Data Analytics Project Report

Claudia Citera, Riccardo Evangelisti

Academic Year 2023/2024

Contents

Introduction

The following project was developed as part of the **Data Analytics** course of the Master's Degree in Computer Science at the University of Bologna.

The objective of the project is to carry out a data analytics study, which involves the implementation of all the analytical pipeline phases studied during the course:

- Data Acquisition
- Data Visualization
- Data Preprocessing
- Modeling
- Performance Evaluation

The main purpose of this study is to recognize the **year** in which a song was published based on the features of **its audio track**.

The following functionalities were developed:

1. Traditional non-deep supervised Machine Learning techniques
2. Supervised ML techniques based on Neural Networks
3. Supervised ML technique with deep models for TabularData

Data Acquisition

The Data Acquisition phase involves collecting the data that needs to be analyzed. Data can be acquired in various ways, including **static acquisition**, which was used in this project.

The dataset used in this project consists of a single csv file with 252175 rows and 91 columns. One of the columns represents the **year** of publication of the song, ranging from 1956 to 2009. All other columns contain **floating-point numbers** related to the **audio track**. Since the error calculation of a classification problem would not distinguish the error gravity (for example the error for a prediction of "2007" instead of "2008" would be equal to the error of a prediction of "1956" instead of "2008"), **regression** models were used to solve the problem.

Before proceeding with further operations, the dataset was analyzed to check for missing or duplicate values. It was found that there are **no missing values** and only **52 duplicate rows**. Since they are very few and mainly related to the most represented classes, we decided to **drop** them.

	Year	S0	S1	S2	S3	S4	S5	S6	S7	S8	...
0	2007	44.76752	114.82099	3.83239	27.99928	1.49153	-15.90853	28.24844	3.61650	-7.24653	...
1	2004	52.28942	75.73319	11.35941	-6.20582	-27.64559	-30.75995	12.50955	7.47877	9.88498	...
2	2005	33.81773	-139.07371	134.19332	17.85216	63.47408	-25.28005	-34.65911	-5.99135	1.27848	...
3	1998	41.60866	3.17811	-3.97174	23.53564	-19.68553	20.74407	18.80866	6.24474	-7.98424	...
4	1987	44.49525	-32.25270	58.08217	3.73684	-32.53274	-18.72885	-15.85665	-3.34607	22.63786	...

Figure 2.1: Some example rows of the dataset

The dataset was divided into three parts: **train**, **validation** and **test**, using a **70-20-10** split scheme. Since the dataset is very imbalanced, the "stratify" option (of the train_test_split function of sklearn) is set to be sure of having some samples of every class (Year) in all the sets. In addition, a seed was set to ensure that the dataset split is consistently reproducible in the future.

The same validation test was used to find the best hyperparameters across the three different functionalities and the same test set was used to evaluate all models, to ensure consistent comparability of the results obtained.

Data Visualisation

Data Visualization is a step in the pipeline that aims to analyze data clearly and effectively through the use of graphs. It is useful for exploring data efficiently and discovering possible relationships.

In this section of the project, several graphs and visual representations were produced to provide an overview of the dataset in question. Since the dataset has a high number of features, we chose to focus only on some key information and represent it visually to make it easier to understand and analyze the data.

As a first step, we checked the **correlation** between the various columns to see if it was possible to remove any of them.

From *Figure 3.1* we can notice that the dataset shows a **low degree of correlation** among its various features. Even the correlation with the Year column is very low. However, a closer examination of the top-left quadrant reveals a cluster of variables exhibiting a stronger correlation. A zoomed-in view of this region is provided below (*Figure 3.2*) for further analysis.

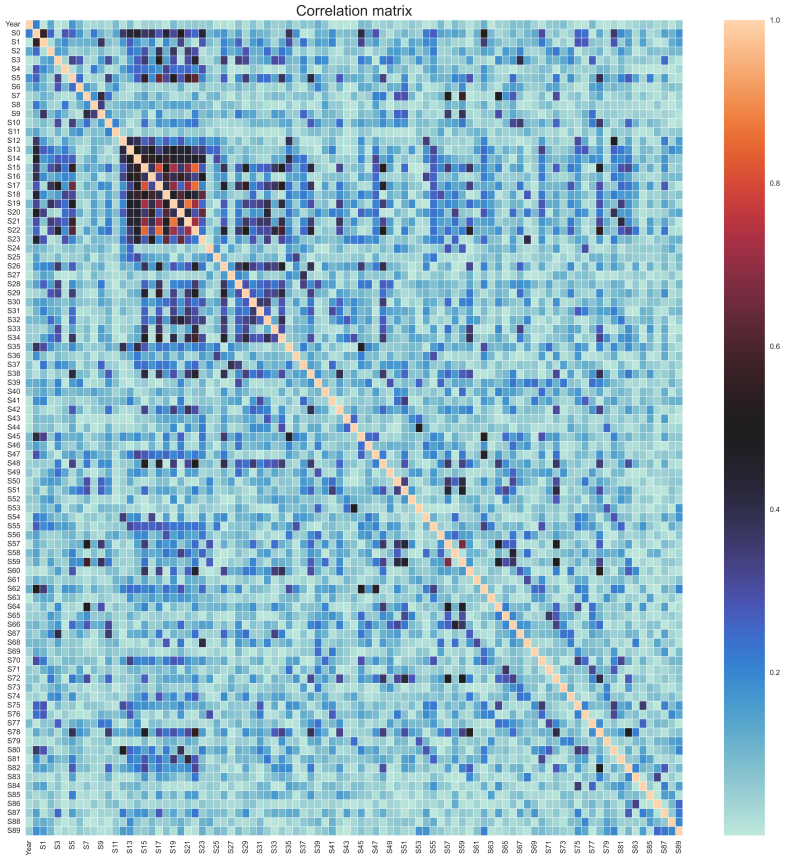

Figure 3.1: Correlation matrix between all the features

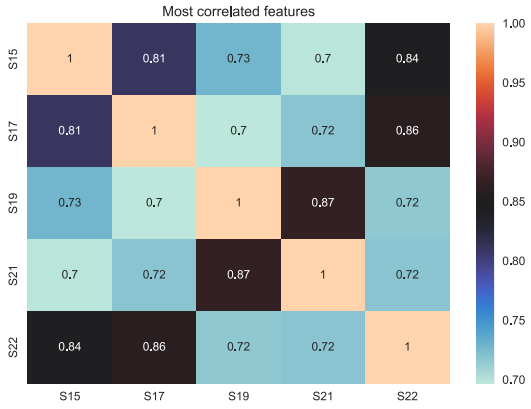

Figure 3.2: Correlation matrix close up to the Top-Left corner

A very relevant piece of information is the distribution of the Year attribute. *Figure 3.3* reveals that the dataset is very **imbalanced**. In fact, more recent songs (from around 1990 to 2009) are much more represented than older ones (from 1990 backwards).

