Project - Valerio Maglianella

Aim of the project

The aim of the present project is to develop machine learning and deep learning models for classification and regression tasks.

- **Classification task:** the aim is to build a model for the detection of patients with Parkinson's disease from biomedical voice measurements.
- **Regression task:** the aim is to build a model that predicts the price of houses in Boston suburbs based on various socioeconomic characteristics of the neighbourhood.

Classification task

Dataset

The dataset can be accessed online at https://archive.ics.uci.edu/ml/datasets/parkinsons.

Source. The dataset was created by Max Little of the University of Oxford, in collaboration with the National Centre for Voice and Speech, Denver, Colorado, who recorded the speech signals. The original study published the feature extraction methods for general voice disorders (Little, M. A., McSharry, P. E., Roberts, S. J., Costello, D. A. E., & Moroz, I. M. (2007). Exploiting nonlinear recurrence and fractal scaling properties for voice disorder detection. *BioMedical Engineering Online*, *6*, 23.).

Info. This dataset is composed of a range of biomedical voice measurements from 31 people (actually 32, Ed.), of which 23 (actually 24, Ed.) with Parkinson's disease (PD). Each column in the table is a particular voice measure, and each row corresponds to one of 195 voice recordings from these individuals ("name" column). There are around six recordings per patient (besides 3 participants having 7 recordings each, Ed.). The main aim of the data is to discriminate healthy people from PD, according to the "status" column which is set to 0 for healthy and 1 for PD.

Attribute Information:

- name - ASCII subject name and recording number
- MDVP:Fo(Hz) - Average vocal fundamental frequency
- MDVP:Fhi(Hz) - Maximum vocal fundamental frequency
- MDVP:Flo(Hz) - Minimum vocal fundamental frequency
- MDVP:Jitter(%), MDVP:Jitter(Abs), MDVP:RAP, MDVP:PPQ, Jitter:DDP - Several measures of variation in fundamental frequency
- MDVP:Shimmer, MDVP:Shimmer(dB), Shimmer:APQ3, Shimmer:APQ5, MDVP:APQ, Shimmer:DDA - Several measures of variation in amplitude
- NHR, HNR - Two measures of ratio of noise to tonal components in the voice
- status - Health status of the subject (one) - Parkinson's, (zero) - healthy
- RPDE, D2 - Two nonlinear dynamical complexity measures
- DFA - Signal fractal scaling exponent
- spread1, spread2, PPE - Three nonlinear measures of fundamental frequency variation

Data exploration

Pandas. I took a first look at the data using methods from the Pandas library (*head( )*, *info( )*, *describe( )*). I checked the number of subjects and observations per subject to verify that the dataset is balanced. In total there are 195 instances from 32 participants, each having 6 recordings sessions, besides 3 participants having 7 recordings each. Therefore, the dataset is balanced concerning the relevance of each subject. However, after dividing the dataset in *x* (features) and *y* (labels), I checked how many subjects were PD (status: 1) or healthy (status: 0) and I found an imbalance in this respect. Out of 32 subjects, there are 24 PD (147 instances) and 8 healthy (48 instances) (ratio 3:1).

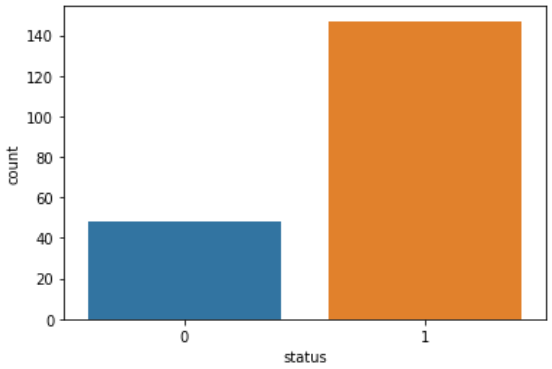

Figure 1 – Instances for healthy (=0) and PD (=1)

Imbalanced dataset. It is important to bear in mind that the **data is imbalanced** for the next steps of analysis. In particular, in order to overcome this problem and avoid biases, I will add the *stratify* parameter when using the *train_test_split( )* function, to make sure that our splits represent the distribution of the target variable in a more accurate way.

