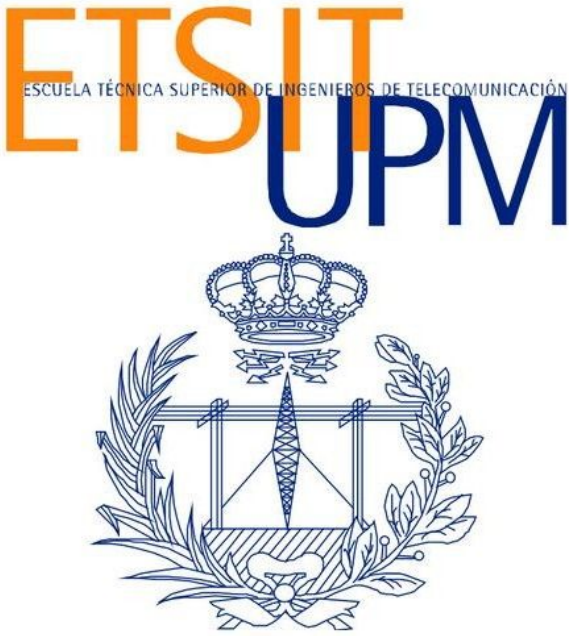
Machine Learning Laboratory

Half-term Report



Sergio Pérez Morillo

5/11/2019

Table of Contents

1. Methodology

1.1. Guidelines

The main inspiration source for this project is *"Hands-On Machine Learning with Scikit-Learn & Tensorflow"* by Aurélien Géron **[1]**. My methodology is based on the one provided in this book. From his point of view, machine learning projects feature eight steps:

1. **Looking at the big picture:** Define the objectives, check current solutions, frame the problem, define performance metrics, compare with similar problems, list and verify assumptions.
2. **Getting the data:** List the necessary data, find data sources, check legal issues, get and transform the data to enhance their manipulation and check data types.
3. **Exploring the data:** Sample the data if necessary, create a notebook, study data information from different perspectives, visualize the data, study correlations, study potential data transformation and document everything.
4. **Preparing the data:** Write transformation functions, clean the data, perform feature selection, feature engineering, and feature scaling.
5. **Selecting the model and training it:** Automate the process as much as possible, train many models as possible from different categories with pre-set parameters, measure their performance with cross-validation, analyze the errors, check feature engineering and selection and list the most promising ones.
6. **Fine-tuning your model:** Try different hyperparameters, grid search for few hyperparameters and random search for many hyperparameters, test the best model with a test dataset to check the generalization error.
7. **Presenting your solution:** Document every step, explain and present the solution, give details, use beautiful visualizations for better communication.
8. **Launching, monitoring, and maintaining your system:** Prepare the solution for production, automate the process, retrain the model with new data.

1.2. Case Study: Obstructive Sleep Apnea

His procedure clearly aims for real business solutions. For the OSA case study, this methodology is reshaped to fit its goal as follows:

1. **Framing the problem**: Description of the problem and study of specific knowledge on obstructive sleep apnea.
2. **Wrangling the data**: Quick data overview to check meaningful information and asses their usability based mostly on specific knowledge. The final outcome is an easier dataset to manipulate for the next steps.
3. **Exploratory Data Analysis:** Comprehensive data exploration focus on multiple visualizations to gain insights from correlations, distributions, and statistics.
4. **Preparing the data:** Data transformations, feature selection techniques, and feature scaling to the dataset before fitting the model. A feature selection analysis was developed before applying these techniques to the models.
5. **Model testing and fine-tuning**: Automation of model training from a list of models based mostly on the ones proposed in the book that includes a large variety of

algorithms and fine-tuning them with cross-validation and grid search using few hyperparameters to avoid exponentially increasing the complexity of the goal. Aside from that, the results are stored in a database for the next step.

6. **Results and model comparison**: Comprehensive analysis along with visualizations for determining the best models for both regression and classifications based on aspects such as accuracy, time and dimensionality.

1.3. Architecture and Tools

The hardware architecture utilized to test this methodology consists of a laptop that includes an Intel Core i7-8750H CPU, a 16GB DDR4 RAM, and Kubuntu as the operating system. I discarded the Google Colab option as I already had a software architecture deployed in this laptop that I use in my research lab. Moreover, this hardware architecture suffices for small and medium-size datasets.

The software architecture consists of several Jupyter notebooks and an Anaconda's virtual environment that features several Python libraries for each different task:

1. **Data manipulation:** numpy and pandas.
2. **Visualization**: matplotlib, seaborn, plotly, scikit-learn's sub-module manifold (t-SNE) and missingno.
3. **Data transformation**: scikit-learn's sub-modules model_selection, feature_selection, preprocessing and decomposition.
4. **Machine Learning Modeling**: xgboost and scikit-learn's sub-modules linear_model, naive_bayes, neighbors, tree, ensemble, SVM, neural_network.
5. **Metrics for model assessment**: scikit-learn's sub-module metrics.
6. **Other tasks**: IPython, os, time, datetime, math and pprint.

In the next chapters, I describe thoroughly the relevance of each library and its functions. I have decided to use Python since I have been working with this language for over two years in different areas such as data science (the main one), system administration and IoT.

2. Data Wrangling

The first part of the project is to clean the dataset. For doing so, I have used just pandas, numpy and missingno. Other tools such as *pandas profiling* were considered to perform a preliminary data analysis but were ultimately discarded due to the dataset size, I thought they were not necessary for such a small data sample.

2.1. Global Procedure

First, I imported the dataset from the excel file and transformed it into a pandas' dataframe:

```
file = 'Info_BDApnea_QuironMalaga_rev1.xlsx'
xl = pd.ExcelFile('Info_BDApnea_QuironMalaga_rev1.xlsx')
df_tmp = xl.parse('Hoja1')
```

Then, I used the pandas dataframe functions ***describe***, ***dtypes*** and ***head*** for getting a glimpse of the dataset features and their usability. An example of the functions' output:

	Patient	Comentarios	Audios tumbado	Fotos	Audio fs KHz	Gender
0	P0001	es el Patient0002 (fotos) 3 (sentado) y 4 (tum...	si	si	16	hombre

	IAH	Peso	Talla
count	649.000000	676.000000	677.000000
mean	20.364653	87.665680	171.144756
std	18.692784	18.542861	11.661385

Figure 1: Pandas' *describe* and *head* functions output.

According to the teaches, '-1' values correspond to *NaNs*, so I transform those using pandas' *replace* function and then I displayed the *NaNs* to asses them better using the python library missingno and its function ***matrix***.

