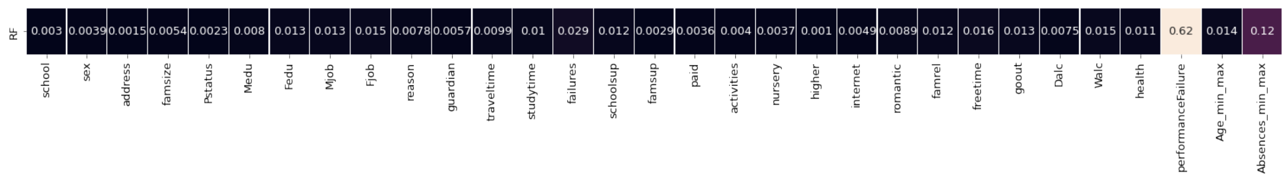


Figure 19: Weights of our final model (*Random Forest Regressor*)

As you can see in *Figure 20*, the most important feature is `performanceFailure` (which we can also confirm by looking at *Figure 18*). The second most important feature is `Absences_min_max` which is also true. Finally, we can compare our best tuned model weights to, for example, the best linear model weights, the Linear Regression model.

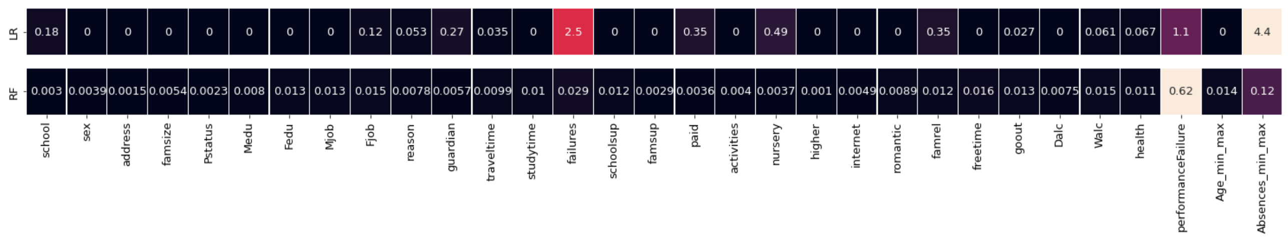


Figure 20: Weight comparison between *Random Forest Regressor* and *Linear Regression*

In the Linear Regression model, we observe that `Absences_min_max` is the most important feature, followed by `failures` which is different from our Random Forest Regressor model.

7 Conclusions

7.1 Self-assessment

For the development of this project, we closely followed the guidelines shown in the guideline document. By doing this, we were able to properly organize our work schedule. It also helped us create the main structure of the document and its sections.

We found that we had a general good notion of most of the concepts, but not as in depth as the project demanded. And so by doing the project, by checking information online or re-visiting the laboratory notebooks, we were able to learn details to machine learning subjects that we vaguely detail about.

We think that after doing the project we have solidified most of the knowledge that we have seen on the course, now we are able to take on regression and classification problems with ease and generally understand most technical terms and concepts in machine learning.

7.2 Scientific and personal conclusions

From the scientific perspective, we believe that these models are really useful and powerful. It would not surprise us at all that with a better preprocess and a finer granularity of the tuning of hyperparameters, the model could reach 99% of accuracy in the predictions.

On the other hand, from our personal perspective, we would like to know more about this field of **Machine Learning**. We loved this project, but we have not been able to spend the time that we would like to because of other assignments projects and exams.

7.3 Possible extensions and known limitations

The main problem we had with the project was that our data wasn't adequate for all the different models, even after pre-processing them accordingly. We observed that the non-lineal `SVR` and `MLP` models didn't converge easily. They usually reached the set maximum iterations.

As we have found, predicting students performance with these attributes and regression models is quite difficult. A possible extension to solve or ease this problem could be using classification models to predict if the students marks fall into specific ranges.