For similar reasons, I will try different algorithms and combinations of hyperparameters setting up a **stratified cross-validation** strategy with **grid-search**, aiming at having a more robust strategy to deal with class imbalances. Also, I will include decision trees models, which often perform well on imbalanced datasets.

Finally, to measure the performance of our models, we will focus more on **F1** scores, given that accuracy would not be the most appropriate metric for imbalanced datasets.

Also, I considered resampling the dataset. In particular, since the dataset is small, we could try some oversampling strategies using **SMOTE**. I considered this as a last option to apply eventually only if the performance of the models could not be increased with the other strategies described above. Since this was not the case, in the end I did not apply SMOTE on this occasion.

Data visualization

Data distribution. I analysed the distribution of the data plotting the dependent (labels, *y*) and independent (features, *x*) variables, using Seaborn and Matplotlib libraries. After plotting the **features distribution**, I noticed that many features show a **positively skewed distribution**, meaning that they have a higher number of data points having low values. I considered applying a logarithmic transformation in order to make data more normally distributed.

After a little test, I also reflected on the fact that there are models that do not make assumptions that the underlying data distribution is a normal distribution. For example, **Support Vector Machine** just cares about the boundaries of the separating hyperplane and does not assume the exact shape of the distributions. **Decision Tree** models also do not make such assumptions. I preferred applying these models, rather than applying log transformations.