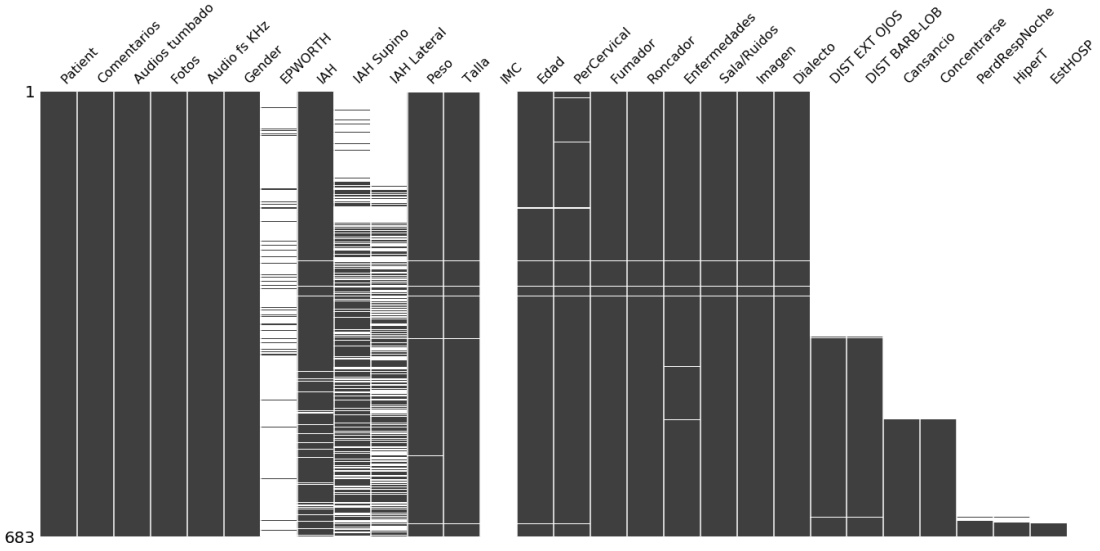


Figure 2: Missingno's *matrix* function output.

After dropping useless features due to their number of *NaNs* or overall usability in machine learning models, I checked the remaining *NaNs* by column and row:

```
df_tmp1.isnull().sum(axis=0) #column
df_tmp1.isnull().sum(axis=1).sort_values(ascending=False).head(15) #row
```

As explained in the PRDL report, I dropped the NaNs in features *IAH*, *PerCervical, Peso* and substituted them in feature *Enfermedades* as follows:

```
df_ = df_[np.isfinite(df_['PerCervical'])] #same for IAH and Peso
df_['Enfermedades'].fillna('ns', inplace=True)
```

Then, I checked the categorical features and their values using the pandas' function ***value_counts*** to decide if I transform, drop or leave them intact. For the former, I used the pandas' series function ***replace*** and ***apply*** to reduce the total number of categories.

Finally, I applied feature engineering by creating a new one, the body mass index (BMI) using the features *Weight* and *Height*. For better manipulation, I also rename (using pandas' ***rename***) and reorder the columns. The next figure shows the final dataframe:

Patient	Gender	Age	IAH	Cervical	Weight	Height	BMI	Smoker	Snorer	Illness
P0002	hombre	56.0	29.6	48.0	119.0	174.0	39.31	si	ns	ns
P0004	hombre	39.0	19.7	42.0	78.0	168.0	27.64	no	ns	si
P0005	hombre	32.0	9.0	40.0	80.0	173.0	26.73	no	ns	si
P0006	hombre	32.0	2.0	42.0	109.0	190.0	30.19	no	ns	si
P0007	hombre	39.0	34.0	42.0	86.0	169.0	30.11	no	ns	si

Figure3: Final dataframe version.

I preferred to store it as a CSV file instead of an Excel file since I am used to working with the former:

```
df_final.to_csv(r'OSA_propio.csv', index=None, header=True)
```

2.2. Extra Procedure for Classification Modeling

So far, the processing is applied to both regression and classification. In addition to that, I performed further data augmentation as suggested by the teachers by combining two datasets. Aside from that, I created a binary feature called *OSA* to use it as the target feature as follows:

```
def foo(x):
    if x<=10.0:
```

```
        return "Healthy"
    elif x>=30.0:
        return "Severe"
df = df.loc[(df["IAH"]<=10.0) | (df["IAH"]>=30.0)]
df["OSA"] = df["IAH"].apply(foo)
```

On the other hand, I merged the two datasets to perform the data augmentation as follows:

```
df_OSA_inner = pd.merge(df, df_OSA_speech, on='Patient', how='inner')
```

The parameter *df* is the processed dataframe from the previous section and *df_OSA_speech* is a dataframe that contains extra audio features from male patients. I decided to perform an inner merge to avoid creating more *NaNs* values. Finally, I used again ***vales_counts*** to check that the binary feature was balanced and stored the resulting dataframe into a CSV file as in the previous section.

3. Exploratory Data Analysis

The next step is to perform a data analysis to gain insights from the different features in our processed dataset. For doing so, I relied mainly on visualizations of scatters, distributions and correlations from categorical features, numerical features, and the combination of both.

At my research lab, I usually utilize plotly or matplotlib for this task. This time, I seized the opportunity of learning a new library, hence I chose seaborn as my primary option. Seaborn is really simple to use and provides beautiful graphs in just a few lines of code.

3.1. Categorical Features

My first step was to set the feature *Patient* as the index using the pandas' function ***set_index*** and cast the object features to categorical by using the pandas' function ***astype***. Then, I started with the analysis of the categorical feature. The first set of graphs were bar plots due to their simplicity. I wrote a simple loop to show them all at once:

```
for col in df.select_dtypes(include='category').columns:
        fig = sns.catplot(x=col, kind="count", data=df, palette="Set3")
        plt.show()
```

The following figure shows an example of one of the resulting plots:

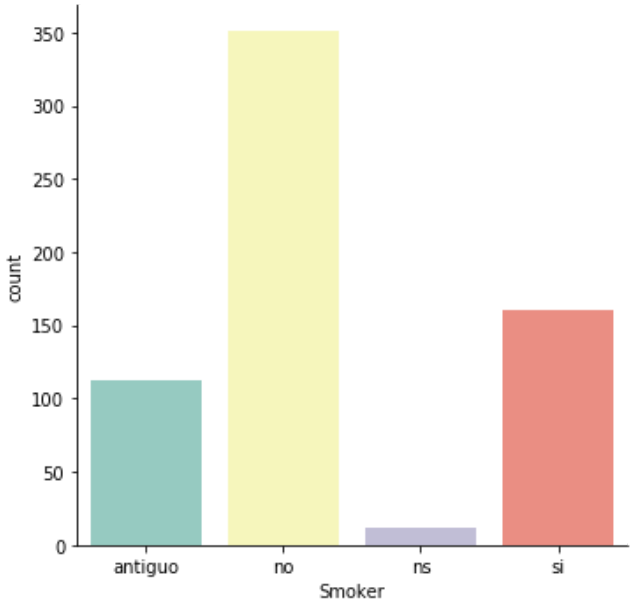


Figure 4: Bar plot from the Smoker feature.

3.2. Numerical Features

On the other hand, there was a data analysis for the numerical features. First, I used the pandas' function ***describe*** again to check simple statistics such as mean, standard deviation, minimum values, maximum values, and quartiles to look for outliers and odd values. Then, I plotted their single and combined distributions in just one plot as follows:

```
sns.set(style="ticks")
sns.pairplot(df,palette="Set3")
```

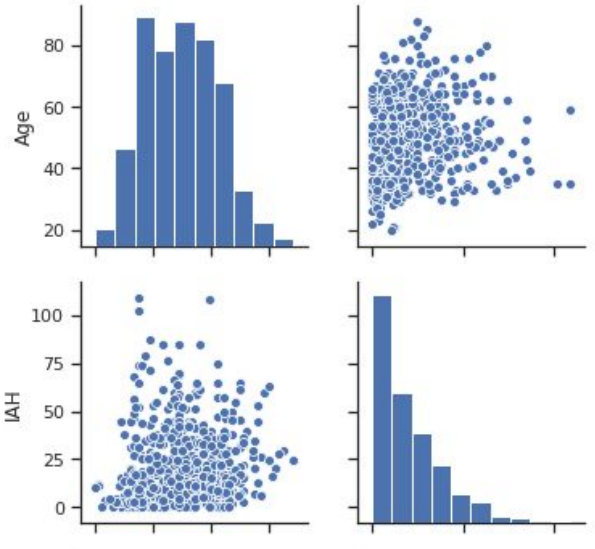


Figure 5: Example of the correlation and distribution of two features.