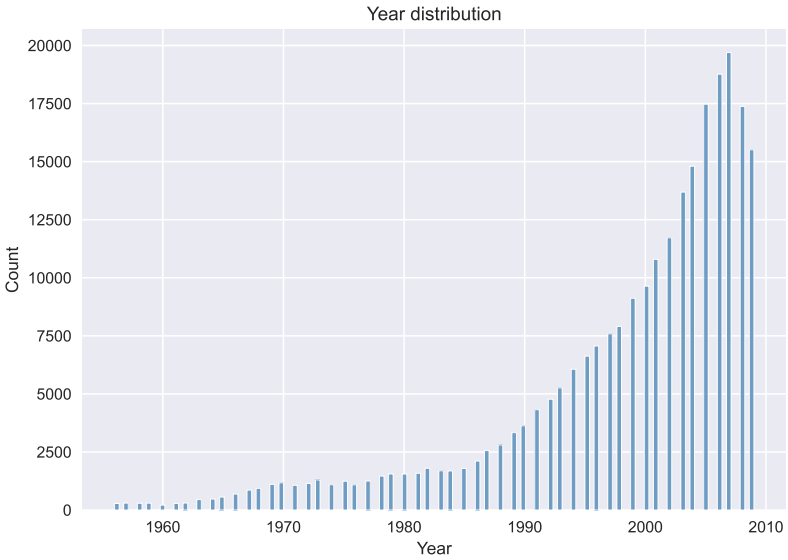

Figure 3.3: Distribution of the feature **Year**

Then we checked the distribution of the other variables and we noticed that all the attributes exhibit a **Gaussian distribution** with varying skewness and their means are centred around 0 (or close to it), except for the attribute *S0*. To make it easier to see, we split the 90 features into two plots, as shown into *Figure 3.4* and *Figure 3.5* The plots indicate that the distribution ranges differ a lot across the various columns. Therefore, it's probable that we will require a preprocessing model to **rescale** the data.