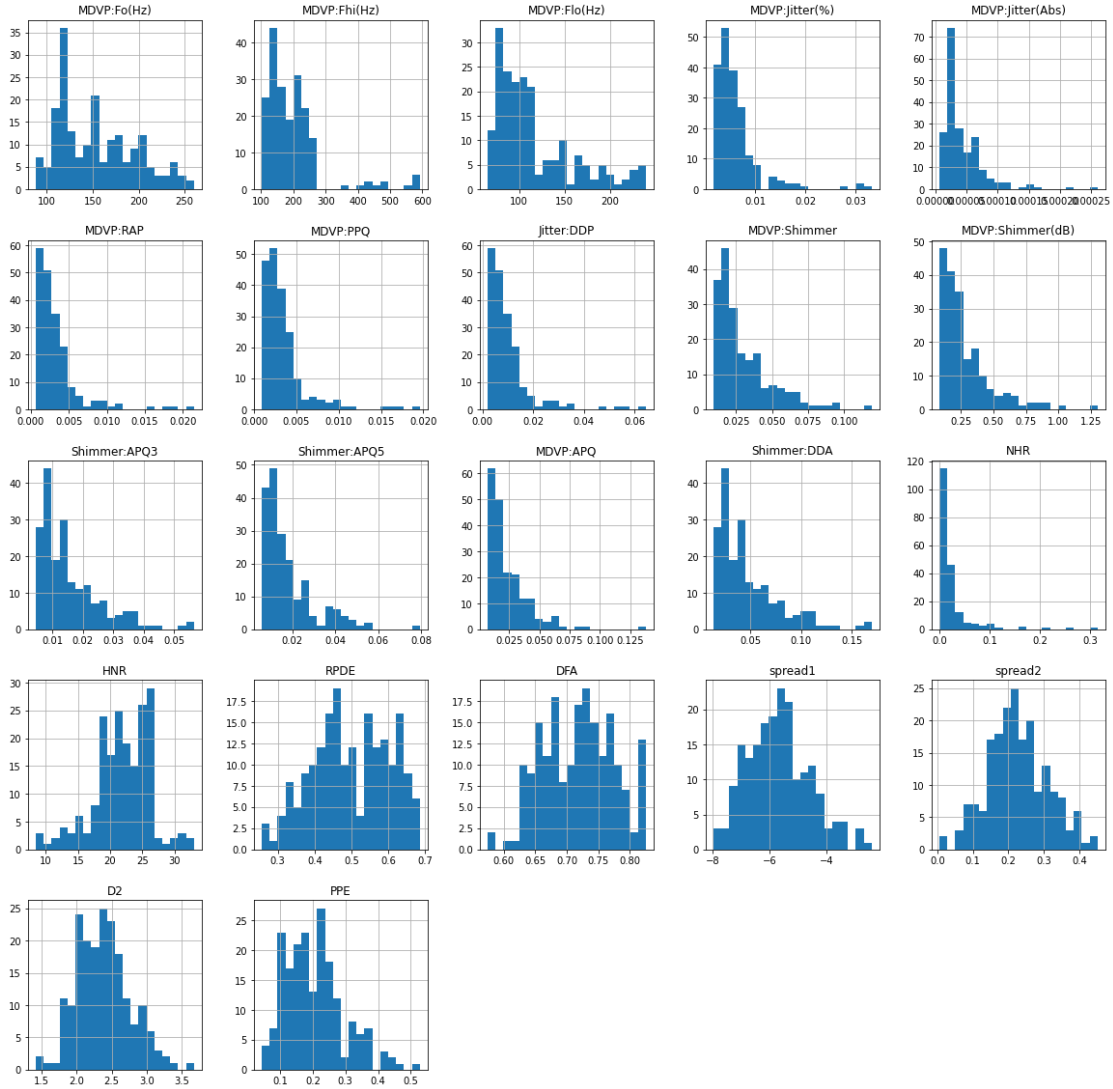

Figure 2 – Overall distribution of features

Data correlation. I plotted the **correlations between features** using different graphical representations. Overall, I noticed various correlations between groups of features, even high correlations (> ~0.9). I considered this multicollinearity issue and I considered dealing with it with two main feature selection strategies: 1. dropping highly correlated variables; 2. extracting new features with Principal Component Analysis (PCA).

However, in this occasion I did not apply these methods, given that machine-learning-based methodologies such as random forests and neural networks (but also SVM with a non-linear kernel) are largely immune to biases caused by collinearity.