Then, I plotted a heatmap to check numerically the correlation between features. It is worth noting that there is a compatibility problem between seaborn and matplotlib newer versions that results in heatmaps not showing correctly. To solve that, I add the following line of codes:

```
hm = sns.heatmap(df.corr(), cmap="Blues", annot=True)
bottom, top = hm.get_ylim()
hm.set_ylim(bottom + 0.5, top - 0.5)
plt.show()
```

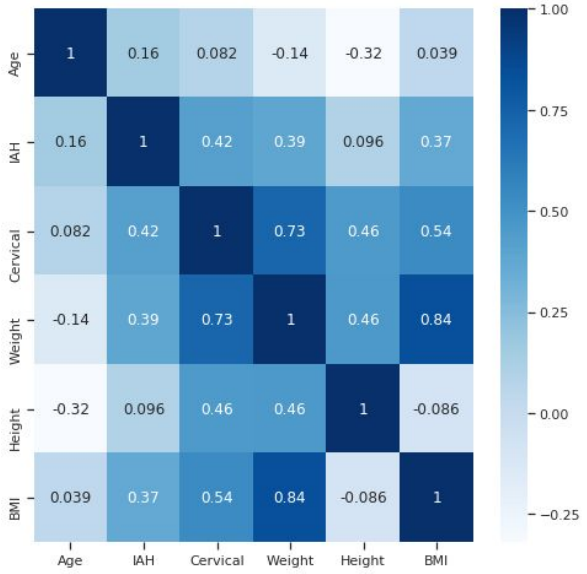


Figure 6: Correlation heatmap of the numerical features found in the dataset.

3.3. Combining Features

Finally, I combined both categorical and numerical features to better understand them and thus draw more insightful conclusions. To do so, I plotted the boxplot of each numerical feature as a function of the *Gender*.

```
for col in df.select_dtypes(include='float64').columns:
    sns.boxplot(x="Gender", y=col, data=df, palette="Set3")
    plt.show()
```

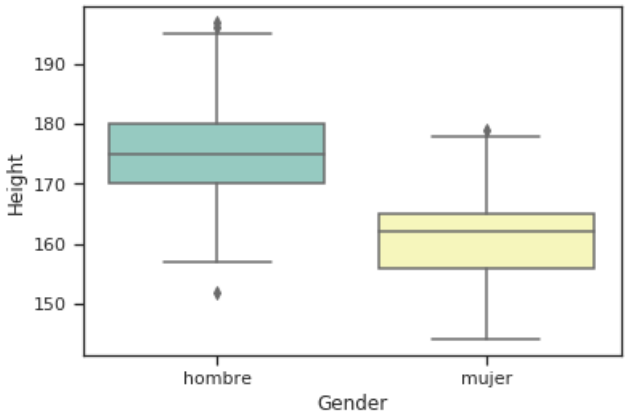


Figure 7: Boxplot example of the feature *height* by *gender*.

Then I plotted boxplots of the target feature *IAH* as a function of each categorical feature:

```
for col in df.select_dtypes(include='category').columns:
    sns.boxplot(x=col, y="IAH", data=df, palette="Set3")
    plt.show()
```

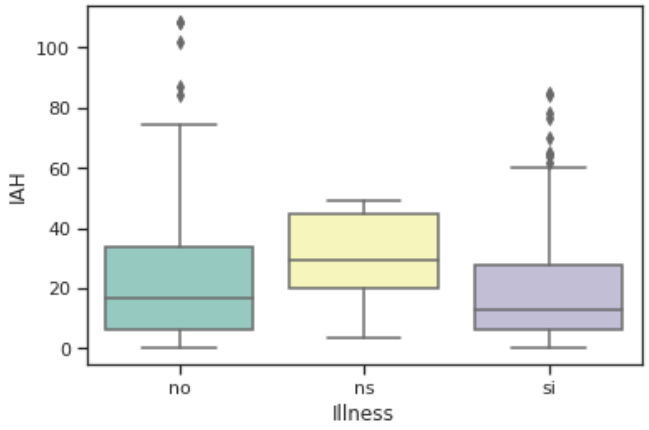


Figure 8: Boxplot example of the target feature *IAH* by *Illness*.

An alternative to boxplot is violin plots. They can combine up to three features in the same graph. They are useful for relating categorical features using a numerical one. Here's an example of the target feature *IAH* by *Gender* and *Illness* and the code snippet:

```
for col in df.select_dtypes(include='float64').columns:
```

```
    sns.violinplot(x="Illness", y=col, hue="Gender",
                   split=True, inner="quart", data=df,
                   palette="Set2")
    plt.show()
```

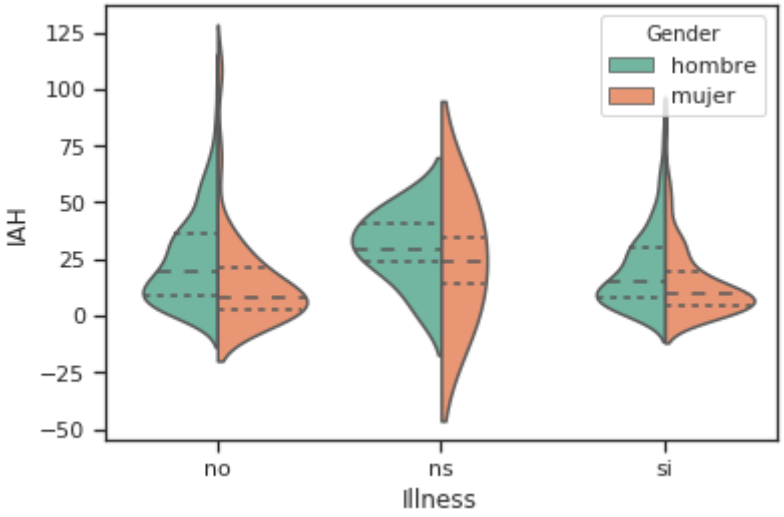


Figure 9: Violin example of the target feature *IAH* by *Illness* and *Gender*.

The last plots of this EDA are pair plots like the numerical ones that show the individual and combined distribution between features, but this time they are colored by categorical features:

```
sns.pairplot(df, hue="Gender", palette="Set2")
```

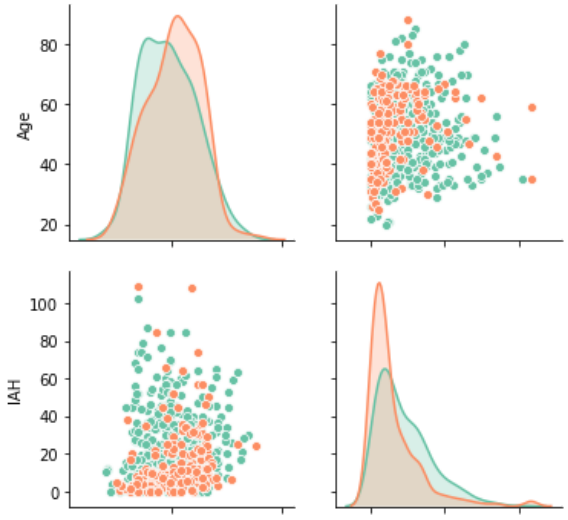


Figure 10: Example of the correlation and distribution of *IAH* and *Age* by *Gender*.