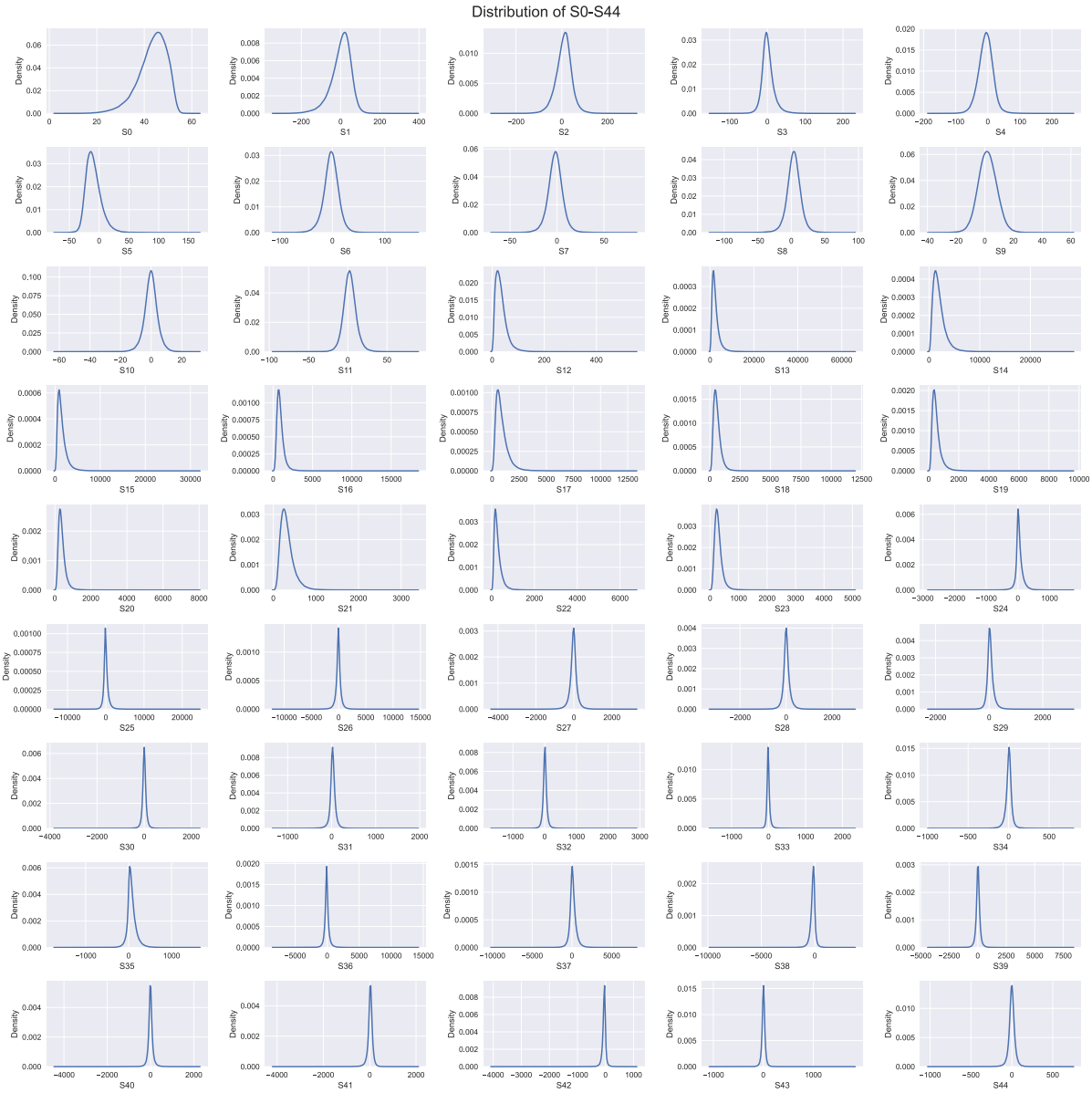

Figure 3.4: Distribution of the features S0-S44

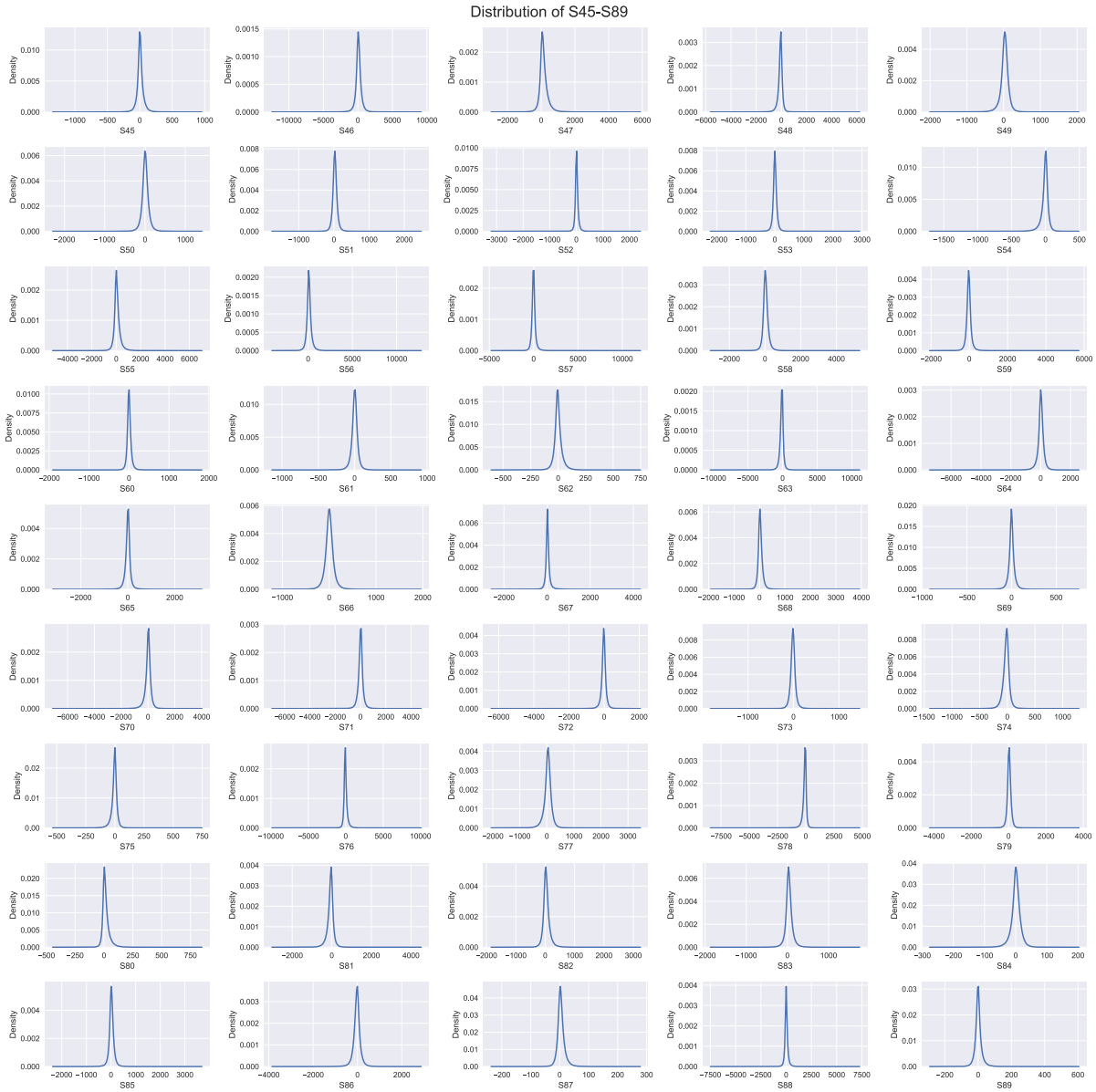

Figure 3.5: Distribution of the features S45-S89

Data Preprocessing

Preprocessing the data is a crucial phase before applying any ML model. That's because some models require a specific data distribution, are sensitive to different feature ranges of values or are affected by the curse of dimensionality. Here follows a description of all the preprocessing techniques we tried and why.

4.1 Scaling and Normalizing

Standard Scaling

Since some features haven't the mean in zero, Standard Scaling was tried. This method removes the mean and scales the data to unit variance.

Min-Max Scaling

The value ranges of the data differ a lot among the features. Many models perform generally better when the data is confined between fixed values. This method scales the data to a [0,1] range.

L1/L2/Lmax Normalization

For each sample, the norm is computed over all the features and applied (divided) only to the single sample.

4.2 Dimensionality Reduction

Since many models suffer from the curse of dimensionality and the dataset has many features, dimensionality reduction was tried.

4.2.1 PCA

PCA (Principal Component Analysis) is an unsupervised method that uses an orthogonal transformation to convert a set of correlated variables into a set of uncorrelated variables, called Principal Components. The primary goal of PCA is to reduce the dimensionality of a dataset while preserving the most important patterns or relationships between the variables, without any prior knowledge of the target variables (it does not consider the class labels of the samples).

The method calculates the covariance matrix of the (preferably standardized) data. Then, the eigenvalues and eigenvectors of the covariance matrix are calculated. The eigenvectors (which represent the directions in which the data vary the most) become the axes of the new space, while the eigenvalues (which indicate the amount of variance along each axis) determine the importance of each eigenvector. The Principal Component features are derived by taking the dot product of the eigenvector and the standardized columns.

In our study case, the number of Principal Components chosen was the one that preserves 95% of the total variance: **54** Principal Components.

Unfortunately, PCA works on the correlation between the various features and in our study case they are very low correlated. Therefore we tried another method for dimensionality reduction.

4.2.2 LDA

LDA (Linear Discriminant Analysis) is a supervised learning algorithm used to find a linear combination of features that best separates the classes in a dataset. LDA works by projecting the data into a lower-dimensional space that maximizes the separation between the classes. In our case, the classes are the Year feature, even if the task is later treated as a regression problem.

LDA assumes that the data has a Gaussian distribution, like our study case.

The method calculates the *within-class* (S_W) and *in-between-class* (S_B) scatter matrices: the first captures the information about the spread of the data within each class, while the second gathers information about how classes are spread between themselves. Then, LDA calculates a matrix, called the Projection matrix, that, multiplied by the original dataset, gives as a result a new space such that the class separability is maximized. It means maximizing the *in-between-class* scatter matrix while minimizing the *within-class* matrix. The columns of the Projection matrix are a subset of the $C-1$ largest (non-orthogonal) eigenvectors of the matrix $J = S_W^{-1} S_B$. Finally, LDA projects the original dataset into this new subspace, taking the dot product of the Projection matrix and the standardized columns.

The number of features in the new space after the LDA execution is **53**.

To improve LDA's performance as much as possible, we tried some different **Covariance Estimators**:

- Oracle Approximating Shrinkage and Empirical Covariance were tested because they require normal distributed data, like our case of study.
- Elliptic Envelope was tested because it's robust to anomalies, requires normal and unimodal distributed data.