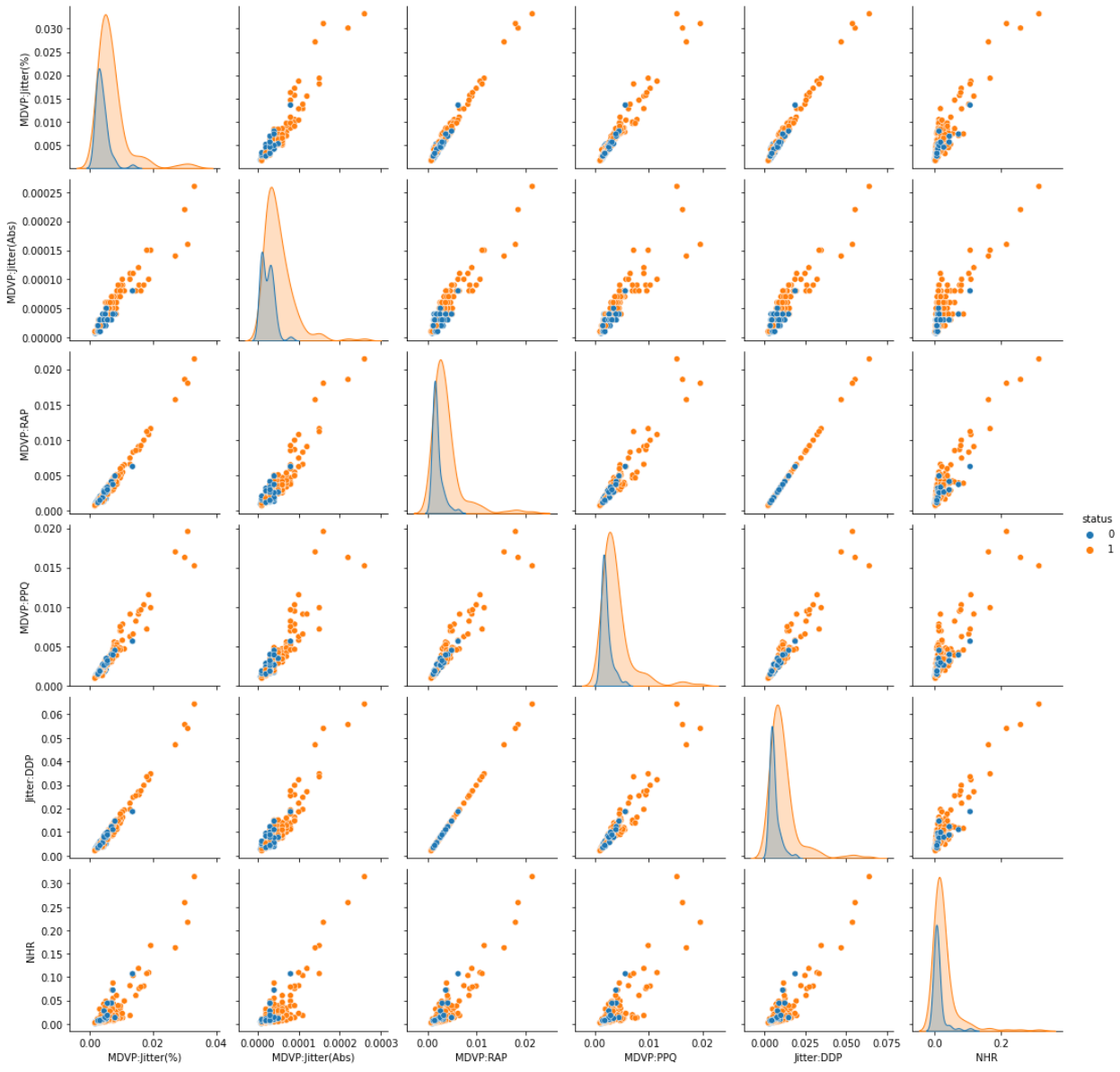

Figure 3 - Plot of some features showing high correlation (> .90).
The diagonal shows the distribution of the features depending on the status.

Data preparation

We already checked that there are no missing values. However, the data present different scales and units of measure, therefore we will need to apply some Feature scaling technique, such as *MinMaxScaler*, *StandardScaler*, *RobustScaler*. Since the data are not normally distributed, we may prefer to use *MinMaxScaler*. Still, we may try different scalers depending on the model in use.

In particular, feature scaling will not be required for Decision Tree models. The results shown below were obtained applying *MinMaxScaler* to all models, except for Decision Tree/Random Forest for which we kept the original values without any scaling.

Train-test split & Feature scaling

I split the data into train (70%) and test (30%) subsets setting the *random_state* and *stratify* arguments to ensure that both the train and test sets have the proportion of examples in each class that is present in the provided *y* array. I apply the feature scaling procedures only after splitting the data to avoid data snooping. When you fit the scaler on the whole dataset, information from the test set is used to normalise the training set. This is a common case of "data leakage", which means that information from the test set is used while training the model. This often results in overestimates of the model's performance. To avoid this, we apply the scaler at this stage, after *train_test_split*.

I used 4 different models to perform the classification task: Support Vector Machine (SVM); Decision Tree/Random Forest; Multilayer Perceptron Classifier (MLP, imported from sklearn + implemented manually with Keras).

Classification models: training, validation, test

Support Vector Machine (SVM) Classifier

The Support Vector Machine is a supervised learning algorithm mostly used for classification but it can be used also for regression. The main idea is that based on the labelled data (training data) the algorithm tries to find the optimal hyperplane which can be used to classify new data points. In two dimensions the hyperplane is a simple line. In order to build this hyperplane, this "margin" that separated the classes, SVM finds the most similar examples between classes, the so-called "support vectors".

SVM can be a very efficient and accurate method, which works very well on small datasets like the one we are analysing in the present project.

For SVM (and also for the next models) I run a stratified 5-fold cross validation to create better fitting models by training and validating on all parts of the training dataset. In particular, I run five iterations of the model, each of which treats a different fold as the validation set and trains on the other four folds. Once all five iterations are complete the resulting iterations are averaged together creating the final cross validation model. I used cross validation together with a grid-search approach in order to find the hyperparameters that maximise classification performances.