Bibliography

[CS08] Paulo Cortez and Alice Silva. "Using data mining to predict secondary school student performance". In: *EUROSIS* (Jan. 2008).

[car] carloscliment. *Random forest regressor and gridsearch*. URL: https://www.kaggle.com/carloscliment/random-forest-regressor-and-gridsearch.

[cnv] cnvrg. *RandomForest Regression*. URL: https://cnvrg.io/random-forest-regression/.

[hol] holypython. *K-NearestNeighbor parameter optimization*. URL: https://holypython.com/knn/k-nearest-neighbor-optimization-parameters/.

[maca] machinelearninghd. *GridSearchCV hyperparameter tunning*. URL: https://machinelearninghd.com/gridsearchcv-hyperparameter-tuning-sckit-learn-regression-classification/.

[macb] machinelearninghd. *Sklearn metrics for machine learning*. URL: https://machinelearninghd.com/sklearn-metrics-classification-regression/.

[meda] medium. *Random forest regression*. URL: https://medium.datadriveninvestor.com/random-forest-regression-9871bc9a25eb?gi=73d23d042398.

[medb] medium. *What is cross validation*. URL: https://medium.com/the-rise-of-unbelievable/what-is-cross-validation-and-when-to-use-which-cross-validation-327d25bbb3f3.

[ove] overleaf. *Learn basics*. URL: https://www.overleaf.com/learn.

[ram] ramontanoeiro. *Student performance notebook*. URL: https://www.kaggle.com/ramontanoeiro/student-performance/notebook.

[scia] scikit-learn. *Ensemble Models*. URL: https://scikit-learn.org/stable/modules/classes.html#module-sklearn.ensemble.

[scib] scikit-learn. *GridSearchCV*. URL: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html.

[scic] scikit-learn. *KFold*. URL: https://scikit-learn.org/0.17/modules/generated/sklearn.cross_validation.KFold.html.

[scid] scikit-learn. *KNeighborsRegressor*. URL: https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsRegressor.html.

[scie] scikit-learn. *Linear Models*. URL: https://scikit-learn.org/stable/modules/classes.html#module-sklearn.linear_model.

[scif] scikit-learn. *Neural Network Models*. URL: https://scikit-learn.org/stable/modules/classes.html#module-sklearn.neural_network.

[scig] scikit-learn. *Plot SVM Regression*. URL: https://scikit-learn.org/stable/auto_examples/svm/plot_svm_regression.html.

[scih] scikit-learn. *Preprocessing*. URL: https://scikit-learn.org/stable/modules/preprocessing.html.

[staa] stackexchange. *Grid Search with MLPRregressor*. URL: https://datascience.stackexchange.com/questions/52348/gridsearchcv-with-mlpregressor-with-scikit-learn.

[stab] statology. *Learn basics*. URL: https://www.statology.org/ridge-regression/.

[towa] towardsdatascience. *Feature importance on a RandomForest*. URL: https://towardsdatascience.com/explaining-feature-importance-by-example-of-a-random-forest-d9166011959e.

[towb] towardsdatascience. *Feature importance on a RandomForest*. URL: https://towardsdatascience.com/explaining-feature-importance-by-example-of-a-random-forest-d9166011959e.

[towc] towardsdatascience. *Hyparameter tuning for RandomForest.* URL: https://towardsdatascience.com/hyperparameter-tuning-the-random-forest-in-python-using-scikit-learn-28d2aa77dd74?gi=2cad90f76ffb.

[vid] vidhya. *Tester guide for machine learning.* URL: https://medium.com/analytics-vidhya/testers-guide-for-testing-machine-learning-models-e7e5cea81264.

[wil] willkoehrsen. *Bayesian linear regression in python using machine learning to predict student grades.* URL: %5Ctexttt%7Bhttps://willkoehrsen.github.io/bayesian/modeling/project/bayesian-linear-regression-in-python-using-machine-learning-to-predict-student-grades-part-1/%7D.