4.3 Kernel Approximation

Kernel approximation is a technique that approximates the feature mappings that correspond to certain kernel methods, like the ones used by the Support Vector Machine model. Kernel approximation can significantly reduce the cost of learning with very large datasets, mapping the data to a higher-dimensional space where it is easier to separate the data points.

Since the dataset is very large, the Nystroem Method was tested with RBF kernel for the SVR task, as suggested from the model documentation, and later tested on all other models.

4.4 Outlier Removal

Outliers are data points that significantly differ from other observations. They may arise due to variability in the data or errors during data collection. Removing outliers can have a substantial impact on the results of machine learning models. However, in some cases, outliers can provide valuable insights and should not be ignored.

In our case study, we do not know whether all observations are truly valuable or whether there may be some anomalies. Therefore we ran the models both with all the data and without the values that differ the most from the others, treating them as outliers.

We tried LocalOutlierFactor and IsolationForest. They're both unsupervised methods. The first one detects anomalous points by looking at how isolated the points are with respect to the surrounding neighbourhood. The second one is similar to a tree model, recursively generating partitions on the sample by randomly selecting an attribute and then randomly selecting a split value between the minimum and maximum values allowed for that attribute; the anomalous points are the ones which have a smaller path length in the tree (the threshold is the average over a forest of such random trees). The hyperparameters of the models were been chosen during a preliminary search.

4.5 Under/Over sampling

To overcome the imbalance of the dataset, we tried techniques of oversample and undersample.

Undersampling was implemented using RandomUnderSampler, setting a maximum number of samples and decrementing only the most frequent classes (years).

Oversampling was implemented using SMOTE (Synthetic Minority Over-sampling), incrementing all the classes to the number of samples of the most populated class, or incrementing by doubling the number of samples of the less populated class, or a combination of the two. SMOTE generates new samples, similar - but not equal - to its neighbours.

4.6 Best results

It follows, for each implemented model, the preprocessing techniques that we have the best results with. We did not try all the possible combinations, but we followed the most promising paths.

Model	**Preprocessing**	**Prep. parameters**	
Linear Regressor	Min-Max + LDA + Nystroem	LDA cov. estimator	OAS
		Nys. gamma	0.01
		Nys. num. components	1000
Random Forest Regressor	Min-Max + LDA	LDA cov. estimator	OAS
K-Nearest Neighbors Regressor	Min-Max + Lmax + LDA	LDA cov. estimator	OAS
Support Vector Machine Regressor	Min-Max + L1 + LDA + Nystroem	LDA cov. estimator	OAS
		Nys. gamma	0.01
		Nys. num. components	1000
Deep Neural Network	Standard Scaling + L2		
TabNet	Min-Max + Lmax + LDA	LDA cov. estimator	OAS
TabTransformer	Standard Scaling + L2		

Table 4.1: Best preprocessing techniques found

Modeling

In this section are presented short descriptions of the model implemented and the corresponding tested hyperparameters. In **Section 6** are specified the best configurations and the results obtained.

5.1 Classic Machine Learning Models

5.1.1 Linear Regression

Linear Regression assumes a linear combination between the features and the output. It finds the best-fitting line that represents the relationship between the variables.

Given x as the features and $\hat{f}$ the predicted output, the model is defined by the following linear function: $\hat{f}(x) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_p x_p$. β_0 is the intercept of the regression line, while $\beta_{1..p}$ are the slopes.

5.1.2 Random Forest Regressor

Random Forest is a supervised learning algorithm belonging to a class of models called Bootstrap aggregating ("Bagging").

During training, an ensemble of multiple independent decision trees is generated. Each tree can be built on a random subset of the training data and on a random subset of the input features. This leads to generating trees that follow different paths, trying to catch the most hidden ones among data and among the features. Therefore, it can help to avoid overfitting, sometimes at the cost of a slight increase in bias. For the regression model, the predicted output is the average of all the trees' predictions.

Here follow the tested hyperparameters and their corresponding values. To overcome overfitting, some "pruning" has been tested, as well as different proportions of samples and features.

Hyperparameters	**Values**
max_samples	0.1, 0.5, 0.66, 0.8, 1.0
n_estimators	5, 100, 200, 500
max_depth	5, 10, 15, 20, 25, None
max_leaf_nodes	50, 100, 200, 300, 400, None
ccp_alpha	0.0, 0.00001, 0.0001, 0.001, 0.01, 0.1, 1.0
max_features	0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.9, "sqrt", "log2", None

Table 5.1: Random Forest tested hyperparameters

5.1.3 K-Nearest Neighbors Regressor

The K-Nearest Neighbors (K-NN) algorithm predicts the target label by local interpolation of the target labels associated with the k nearest neighbours in the training set. Chosen a distance metric (for comparing the unlabeled sample with the training samples), the neighbour of the target can be

defined as the k nearest samples, shaping the space into different neighbours. For a regression problem, the target label is the average of the neighbour's labels.

Here follow the tested hyperparameters and their corresponding values. Of course, different values of k and different metrics has been tested. Additionally, it was tried a weighting system based on the distance: closer neighbors has a greater influence than neighbors which are further away.

Hyperparameters	**Values**
n_neighbors	5, 10, 15, 19, 20, 21, 22, 25
weights	"distance", "uniform"
metric	"cosine", "euclidean", "cityblock", "nan_euclidean"

Table 5.2: KNN tested hyperparameters

5.1.4 Support Vector Regressor

The Support Vector Regressor (SVR) algorithm uses a machine learning algorithm based on **Support Vector Machine** to train a regression model.

SVM works by finding a hyper-plane (or a set of) that best separates the data space into different classes. The **support vectors** are the data points in the dataset closest to the hyperplane. The goal is to find a hyperplane that maximally separates these support vectors, ensuring the maximum distance (called *functional margin*) between the hyperplane and the support vectors. In general, the larger the distance, the lower the generalization error of the classifier. Defined the hyperplanes, the model classifies the query point depending on whether it lies on the positive or negative side of the hyperplane depending on the classes to predict.

The SVM Regressor uses that hyperplane as the line that will be used to predict the continuous output. SVR finds the hyperplane that holds the maximum training observations within the margin ϵ (tolerance level). Too high ϵ will broaden to an imprecise model, while too low ϵ will broad to overfitting. The points outside of the margin will be the support vectors, and they dictate how the hyperplane will be directed, because SVR aims to minimize the distance between those points to the margin (in this sense the hyperplane position maximizes the distance between support vectors). The C parameter is the cost of having observations outside the margin: higher C means higher cost and can tend to overfit; smaller values can tend to high bias.