In particular, I tuned the following hyperparameters:

1. "C": the regularisation parameter, which tells us how much misclassification we want to avoid (higher C → "hard margin" of SVM and risk of overfitting; lower C → "soft margin")
2. "Kernel": the set of mathematical functions that transform data into the required form
3. "Degree": of the polynomial Kernel function
4. "Gamma": which defines how far the influence of a single training example reaches (decreasing the Gamma will result that finding the correct hyperplane will consider points at greater distances so more and more points will be used → less overfitting)

All the configurations and the relative accuracy, precision, recall and F1 scores have been evaluated. For each model, I selected the hyperparameters that maximised the F1 score, to balance both precision and sensitivity of the predictions (reminder: accuracy is a less appropriate metric for imbalanced datasets like this one).
Grid-search identified the following best parameters:

1. C: 100
2. Kernel: 'rbf'
3. Degree: 1
4. Gamma: 'scale', which equals to a value of 1/(tot num features * tot feature variance)

The selected model was able to correctly predict the status of 136/136 observations in the train set with a mean cross-validated F1 score of 0.94.

To test generalisation, the selected model was able to correctly predict the status of 57/59 observations in the test set, achieving a weighted average of F1-score of 0.97; 1 healthy subject and 1 PD were mistakenly attributed to the wrong class (see Table 1).

	Predicted healthy	**Predicted PD**	**Precision**	**Recall**	**F1-score**
True healthy	14	1	0.93	0.93	0.93
True PD	1	43	0.98	0.98	0.98

Table 1 - Confusion matrix and classification metrics of the Support Vector Machine Classifier

Decision Tree/Random Forest Classifier

Random forest algorithm is one of the most popular and powerful supervised machine learning algorithms capable of performing both classification and regression tasks. This algorithm creates a forest with several decision trees. Initially, I tried using a decision tree algorithm, but since the performance was not great, I switched to a Random Forest, in order to limit overfitting, increase generalization and deal better with the imbalanced dataset.

I used a grid-search approach again to find the best hyperparameters that maximize classification performances. In particular, I tuned the following hyperparameters:

1. "n_estimators": the number of trees in the forest
2. "max_features": the number of features to consider when looking for the best split
3. "max_depth": the maximum depth of the tree
4. "criterion": the function to measure the quality of a split

Grid-search identified the following best parameters:

1. n_estimators: 300
2. max_features: 'sqrt'
3. max_depth: 8
4. criterion: 'entropy'

The selected model was able to correctly predict the status of 136/136 observations in the train set with a mean cross-validated F1 score of 0.92.

To test generalization, the selected model was able to correctly predict the status of 56/59 observations in the test set, achieving a weighted average of F1-score of 0.95; 2 healthy subjects and 1 PD were mistakenly attributed to the wrong class (see Table 1).

	Predicted healthy	**Predicted PD**	**Precision**	**Recall**	**F1-score**
True healthy	13	2	0.93	0.87	0.90
True PD	1	43	0.96	0.98	0.97

Table 2 - Confusion matrix and classification metrics of the Random Forest Classifier

Multilayer Perceptron Classifier (MLP)

Multilayer Perceptron (MLP) consists of a network of nodes (processing elements) arranged in layers: an input layer that receives external inputs, one or more hidden layers, and an output layer which produces the classification results. Except for the input nodes, each node is a neuron that uses a nonlinear activation function. Backpropagation is the learning mechanism that allows the Multilayer Perceptron to iteratively adjust the weights in the network, with the goal of minimizing the cost function.

It is possible to implement an MLP using sklearn library. I did this using a grid search approach again. Then, I also tried to implement an MLP manually using Keras (see the next paragraph).

I tuned the following hyperparameters:

1. "hidden_layer_sizes": which specifies the number of hidden layers and of neurons inside each of them
2. "activation": the activation function for the hidden layers, which transforms the input received by every neuron and is fundamental to introduce non-linearity into its output
3. "solver": the algorithm used for optimizing the nodes weights
4. "learning_rate": which controls whether the learning rate is constant or changes across training
5. "learning_rate_init": is a number which controls how much to change the model in response to the estimated error each time the model weights are updated
6. "max_iter": maximum number of iterations

Grid-search identified the following best parameters:

1. "hidden_layer_sizes": (100, 100, 100)
2. "activation": 'relu'
3. "solver": 'lbfgs'
4. "learning_rate": 'constant'
5. "learning_rate_init": 0.001
6. "max_iter": 200

The selected model was able to correctly predict the status of 136/136 observations in the train set with a mean cross-validated F1 score of 0.95.