Other plots were considered such as strip plots, *FacetGrids*, KDE plots, and joint plots but were discarded due to time constraints and their usefulness for drawing additional insights in this small dataset. To check all the plots of my EDA, visit on my GitHub account the notebook related to this analysis **[2]**.

4. Data Preparation

In this section, I describe data transformations, feature scaling methods, feature scaling methods, feature selection approaches, and dimensionality reduction techniques. These are applied to the dataset before fitting the models. This part of the machine learning workflow is usually called pre-processing.

4.1. Data Transformation

Starting with the transformations, I developed two of them: *log(x+1)* and *polynomial features*. The former is regarded as one of the most popular transformations due to its simplicity and usefulness. The latter is used for example as a kernel in support vector machines or linear regression (to make them polynomial). The implementation was performed using numpy and scikit-learn as follows:

```
#Polynomial transformation
poly_features = PolynomialFeatures(degree=poly_deg, include_bias=False)
X = poly_features.fit_transform(X)
#log(x+1) transformation
X, y = np.log10(X+1), np.log10(y+1)
y = 10**y-1 #inverse transform for the results
```

Other transformations were considered such as X^2 and *ln(x)* but were discarded to avoid exponentially increasing the total number of combinations.

4.2. Feature Scaling

For feature scaling methods, I used two of the most popular ones: minimum-maximum scaling and standard scaling. As explained in the PRDL report, they have different goals, although those goals overlap sometimes. The implementation is similar for both of them:

```
sc_x, sc_y = MinMaxScaler(), MinMaxScaler() #or StandardScaler()
X = sc_x.fit_transform(X)
y = sc_y.fit_transform(y) #inverse transform for the results
```

From experience working in my research lab, I always use two scalers one for the *X* split and one for the *y* split to make it handier. Note that the sklearn's function ***MinMaxScaler*** uses the range (0,1) unless the user specifies otherwise. There are other feature scaling methods but these two suffice for this case study.

4.3. Dimensionality Reduction

For dimensionality reduction, I considered two branches of algorithms: *principal component analysis* (PCA) and t-SNE (a *manifold* sub-branch). For the former, I first analyzed the cumulative variance plotting it for a PCA of 50 components using scikit-learn for data scaling

and the PCA transformation and seaborn for the graph. The most important parts of the implementation are as follows:

```
pca = PCA(50)
Xnormalized_pca = pca.fit_transform(Xnormalized)
ax = sns.lineplot(x=list(range(50)),
                  y=np.cumsum(pca.explained_variance_ratio_))
```

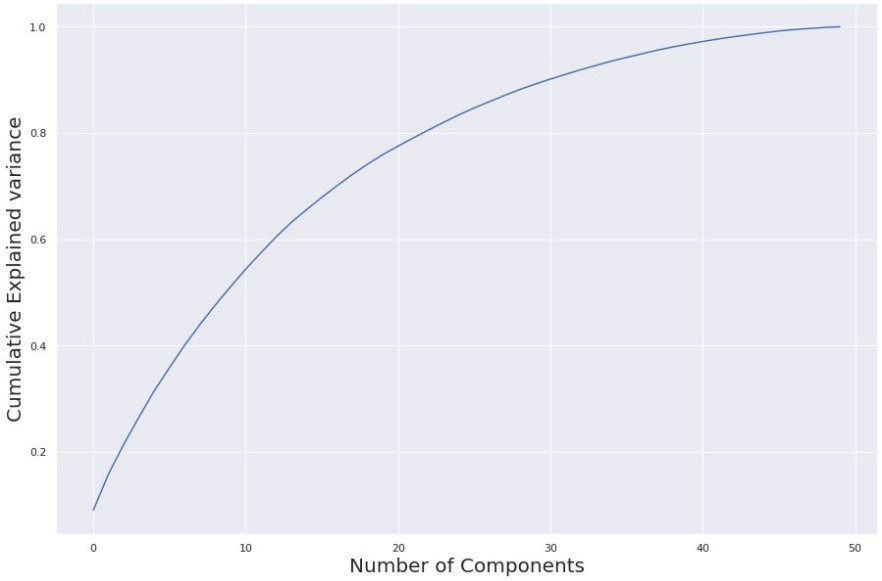


Figure 11: Cumulative explained variance as a function of PCA components.

Following this analysis, I applied a PCA of two components to the classification dataframe to check the distribution of the binary target *OSA*.

```
sns.scatterplot(x="pca-one", y="pca-two", hue="OSA",data=df)
```

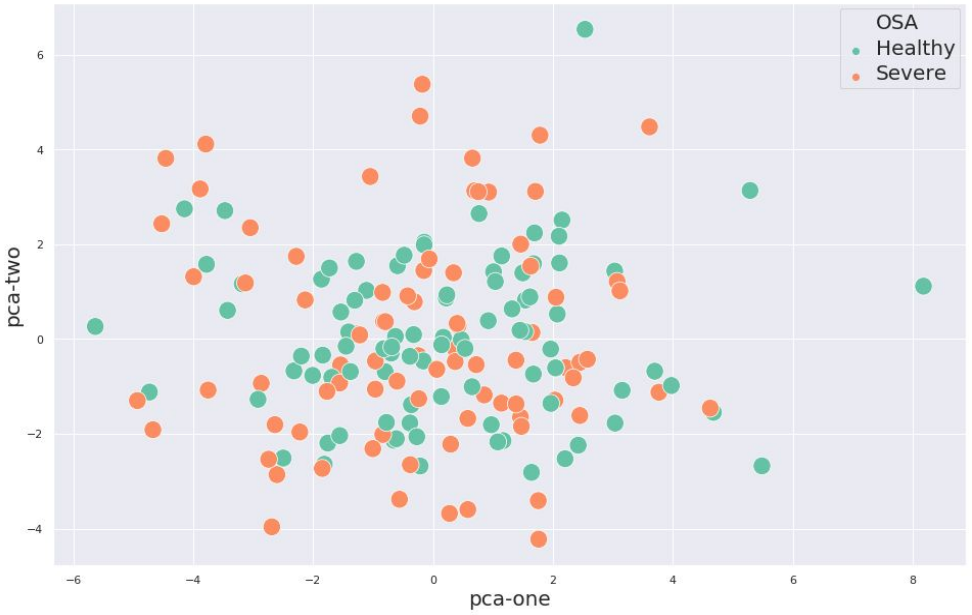


Figure 12: Scatter plot of the two PCA principal components.

For a better visualization in real-time, I also applied a PCA of three components to the classification dataframe and plot it with plotly that allows exploring it as it is a dynamic graph.

```
import plotly.express as px
f = px.scatter_3d(df,x='pca-one',y='pca-two',z='pca-three',color='OSA')
f.show()
```

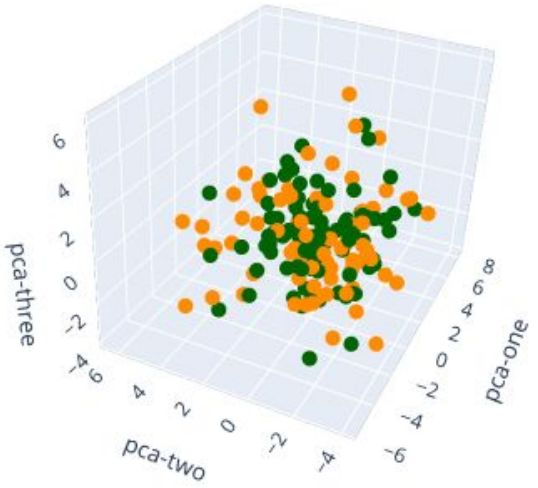


Figure 13: Snapshot of a 3D Scatter plot of the three PCA principal components.