In general, when the problem isn't linearly separable, the support vectors are the samples within the margin boundaries. The non-linearity of boundaries can be overcome with a non–linear mapping by the *kernel functions*: the data are mapped into a new space, usually called *feature space*, such that a linear boundary in the feature space can correspond to a non–linear boundary in the original space. The feature space can have a number of dimensions higher than the original one. The mapping does not need to be explicitly computed, and the computation is done in the input space, avoiding an increase in the complexity.

However, since the dataset of our case study is very large, for computational reasons we opted for the linear kernel version of the algorithm (Linear SVR), preceded by the Nystroem explicit kernel mapping approximation.

Here follow the tested hyperparameters and their corresponding values. Different values of ϵ and C have been tested, as well as different cost functions.

Hyperparameters	Values
epsilon	0, 0.00001, 0.0001, 0.001, 0.01, 0.1, 1.0
C	0.1, 1, 5, 8, 9, 10, 15, 20, 100, 500
loss	"epsilon_insensitive", "squared_epsilon_insensitive"
max_iter	2000
tol	1e-5, 1e-4, 1e-3, 1e-2

Table 5.3: SVR tested hyperparameters

5.2 Deep Neural Networks

Neural Networks are a type of machine learning algorithm that consists of a set of artificial neurons that process input information, transmit it to other neurons in the network, and produce an output based on the processed data. During training, the neural network adapts to the training data by optimizing the weights of the neurons, aiming to minimize the prediction error.

The task involves the use of Feed Forward Neural Networks, where each layer moves in a single direction (there are no cycles or backward connections) and each neuron is connected to the other neurons in the subsequent layer.

The Neural Network used for the regression task has one single output layer since the output requested is a single value and not a vector of probabilities as happens for classification tasks.

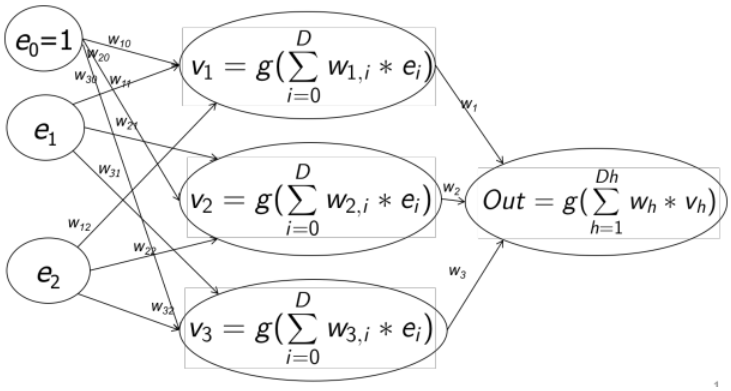

Figure 5.1: Example of Feed-Forward Neural Network

During this processing, a linear combination of weighted inputs occurs, followed by the application of a non-linear *activation function*. The choice of activation function is crucial because it introduces non-linearity into the network, allowing for modelling more complex problems and solving the vanishing gradients issue. In our study case, the **ReLU** function was applied.

The neural network used in this project was built using the PyTorch framework. As said in chapter 2, 20% of the dataset is used as a validation set to evaluate the performance of the model on unseen data during training and to choose the model parameters in order to optimize the performance on the remaining 10% test dataset.

In the process of finding the best model, different Neural Network Architectures and methods were tried. First of all, we opted to test several values of neural network depth and layer size.

Typically, feed-forward networks are trained using iterative and gradient-based optimizers, which update neuron weights to minimize the error between the expected output and the actual output of the network. The goal of the gradient descent is to find the best weights that minimize the loss

function. In our study case, principally **MSE Loss function** was used, which corresponds to the average of squared differences between the actual and the predicted values.

Since the dataset is very large, a **Stochastic Gradient Descent** was used, updating the model's parameters more frequently and therefore speeding up the gradient descent. Different sizes of batches were tested.

RMSprop and **Adam** optimizers were tested, to speed up and make more noise-clear the gradient decent.

In order to bring the Neural Network to fit well on all the data, the **Dropout** technique was tried, disabling some connections and discovering possibly different patterns. A Dropout layer was inserted between each hidden layer.

Although scaling preprocessing was applied, a layer of **Batch Normalization** was inserted after each hidden-layer node, normalizing the data before it entered the following layer. Since stochastic optimization has been used, the normalization is conducted over each mini-batch. Of course, Dropout and Batch Normalizations are not applied after the output node and were disabled at test time.

Finally, a **learning rate decay** method was applied, to bring the model to make smaller steps when it reached a local minimum, in order to decrease even more.

To avoid training the model beyond the necessary, the **early stopping** technique was used, which allows to stop the training of the model before the end of the fixed epochs, if no improvement is observed in the validation loss. In case the validation loss does not improve for a predetermined number of epochs, called **patience**, the training is interrupted and the best model reached during the training is returned. By doing so, the model is not trained beyond the necessary and the best possible model in terms of performance on the validation set is guaranteed.

Below in *5.1* is displayed the Python code of the best architecture found.

```
class NeuralNetworkDropoutBatchNorm(nn.Module):
    def __init__(self, input_size, hidden_size, depth, dropout_rate):
        super(NeuralNetworkDropoutBatchNorm, self).__init__()

        self.input_layer = nn.Sequential(
            nn.Linear(input_size, hidden_size),
            nn.BatchNorm1d(hidden_size),
            nn.ReLU(),
            nn.Dropout(dropout_rate),
        )

        self.hidden_layers = nn.Sequential(
            *[
                nn.Linear(hidden_size, hidden_size),
                nn.BatchNorm1d(hidden_size),
                nn.ReLU(),
                nn.Dropout(dropout_rate),
            ]
            * (depth - 1)
        )

        self.output = nn.Linear(hidden_size, 1)

    def forward(self, x):
        h_in = self.input_layer(x)
        h_out = self.hidden_layers(h_in)
        out = self.output(h_out)
        return out
```

Listing 5.1: Best DNN Architecture found

In *Table 5.4* are displayed the hyperparameters and their corresponding values tested in this project. In **Section 6** we will specify the best configuration and the results obtained.