To test generalization, the selected model was able to correctly predict the status of 56/59 observations in the test set, achieving a weighted average of F1-score of 0.95; 2 healthy subjects and 1 PD were mistakenly attributed to the wrong class (see Table 1).

	Predicted healthy	**Predicted PD**	**Precision**	**Recall**	**F1-score**
True healthy	13	2	0.93	0.87	0.90
True PD	1	43	0.96	0.98	0.97

Table 3 - Confusion matrix and classification metrics of the Multilayer Perceptron Classifier

Deep Neural Network

Finally, I also tried to implement an MLP manually using Keras.

Models.Sequential() allow initializing a new model. Each layer is added sequentially to the previous one, using *model.add()* and *layers.Dense()*, to add a layer where each neuron has connections with all neurons of the previous layer. The hidden layers have ReLU as the activation function, the output layer has Softmax as the activation function. The number of output units is equal to the number of classes that the network can recognize, in this case 2.

Model.compile() configures the model for the training phase. I set the optimizer, loss function and training metrics. I chose "adam" as optimizer, which is the most used and computationally efficient optimization algorithm; then "SparseCategoricalCrossentropy", which is a loss function used when there are two or more label classes provided as integers (it does not require one-hot encoding); then "accuracy" as metric.

I trained the model for 500 epochs and plotted the model training history. Finally, I tested the model on the test set and evaluated the model performance with classification report and confusion matrix.

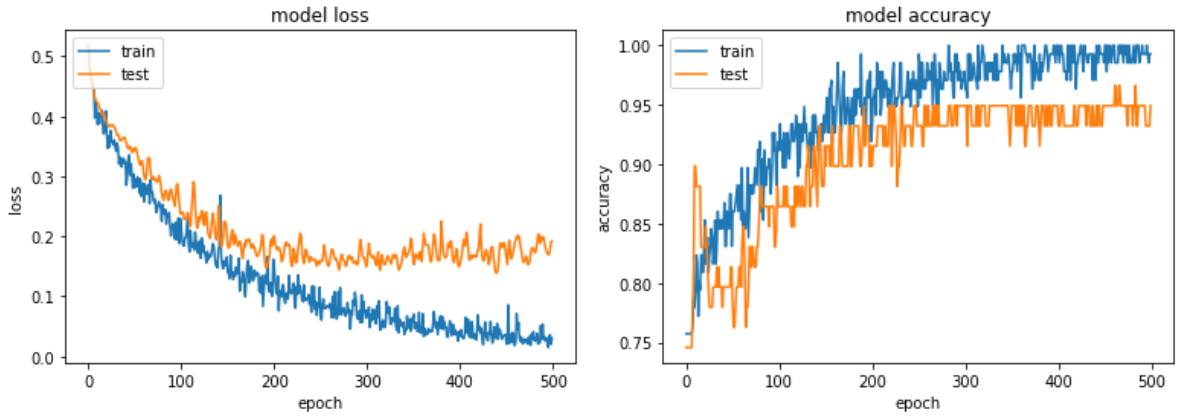

Figure 4 - Plot of model loss and accuracy on train and test sets

The selected model was able to correctly predict the status of 136/136 observations in the train set.

To test generalization, the selected model was able to correctly predict the status of 57/59 observations in the test set, achieving a weighted average of F1-score of 0.97; 2 PD subjects were mistakenly attributed to the healthy class (see Table 1).

	Predicted healthy	**Predicted PD**	**Precision**	**Recall**	**F1-score**
True healthy	15	0	0.88	1.00	0.94
True PD	2	42	1.00	0.95	0.98

Table 4 - Confusion matrix and classification metrics of the MLP using Keras.