The final implementation for the regression and classification modeling is the following. Note that instead of choosing the number of PCA components I set this number as the minimum cumulative variance required (in this case 80%).

```
pca = PCA(0.8)
X = pca.fit_transform(X) #always normalizing it first
```

Then, I applied t-SNE to the dataframe similar to the PCA case but the results were not relevant. This algorithm requires a bigger dataset. The implementation is as follows:

```
tsne = TSNE(n_components=2, verbose=1, perplexity=40, n_iter=300)
X_tsne = tsne.fit_transform(Xnormalized[:,:40]) #only continuous values
```

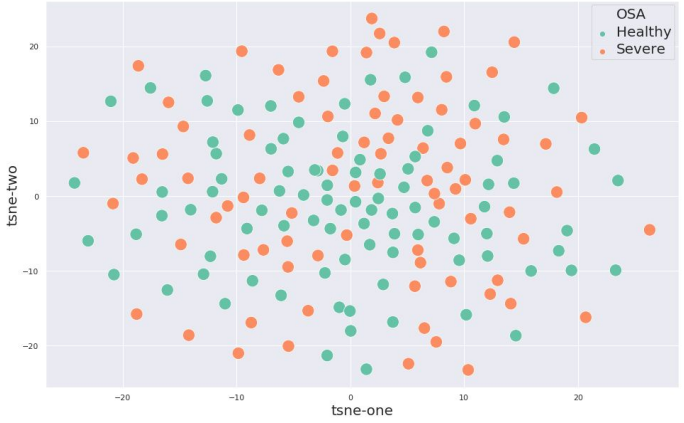


Figure 14: Scatter plot of the two t-SNE components.

4.4. Feature Selection

As I had not applied before feature selection, I decided to perform a comprehensive analysis of several options before implementing it in the modeling workflow. Note that this was only used for the classification dataset which includes more than 50 features.

4.4.1 Filtering Techniques

The first technique was plotting a correlation matrix as in the EDA section, although this time using the classification dataset. Since it is a big heatmap I have plotted it in smaller patches by filtering columns, for instance:

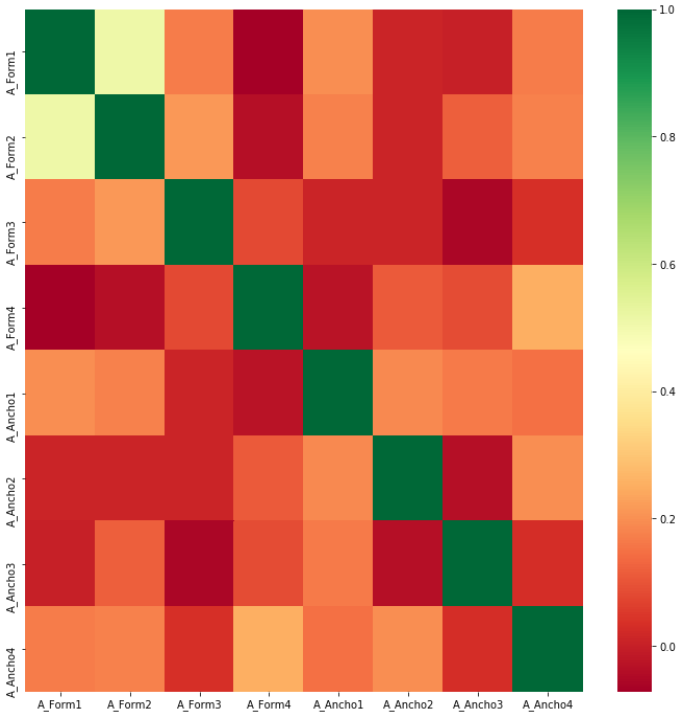


Figure 15: Heatmap of columns filtered by 'A' substring.

Then, for performing *univariate selection* I used sklearn's functions ***SelectKBest*** and ***chi2*** and plot the results again with seaborn. First, I fitted the ***SelectKBest*** model with the ***chi2*** function, then I created a dataframe to store the results and plot them. The output is the following:

```
#Applying SelectKBest to extract the 10 best features
bestfeatures = SelectKBest(score_func=chi2, k=10)
fit = bestfeatures.fit(Xnormalized,y)
...
featureScores = pd.concat([dfcolumns,dfscores, dfpvalues],axis=1)
...
display(featureScores.sort_values("Score",ascending=False).head(10))
ax = sns.barplot(x='Feature', y=col, data=featureScores, palette="Set2")
```

	Feature	Score	p-values
50	Snorer_no	2.736264	0.098094
4	BMI	2.674927	0.101940
2	Weight	2.428709	0.119131
1	Cervical	2.337841	0.126264
0	Age	0.970015	0.324676
10	A_Ancho2	0.703303	0.401676

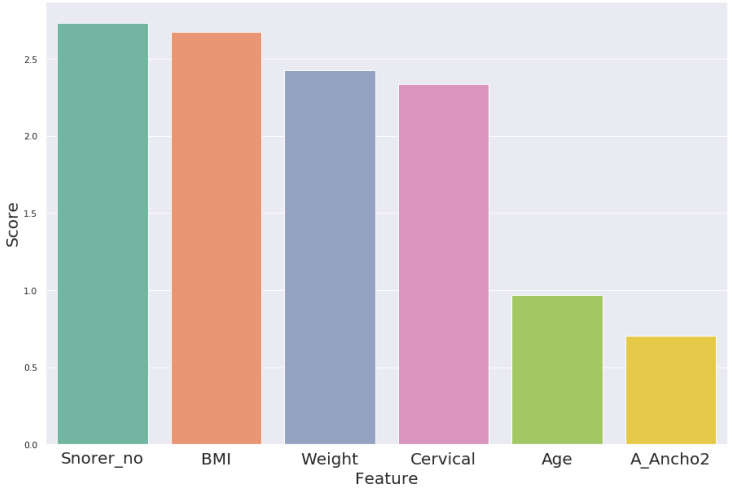


Figure 16: Output of the univariate selection model for scaled data.

4.4.2 Wrapping Techniques

For *recursive feature elimination* (RFE) I utilized the original model and other with cross-validation, both models are sklearn's function: ***RFE*** and ***RFECV***. They require also a model to fit them. I used a logistic regression also from sklearn. Then for the cross-validation case, I plotted the score by the number of features fitted. Here is the implementation just for the second case, as the first one is the same but shorter.

```
rfecv = RFECV(estimator=LogisticRegression(solver='liblinear'),
              step=1, cv=StratifiedKFold(5), scoring='accuracy')
rfecv.fit(Xnormalized, y)
...
sns.lineplot(x=range(1, len(rfecv.grid_scores_)+1),
             y=rfecv.grid_scores_, palette="Set2")
```