Hyperparameters	**Description**	**Values**
epochs	the number of training iterations	30, 100, 200
hidden_size	the size (number of neurons) in the hidden layers	32, 64, 128, 256, 512, 1024
batch_size	how many samples per batch	8, 16, 32, 48, 64
depth	the number of hidden layers in the neural network	2, 3, 4, 5
learning_rate	the step size during gradient descent optimization	0.0001, 0.001, 0.05, 0.1
dropout_rate	the probability that some neurons are deactivated during network training	0.1, 0.2, 0.3, 0.5
criterion	the loss function	nn.MSELoss(), nn.L1Loss(), nn.HuberLoss()
early_stopper_patience	the number of epochs before stopping the training if the validation loss stops improving	30
momentum	RMSprop momentum	0.9

Table 5.4: DNN tested hyperparameters

In *Table 5.5* are listed the different parameters set for the scheduler.

Hyperparameters	**Description**	**Values**
patience	number of epochs with no improvement after which learning rate will be reduced	10, 15, 20, 21, 30, 200
threshold	threshold for measuring the new optimum, to only focus on significant changes	1
threshold_mode	one of rel, abs	abs, rel
factor	factor by which the learning rate will be reduced	0.1
min_lr	the lower bound on the learning rate	0.0000001

Table 5.5: Scheduler parameters

In *5.2* is displayed the pseudo-code of the model training.

```
def train_model(model, criterion, optimizer, epochs, train_loader, val_loader,
    scheduler, early_stopper, device, log_writer, log_name):

    n_iter = 0
    best_valid_loss = float("inf")

    # EPOCHS
    for epoch in range(epochs):
        model.train()

```

```
        # BATCHES
        for data, targets in train_loader:
            data, targets = data.to(device), targets.to(device)

            optimizer.zero_grad()  # gradient to zero

            # Forward pass
            y_pred = model(data)

            # Compute Loss
            loss = criterion(y_pred.squeeze(), targets)
            # reshape because MSELoss requires same dimensionality

            # Backward pass
            loss.backward()
            optimizer.step()

            n_iter += 1

        # Valuation
        y_test, y_pred = test_model(model, val_loader, device)
        loss_val = criterion(y_pred.squeeze(), y_test)
        scheduler.step(loss_val)

        # Save the model with best loss through the epochs
        if loss_val.item() < best_valid_loss:
            best_valid_loss = loss_val.item()
            torch.save(model.state_dict(), model_dir + log_name)

        # Early Stopping
        if early_stopper.early_stop(loss_val.item()):
            print("Early stopped!")
            break
```

Listing 5.2: DNN training function

5.3 Tabular Data

PyTorch Tabular is a powerful library that aims to simplify and popularize the application of deep learning techniques to tabular data. The models of this module are capable of handling large amounts of data and finding complex relationships between variables, improving the model's predictive ability. In this project, we were required to use **TabNet** and **TabTransformer**, which are two different models for tabular data processing in this library.

5.3.1 Data Config

PyTorch Tabular uses Pandas dataframes as the container which holds data and accepts dataframes as is, so there is no need to split the data into X and y like in Sci-kit Learn. Pytorch Tabular handles this using a DataConfig object, where you can specify this parameters:

Hyperparameters	Description
target	list of strings with the names of the target column(s)
continuous_cols	column names of the numeric fields
categorical_cols	column names of the categorical fields to treat differently (Not used in this project since the dataset doesn't contain categorical columns)

Table 5.6: DataConfig parameters

5.3.2 Optimizer Config

The optimizer plays a central role in the gradient descent process and is a critical component for training effective models. By default, PyTorch Tabular utilizes the Adam optimizer with a learning rate of 1e-3. Other settings can be specified by adding parameters to the OptimizerConfig object.

Here are the hyperparameters and their corresponding values tested in this project:

Hyperparameters	Description	Values	
		TabNet	TabTransformer
optimizer	the network optimizer	Adam	
lr_scheduler	the name of the LearningRateScheduler to use	ReduceLROnPlateau	
lr_scheduler_params	dict with the parameters for the LearningRateScheduler	patience: 7, 8, 9, 10, 11, 12 threshold: 1 threshold_mode: abs	patience: 7, 8, 9, 10, 11, 12 threshold: 1 threshold_mode: abs

Table 5.7: OptimizerConfig parameters

5.3.3 Trainer Config

Deep learning model training can become quite complex; for this reason PyTorch Tabular, built on the foundation of **PyTorch Lightning**, simplifies the entire process by leveraging the underlying PyTorch Lightning Framework. It provides an effortless way to train models while allowing users the flexibility to customize the training process according to their needs.

Here are the hyperparameters and their corresponding values tested in this project:

Hyperparameters	**Description**	**Values**	
		TabNet	**TabTransformer**
max_epochs	maximum number of epochs to be run	200	100
batch_size	number of samples in each batch of training	256, 512	512, 1024
early_stopping_mode	the direction in which the loss/metric should be optimized	min	
early_stopping_patience	the number of epochs to wait until there is no further improvements in loss/metric	10	

Table 5.8: TrainerConfig parameters

5.3.4 Model Config

TabNet

TabNet is a deep learning algorithm to process tabular data and it was developed by Google AI in 2019. TabNet takes raw data as input and uses **sequential attention** to select features to use at each decision step. This allows for better interpretation and learning, making the algorithm highly effective in solving classification and regression problems on tabular data. It employs multiple decision blocks that focus on processing a subset of input features. It employs unsupervised pre-training, initializing the model weights using knowledge from pre-trained models on large datasets or similar problems. By applying this technique to tabular data, it has demonstrated performance improvements.

The uniqueness of TabNet lies in its composition: instead of using a single neural network, the model consists of **multiple cascading neural networks**, each selectively processing data and extracting increasingly relevant and specific information. The TabNet architecture comprises three main elements: **Encoder**, **Decoder**, and **Masking**. The encoder consists of several decision units (Decision Points, DPs) that selectively process input data and produce a series of binary masks indicating which features are selected and which are discarded. The decoder, on the other hand, is a Feed-Forward neural network that uses the masks generated by the encoder for final prediction. Lastly, masking prevents the neural network from reusing the same features multiple times during data processing, leading to better model generalization and aiding in preventing overfitting.