Comparisons

Overall, the SVM algorithm outperformed both the Random Forest and the MLP from sklearn, with higher accuracy and weighted average precision, recall and F1 scores (see Table 4). The MLP built manually showed a performance similar to the SVM.

	Accuracy	Precision	Recall	F1-score
SVM	0.97	0.97	0.97	0.97
Random Forest	0.95	0.95	0.95	0.95
MLP	0.95	0.95	0.95	0.95
DNN	0.97	0.97	0.97	0.97

Table 5 – Models comparison. Precision, recall and F1 scores are the weighted average of the two classes

Finally, for further discussion, here I also reported the performance of the models using the macro averages. For an imbalanced dataset like the one we have, using the macro average would be a good choice to treat all classes equally. However, if we want to assign greater contribution to the class with more examples in the dataset, which would make sense in our case since the most numerous class is the one identifying PD, then the weighted average is preferred.

	Accuracy	Precision	Recall	F1-score
SVM	0.97	0.96	0.96	0.96
Random Forest	0.95	0.94	0.92	0.93
MLP	0.95	0.94	0.92	0.93
DNN	0.97	0.94	0.98	0.96

Table 6 – Models comparison. Precision, recall and F1 scores are the macro average of the two classes

Regression task

(Note: the description of this task will be shorter because I will use a lower number of models and many considerations made for the classification task are still valid for this task)

Dataset

Source. The "Boston Housing Dataset" is a part of sklearn library and can be imported with the command *"from sklearn.datasets import load_boston".* The data in this dataset was collected by the U.S. Census Service, and it first appeared in a paper of 1978 (Harrison, D. J., & Rubinfeld, D. L. (1978). Hedonic housing prices and the demand for clean air. *Journal of Environmental Economics and Management, 5*(1), 81–102.)
Info. This dataset has been used in many machine learning papers that address regression problems. The dataset contains information concerning the median prices of Boston houses in the mid-1970s and various socioeconomic characteristics of the area. Each of the 506 rows in the dataset describes a Boston suburb or town, and it has 14 columns with information such as average number of rooms per dwelling, pupil-teacher ratio, and per capita crime rate. The last row describes the median price of owner-occupied homes (this leaves out homes that are rented out), and it's the row that we are trying to predict using regression in this task.

Data exploration

Pandas. I transformed the data into a dataframe, then I took a first look at the data using methods from the Pandas library (*head( ), info( ), describe( )*).

Data visualization

Data distribution. I analysed the distribution of the data plotting the dependent (label, *y*) and independent (features, *x*) variables, using Seaborn and Matplotlib libraries. Overall, I noticed different types of distributions.
Data correlation. I plotted the **correlations between features** using different graphical representations. Overall, I noticed various moderate and high correlations between groups of features.

Data preparation

We already checked that there are no missing values. However, the data present different scales and units of measure, therefore we will need to apply some feature scaling, such as *MinMaxScaler*, *StandardScaler*, *RobustScaler*. Since the data are not normally distributed, we may prefer to use *MinMaxScaler*. The results shown below were obtained applying *MinMaxScaler* to all models.

Train-test split & Feature scaling

I split the data into train (70%) and test (30%) subsets setting the *random_state*. I apply the feature scaling procedures only after splitting the data to avoid data snooping.

I used 2 different models to perform the regression task: Support Vector Machine (SVM) Regressor; Decision Tree Regressor.

Regression models: training, validation, test

Support Vector Machine (SVM) Regressor

The Support Vector Machine is a supervised learning algorithm mostly used for classification but it can be used also for regression, as in this case.

Similarly to the classification task, I used a 5-fold cross validation together with a grid-search approach in order to find the hyperparameters that maximize regression performances. I tuned the same hyperparameters of the SVM Classifier and identified the following best parameters:

1. C: 1000
2. Kernel: 'rbf'
3. Degree: 1
4. Gamma: 'scale', which equals to a value of 1/(tot num features * tot feature variance)

Performance evaluation was obtained as the Root Mean Squared Error (RMSE) and R2 scores. Both RMSE and R2 quantify how well a regression model fits a dataset. The RMSE tells us how well a regression model can predict the value of the response variable in absolute terms, while R2 tells us how well a model can predict the value of the response variable in percentage terms.