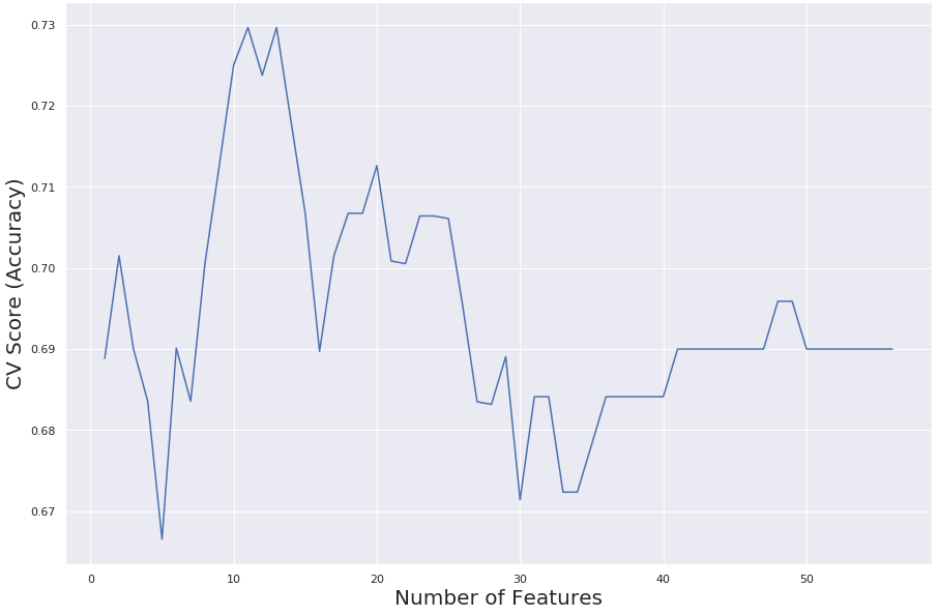


Figure 17: Line plot of the score by the total number of features fitted in the model.

This technique was applied in the final workflow modeling due to its potential. The final implementation is as follows:

```
rfecv = RFECV(estimator=model,scoring='accuracy', #passing the model
              cv=StratifiedKFold(5), step=1)
rfecv.fit(X, y)
#filtering the selected features
rfe_features = X.loc[:,rfecv_support].columns.tolist()
```

4.4.3 Embedded Techniques

I applied two different embedded techniques: *feature importance* and *SelectFromModel*. The former uses in-built class parameters from sklearn models such as tree algorithms. In this case, I used an ***ExtraTreesClassifier*** to asses the importance of the features and then I stored the result and plotted it as in the univariate example.

```
model = ExtraTreesClassifier()
model.fit(X,y)
dfimportance = pd.DataFrame(model.feature_importances_)
...
```

	Feature	Importance
1	Cervical	0.062219
2	Weight	0.055784
4	BMI	0.052326
5	A_Form1	0.034915
24	I_Form4	0.033448
39	U_Form3	0.031894

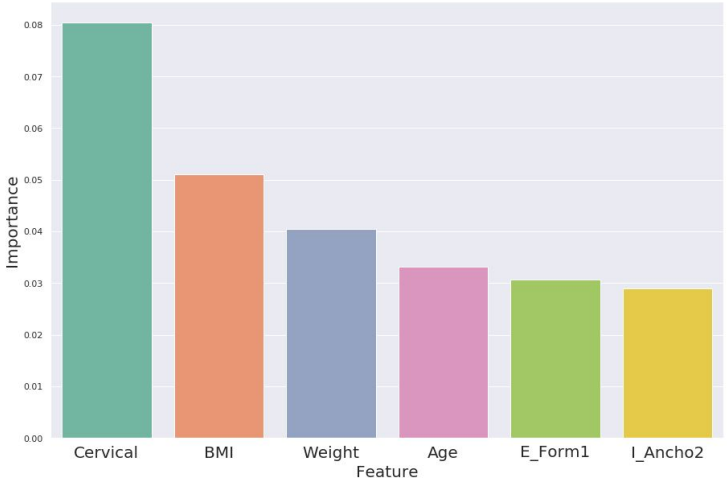


Figure 18: Output of the feature importance for scaled data.

Lastly, *SelectFromModel*. This is also a function from scikit-learn in which a model fits some data using a machine learning model and a maximum number of features. In my case, I set the maximum number of features to five and utilized a logistic regression model with lasso. The selected features were pretty similar to the other methods: *Age*, *Cervical*, *Weight*, *A_Ancho2*, and *Snorer_no*.

```
embeded_lr_selector = SelectFromModel(LogisticRegression(penalty="l1"),
                                      max_features=5)
embeded_lr_selector.fit(Xnormalized, y)
```

5. Machine Learning Modeling

My goal was to automate the process as much as possible while keeping it clean and easy to follow. My modeling notebooks tend to include the least amount of cells possible. Each of these cells is meant to achieve a distinct goal from the rest within the modeling workflow.

There are some small tweaks between the regression and classification notebooks, like the implementation of RFECV in the latter. Aside from fitting and testing the models, I created a dataset to automatically store the results that I wanted to keep for the next step, results and model comparison. In the following subsections, I give some code example of how I wrote the notebooks. For the complete source code, please visit my Github profile **[2]**.

5.1. Regression Modeling

The regression modeling notebook's cells are structured as follows:

Importing libraries. As the name implies, I imported first all the python modules that il was going to use: numpy, pandas, matplotlib, seaborn, many functions from several scikit-learn's submodules and all-purpose ones like os, time or math.

```
import os, time, math
...
import pandas as pd
import matplotlib.pyplot as plt
...
from sklearn.model_selection import cross_val_predict
from sklearn.preprocessing import MinMaxScaler
from sklearn.linear_model import LinearRegression
...
```

Importing the dataset. The next cell includes importing the dataset with pandas, setting the index as *Patient* and getting the dummy columns from categorical features.

```
df = pd.read_csv( 'OSA_propio.csv')
df.set_index("Patient", inplace=True)
df = pd.get_dummies(df, columns=[categorical_columns])
```

Helping functions. After importing the data, the first thing to do is to declare some helping functions to make the rest of the code cleaner. I want to highlight the most important ones since they include relevant parts of the workflow.

The first function is ***fit_model***. Its goal is to fit a model with cross-validation, then compute the metrics with the real and predicted values (either using functions from the scikit-learn's submodule metrics or numpy functions) along with the elapsed time and lastly return these results and the target feature arrays.

```
def fit_model(model, X, y):
    start = time.time()
    y_pred = cross_val_predict(model,X,y,cv=cv_on)
    end = time.time()
    ...
    mae = round(mean_absolute_error(y, y_pred),2)
    ...
    return y,  y_pred, metrics, elaped_time
```

The second function is ***plot_model***. Its goal is to plot the prediction versus real scatter, the error histogram, the absolute error boxplot, and the decision tree from the tree-based model using the results from the prior function and matplotlib. Lastly, I stored the figures just for models that I wanted to keep. Here is a summarized code snippet of one of the four plots:

```
def plot_model(y, y_pred, error, model_name, description, date):
    name_fig = model_name + "_" + description + "_" +  date
    ...
    plt.hist(error, bins=10)
    if save_model:
        plt.savefig("hist_"+name_fig)
    plt.show()
    ...
```

The third function is ***insert_result***. Its goal is to insert the results into a database to further analyze them altogether. Each new row in the database feature the name of the model and description (mostly hyperparameters), metrics, used *X* features, applied data tranformation, elapsed time and date. Here is a summarized pseudocode of the implementation.

```
def insert_result(model,description,mae,rmse,r2,interval,date):
    line = ""
    ...
    columns = [model_desc,metrics,X_features,tranformations,time,date]
    for col in columns:
        line = line + "," + str(col)
    os.system("echo " + line + " >> regression_results.csv")
```

Global parameters. This cell is a list of variable to enable data transformations (from the previous section), cross validation and select the model features. Here is an example:

```
#Feature Selection
x_features = ["Age","Cervical","Weight"]
y_features = ["IAH"]
#Data Transformations
minmax_scaler_on = True
```