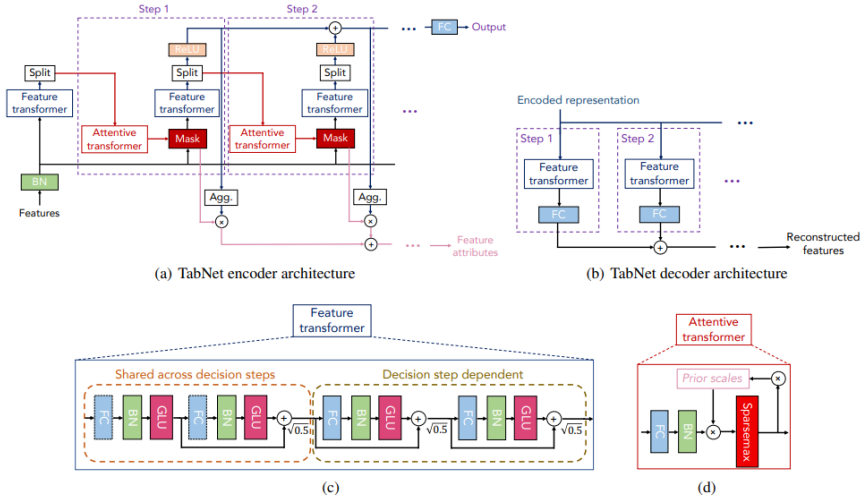

Figure 5.2: TabNet Architecture

Here are the hyperparameters and their corresponding values tested in this project. In **section 6** we will specify the best configuration and the results obtained:

Hyperparameters	**Description**	**Values**
learning_rate	the learning rate of the model	0.005
virtual_batch_size	batch size for Ghost Batch Normalization	128
n_independent	the number of independent features to use in the transformer network	2, 3
n_shared	the number of shared features to use in the transformer network	2, 3
n_step	number of successive steps in the network	3
gamma	scaling factor for attention updates	1.5
seed	the random seed to use for reproducibility	42
n_a	dimension of the attention layer	16, 32
n_d	dimension of the prediction layer	32, 64

Table 5.9: TabNet ModelConfig parameters

Below is included the code for the model training:

```
best_mse = float("inf")
best_model = None
best_params = None
iter = 0

for learning_rate, batch_size, virtual_batch_size, n_epochs, n_d, n_a, n_steps, n_independent, n_shared, gamma in params:
    iter += 1
    print(f"\nIteration: {iter} of {comb}")

    trainer_config = TrainerConfig(
        batch_size=batch_size,
        max_epochs=n_epochs,
        early_stopping_patience=10,
    )

    model_config = TabNetModelConfig(
        task="regression",
        head="LinearHead",  # Linear Head
        head_config=head_config,  # Linear Head Config
        learning_rate=learning_rate,
        virtual_batch_size=virtual_batch_size,
        n_d=n_d,
        n_a=n_a,
        n_steps=n_steps,
        n_independent=n_independent,
        n_shared=n_shared,
        gamma=gamma
    )

    # Outside this code snippet have been defined the objects data_config, optimizer_config and experiment_config
    tabular_model = TabularModel(
        data_config=data_config,
        model_config=model_config,
        optimizer_config=optimizer_config,
        trainer_config=trainer_config,
        experiment_config=experiment_config,
        verbose=False
    )

    tabular_model.fit(train=train, validation=val)
    tabular_model.evaluate(test)

    y_pred = tabular_model.predict(X_test)
    mse = mean_squared_error(y_test, y_pred)
    r2 = r2_score(y_test, y_pred)

    if mse < best_mse:
        best_mse = mse
        best_model = copy.deepcopy(tabular_model)
        best_params = (learning_rate, batch_size, virtual_batch_size, n_epochs, n_d, n_a, n_steps, n_independent, n_shared, gamma)
        print("Best model updated")
```

Listing 5.3: Fine-Tuning loop for TabNet

TabTransformer

TabTransformer is a **transformer-based** model designed for supervised learning that uses self-attention to capture the dependencies between features and is known for its ability to handle sequential data. It builds upon the foundation of self-attention-based Transformers. The Transformer layers play a crucial role in transforming the embeddings of categorical features into robust contextual embeddings. These embeddings enhance prediction accuracy by capturing relevant information from the input data. They are also highly robust against both missing and noisy data features, providing better interpretability. TabTransformer performs well in machine learning competitions and can be used for regression, classification (both binary and multiclass), and ranking problems.

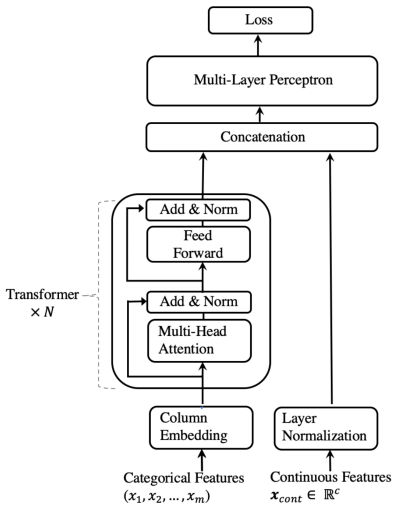

Figure 5.3: TabTransformer Architecture

Here are the hyperparameters and their corresponding values tested in this project. In section 6 we will specify the best configuration and the results obtained:

Hyperparameters	Description	Values
virtual_batch_size	batch size for Ghost Batch Normalization	64, 128
learning_rate	the learning rate of the model	0.01
num_heads	the number of heads in the Multi-Headed Attention layer	8
num_attn_blocks	the number of layers of stacked Multi-Headed Attention layers	6
transformer_activation	the activation type in the transformer feed-forward layers	'ReGLU', 'GEGLU', 'SwiGLU'

Table 5.10: TabTransformer ModelConfig parameters

Below is included the code for the model training:

```
best_mse_tt = float("inf")
best_model_tt = None
best_params_tt = None
iter = 0

for learning_rate, batch_size, epochs, virtual_batch_size, num_heads,
num_attn_blocks, transformer_activation in params:
    iter += 1
```

```
        print(f"\nIteration: {iter} of {comb}")
        trainer_config = TrainerConfig(batch_size=batch_size, max_epochs=epochs,
    early_stopping_patience=10, load_best=True)

        model_config = TabTransformerConfig(
            task="regression",
            head="LinearHead",  # Linear Head
            head_config=head_config,  # Linear Head Config
            loss="MSELoss",
            seed=random_state,
            learning_rate=learning_rate,
            virtual_batch_size=virtual_batch_size,
            num_heads=num_heads,
            num_attn_blocks=num_attn_blocks,
            ff_hidden_multiplier=64,
            transformer_activation=transformer_activation,
        )

        # Outside this code snippet have been defined the objects data_config,
    optimizer_config and experiment_config
        tabular_model = TabularModel(
            data_config=data_config,
            model_config=model_config,
            optimizer_config=optimizer_config,
            trainer_config=trainer_config,
            experiment_config=experiment_config,
        )

        tabular_model.fit(train=train, validation=val)
        tabular_model.evaluate(test)

        y_pred = tabular_model.predict(X_test)
        mse = mean_squared_error(y_test, y_pred)
        mae = mean_absolute_error(y_test, y_pred)
        r2 = r2_score(y_test, y_pred)

        if mse < best_mse_tt:
            best_mse_tt = mse
            best_model_tt = copy.deepcopy(tabular_model)
            best_params_tt = (
                learning_rate,
                batch_size,
                epochs,
                virtual_batch_size,
                num_heads,
                num_attn_blocks,
                transformer_activation,
            )
            print("Best model updated")
```