The model performance for the training set was: **RMSE = 1.849; R2 score = 0.957.**
The model performance for the testing set was: **RMSE = 3.619; R2 score = 0.857.**

Decision Tree Regressor

Decision Tree algorithm is one of the most popular and robust supervised machine learning algorithms capable of performing both classification and regression tasks, as in this case.

Similarly to the classification task, I used a 5-fold cross validation together with a grid-search approach in order to find the hyperparameters that maximize classification performances.

I tuned the following hyperparameters:

1. "criterion", the function used to calculate node impurity and reduce it
2. "splitter", the strategy used to choose the split at each node
3. "max_features", the number of features to consider when looking for the best split
4. "max_depth", the maximum depth of the tree
5. "min_samples_leaf, the minimum observations required in a leaf (used to control overfitting)

I identified the following best parameters:

1. criterion: 'squared_error'
2. splitter: 'best'
3. max_features: None
4. max_depth: 6
5. min_samples_leaf: 1

Performance evaluation was obtained as the Root Mean Squared Error (RMSE) and R2 scores.
The model performance for the training set was: **RMSE = 2.043; R2 score = 0.948.**
The model performance for the testing set was: **RMSE = 3.923; R2 score = 0.832.**

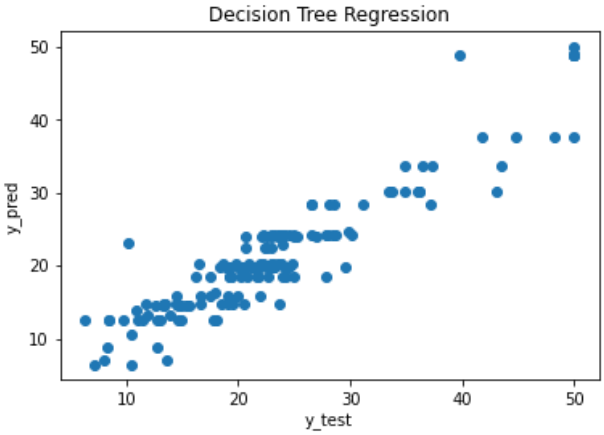

Figure 5 - Scatter plot of predicted versus actual MEDV values

To evaluate the contribution of each feature towards the prediction of house prices, the feature importance of the model was computed using the fitted attribute *feature_importances_* for the *Regression Tree*.

Feature importance analysis revealed that, among the 13 features included in the dataset, the *% lower status of the population* (LSTAT) and the *average number of rooms per dwelling* (RM) were those most informative for the regression task, with scores of 0.57 and 0.25 respectively (Fig 8).

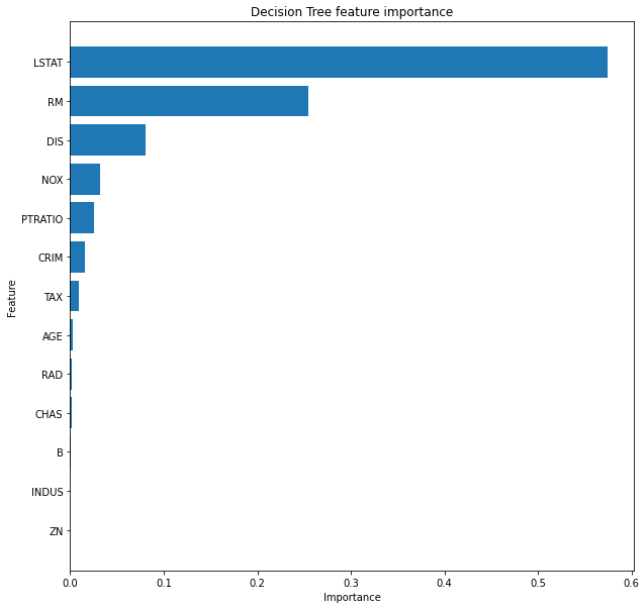

Figure 6 – Estimated feature importance coefficients