Data Transformations. In this cell, I applied the selected data tranformation and filter the columns to fit the model. A better solution would be the implementation of scikit-learn's pipelines. Here is an example:

```
X = df[x_features]
y = np.reshape(df[y_features].values, (len(df[y_features]), 1))
if standard_sc_on:
    sc_x, sc_y = StandardScaler(), StandardScaler()
    X, y = sc_x.fit_transform(X), sc_y.fit_transform(y)
```

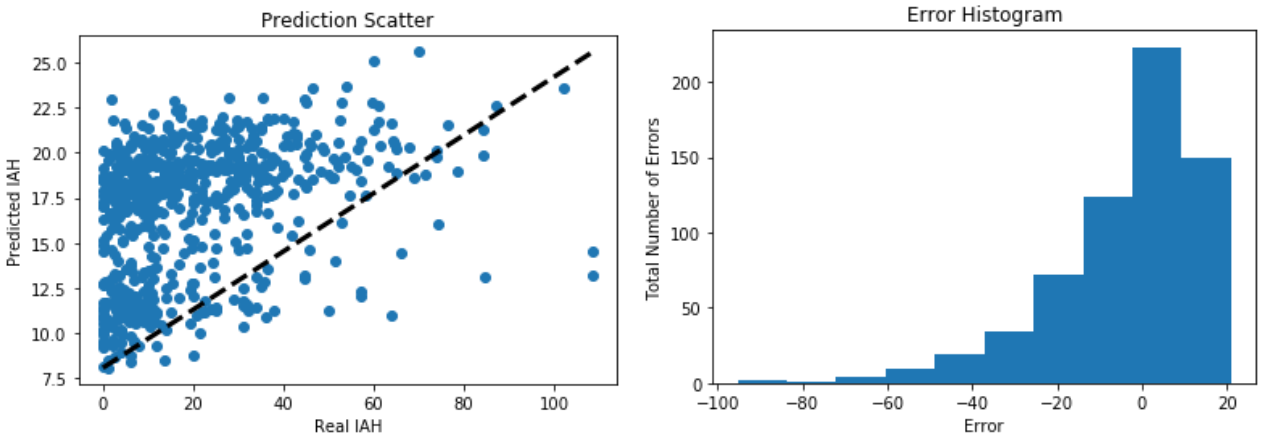
Model Dictionary. This cell includes a python dictionary with all the regression models I have used from scikit-learn. I used it to easily select the model that I wanted to try and change its hyperparameters. Here is an example:

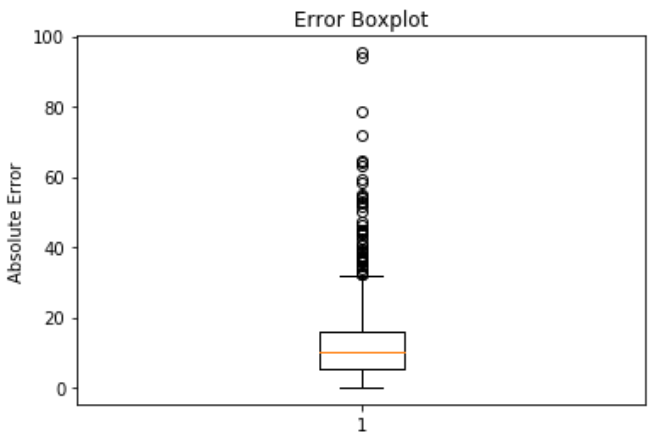
```
models={
    #Classical models
    "linear": LinearRegression(),
    ...
    #SVM models
    "svr_linear": LinearSVR(epsilon=1, C=10),
    ...
    #kNN models
    "knn": KNeighborsRegressor(n_neighbors=15),
    ...
}
```

Model fitting and results. The goal of the last cell is to run the model using the global parameters selected, the model selected and the helping functions. Here is the pseudocode of this implementation:

```
model_name, description = "sgd", "lossEpsilonRegL2"
save_model = True
#Fitting the model
... = fit_model(models[model_name], X, y)
#Plotting the model
plot_model(model_name, description, ...)
#Saving the model
if save_model:
    insert_result(model_name, description, ...)
```

This is the only cell that has an output, which is the plotting and storing of the three (or four) plots and the storing of the results in the database. The following figures shows the output of running this cell (bear in mind the database has more features than those in the figure):





model	description	MAE	RMSE	R^2	cv
mlp	hidlay50actReluSolverAdam	12.97	16.89	17.41	10
mlp	hidlay30-10actReluSolverAdam	12.74	16.88	17.49	10
sgd	lossSquaredRegL2	13.38	17.34	12.89	10

Figure 19: Final output of the regression model.

5.2. Classification Modeling

For the classification model, the workflow is almost the same as for the regression scenario. There were some small tweaks to adapt the notebook for classification such as using classification models and metrics or RFECV for feature selection. The latter was implemented as stated in subsection 4.4.2 and its corresponding graph was stored for each model that could work with this technique.

Regarding classification metrics, I again utilized the sklearn's metrics submodule. First, I imported functions that are focused on computing metrics to be analyzed without further processing. These are ***classification_report*** (it includes f1, precision, recall, and accuracy) and ***roc_auc_score***.

On the other hand, I imported a function that computes metrics to be used in graphs for their analysis. These are ***roc_curve*** and ***precision_recall_curve***. As their names imply, its goal is to generate the values for plotting the ROC curve and the precision and recall curve respectively.

To use these last metrics, it is necessary to set the parameter *method* in the cross-validation function ***cross_val_predict***. Depending on the classification model, this parameter is set to *'decision_function'* or *'predict_proba'*.

Regarding the classification models, it was really straight forward to adapt them to the notebook. It was just importing the classification version and checking some few hyperparameters that did not exist for this version. For models that did not feature the attribute *coef_*, the RFECV selection was turned off, since it did not work.

A summarized pseudocode of these changes and the output of the classification model are shown below. Note that the model also ouputs both a RFECV graph like in subsection 4.4.2 and a new row in the database of results like in the regression model. This two are not shown here to avoid repetition.

```
#Getting predictions for computing regular metrics
y_pred = cross_val_predict(model,X,y,cv=10)
#Getting predictions for computing plot metrics
if model in [list_of_multioutput_models]:
        y_score = cross_val_predict(..., method="predict_proba")
        y_score = y_score[:,1] # prob. of positive class
else:
        y_score = cross_val_predict(..., method="decision_function")
#Regular metrics
report = classification_report(y, y_pred,output_dict=True)
f1_pred = report["weighted avg"]["f1-score"]
#Plot metrics
fpr, tpr, thresholds = roc_curve(y, y_score)
plt.plot(fpr, tpr, linewidth=2)
```

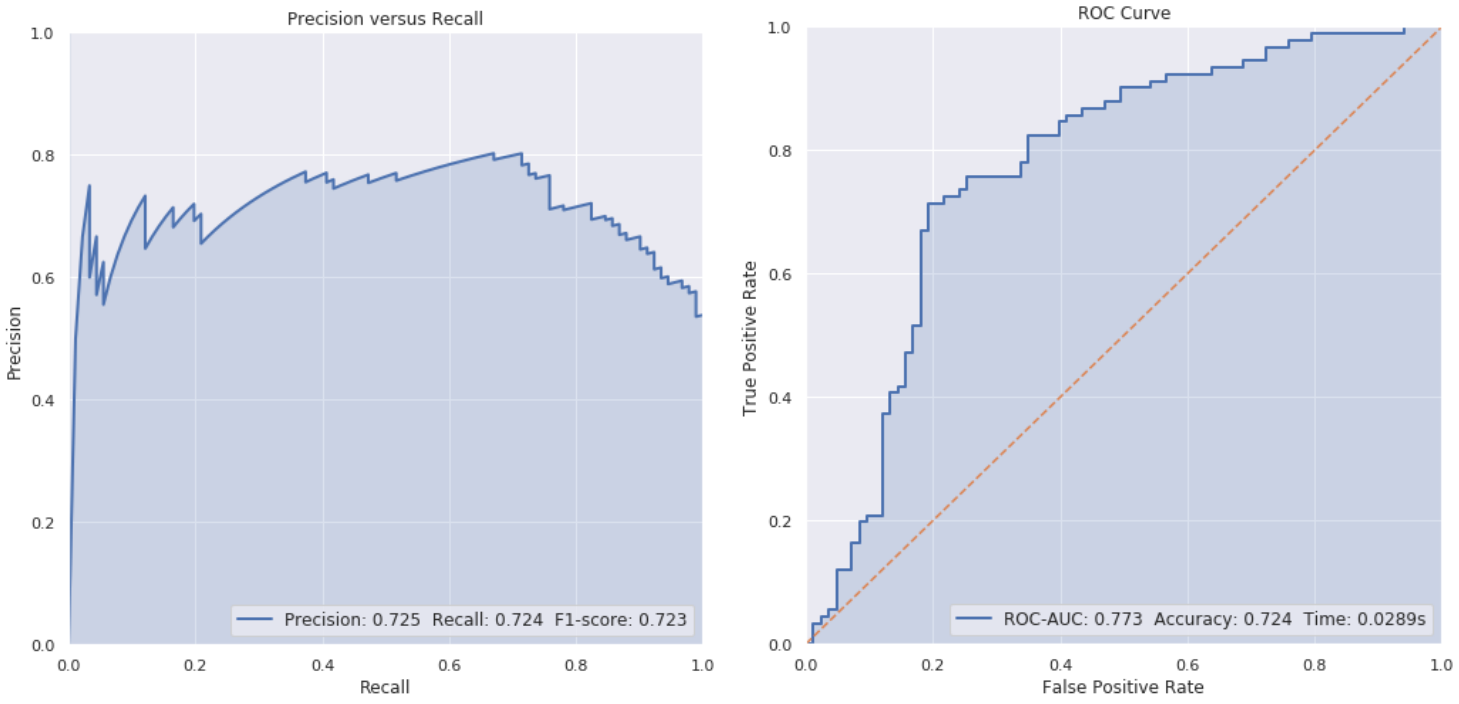


Figure 20: Plots from the output of the classification model.

6. Results and Model Comparison

Implementation-wise, the results were the easiest part of the methodology. It just consisted in importing the database of results (from both regression and classification), then some manipulation using pandas and finally plotting different grpahs using seaborn as in previous steps to draw conclusion from them.

My goal in implementing these plots was to make them look as appealing as possible to the viewer. This helps to communicate the ideas and concepts easier. For doing so, I tweaked the parameters of the seaborn graphs and managed to make them more colorful and pleasing to watch.

The following code snippet and figure example show how I procceded with these plots. It is a scatter plot for comparing the regression models using the metrics MAE and RMSD. In this example, I make the figure size bigger, font size of the axis titles bigger, the marker size bigger, the background white, the markers colorful using a specific color palette, the axis ticks to show up, adjust the axis limits and reduce the number of axis lines from four to two.

```
x, y, hue = "MAE", "RMSE", "model"
plt.rcParams['figure.figsize']=(10,10)
sns.set_style("white"), sns.set_style("ticks")
ax = sns.scatterplot(x=x, y=y, data=df, hue=hue, palette="Set2", s=200)
ax.set_xlabel(x,fontsize=20), ax.set_ylabel(y,fontsize=20)
ax.set(ylim=(16, 19.5), xlim=(11.5,17))
plt.setp(ax.get_legend().get_texts(), fontsize='15'), sns.despine()
```

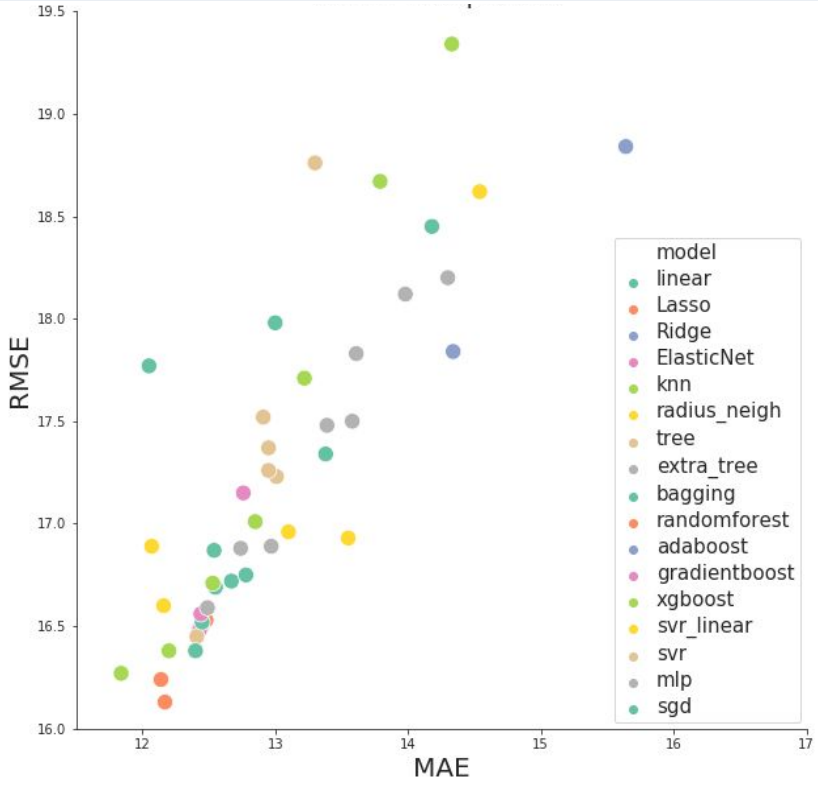


Figure 21: Scatter plot of regression model performance.

7. Conclusions

In summary, Python and its libraries are a great starting point for people to learn data science. They provide simple, yet powerful tools to cover the entire data science spectrum. From machine learning models to visualization, data manipulation, feature scaling, data transformation, model and feature selection or system management, to name a few.

On the other hand, Jupyter Notebook and Anaconda environment help to make the transition to this field easier and more appealing. As an already seasoned python programmer with experience using these libraries, I have not found any issue implementing the code. Apart from that, the dataset was really easy to handle due to its small size and simple features that did not need much processing.

Some extra features I wanted to implement in the final model but I ultimately discarded due to time constraints were more feature aggregation and selection techniques such as the ones presented in section 4, more data transformation techniques such as X^2 and *ln(x+1)*, more feature scaling techniques such as *mean normalization* and *scaling to unit length*, dimensionality reduction techniques based on *manifold* instead of *PCA*, automated *grid search* and *random search* using scikit-learn's cross-validated variants as I performed the *grid search* manually and a study of unsupervised learning techniques such as *K-means*.

I would like to remind the reader again, that the complete implementation and results can be found in my Github account **[2]**.

8. References

[1] *"Hands-On Machine Learning with Scikit-Learn & Tensorflow,"* Aurélien Géron, O'Reilly **[Accessed 3 November 2019]**

[2] Github, "Sergio Pérez Morillo Profile," [Online]. Available: https://github.com/spmorillo **[Accessed 4 November 2019]**