Listing 5.4: Fine-Tuning loop for TabTransformer

Performance Evaluation

In the following paragraph, we will examine the results obtained from various models used in the Modeling section. To measure and compare model performance, two metrics were employed:

- **Mean Squared Error (MSE):** this metric calculates the average of the squared errors between the model's predicted values and the actual observed values in the test data. A low MSE value indicates that the model has good predictive capability for real values, while a high value suggests poor prediction ability. The formula for MSE is as follows:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

  where:

  - n represents the number of samples in the dataset.
  - y_i are the observed values for sample i.
  - $\hat{y}_i$ are the predicted values by the model for sample i

- **Coefficient of Determination** (R^2)**:** this metric measures how closely the variance of the model's predicted data aligns with the variance of the actual data. It can take values between 0 and 1, where 1 indicates a perfect fit of the model to the real output data, and 0 indicates that the model fails to explain the variation in the output data. The formula for calculating this score is as follows:

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$$

  where:

  - n represents the number of samples in the dataset.
  - y_i are the observed values for sample i.
  - $\hat{y}_i$ are the predicted values by the model for sample i
  - $\bar{y}$ is the mean of the observed values.

Below, we will present all the fine-tuning results of the models discussed in the previous sections, with a focus on the optimal configurations. Each model was trained and tested firstly on raw data and secondly applying preprocessing techniques.

6.1 Classic Machine Learning Models

6.1.1 Linear Regression

The best preprocessing pipeline is the following:

```
pipeline = Pipeline(
    steps=[
        ("min-max", preprocessing.MinMaxScaler()),
        ("lda", LinearDiscriminantAnalysis(solver="eigen", shrinkage=None,
covariance_estimator=OAS())),
        ("nys", Nystroem(gamma=0.01, n_components=1000)),
    ]
)
```

Listing 6.1: Pipeline for Linear Regression

The results obtained are:

- **MSE:** 73.598
- **R2 score:** 0.331

The kernel function for the mapping was left by default the Radial Basis Function.

6.1.2 Random Forest Regressor

The best hyperparameter configuration is the following:

```
pipeline = Pipeline(
    steps=[
        ("min-max", preprocessing.MinMaxScaler()),
        ("lda", LinearDiscriminantAnalysis(solver="eigen", shrinkage=None,
covariance_estimator=OAS())),
    ]
)
```

Listing 6.2: Pipeline for Random Forest Regressor

Hyperparameters	**Values**
max_samples	0.66
n_estimators	200
max_depth	25
max_leaf_nodes	None
ccp_alpha	0.01
max_features	0.4

Table 6.1: Random Forest best configuration

The results obtained by this configuration of hyperparameters are:

- **MSE:** 73.903
- **R2 score:** 0.329

A higher number of estimators gives slightly better results but at the cost of a significant increase in the model's size in terms of storage memory.

6.1.3 K-Nearest Neighbors Regressor

The best hyperparameter configuration is the following:

```
pipeline = Pipeline(
    steps=[
        ("min-max", preprocessing.MinMaxScaler()),
        ("lmax", preprocessing.Normalizer(norm="max")),
        ("lda", LinearDiscriminantAnalysis(solver="eigen", shrinkage=None,
covariance_estimator=OAS())),
    ]
)
```

Listing 6.3: Pipeline for KNN

Hyperparameters	Values
n_neighbors	21
weights	"distance"
metric	"cosine"

Table 6.2: KNN best configuration

The results obtained by this configuration of hyperparameters are:

- **MSE:** 73.656
- **R2 score:** 0.331

However, applying a preprocessing pipeline in the form of Min-Max + L2 + LDA + Nystroem, the results are slightly better (R2 score: 0.336), but the model increases in size to almost 2GB. That's because KNN stores the training data in memory and by applying a Nystroem kernel approximation the data increases dimensionality, hence the size of the KNN model.

6.1.4 Support Vector Regressor

The best preprocessing pipeline is the following:

```
pipeline = Pipeline(
    steps=[
        ("min-max", preprocessing.MinMaxScaler()),
        ("l1", preprocessing.Normalizer(norm="l1")),
        ("lda", LinearDiscriminantAnalysis(solver="eigen", shrinkage=None,
covariance_estimator=OAS())),
        ("nys", Nystroem(gamma=0.01, n_components=1000)),
    ]
)
```

Listing 6.4: Pipeline for Support Vector Regressor

The results obtained are:

- **MSE:** 78.721
- **R2 score:** 0.285

The kernel function for the mapping was left by default the Radial Basis Function.

Hyperparameters	Values
epsilon	0.01
C	0.1, 1, 5, 8, 9, 10, 15, 20, 100, 500
loss	"epsilon_insensitive"
max_iter	2000
tol	0.0001

Table 6.3: SVR best configuration

6.2 Deep Neural Networks

The best preprocessing pipeline is the following:

```
    pipeline = Pipeline(
        steps=[
            ("std", preprocessing.StandardScaler()),
            ("l2", preprocessing.Normalizer(norm="l2")),
        ]
    )
```

Listing 6.5: Pipeline for DNN

Hyperparameters	Values
epochs	30
hidden_size	256
batch_size	48
depth	2
learning_rate	0.0001
dropout_rate	0.1
criterion	MSELoss()
early_stopper_patience	30
RMSprop momentum	0.9
scheduler patience	21
scheduler threshold	1
scheduler threshold_mode	abs
scheduler factor	0.1
scheduler min_lr	0.0000001

Table 6.4: DNN best configuration

The results obtained are:

- **MSE:** 70.935
- **R2 score:** 0.356

With MinMax preprocess (with or without any L-Normalization) the model can't descend the local minimum of MSE=110.

Adam optimizer makes the descend too fast, ending in a local minimum. RMSprop, on the contrary, has a more noisy descent but performs better.

Increasing the number of layers and their sizes, the model performs much worse.

Looking at the trend of the loss function over the validation set, we inserted a scheduler that decreases the learning rate at a specific point, keeping the loss function around that local minimum, and preventing it from increasing in value.

The DNN with only Dropout or with only Batch Normalization performs worse. The combination of the two is the best choice.

6.3 Tabular Data

6.3.1 TabNet

The TabNet model was trained by searching for the best configuration by iterating over the possible values of the hyperparameters illustrated in section 6. Below we will illustrate the best combination and the results derived from the training of the model. In this case, the following pipeline was applied as preprocessing:

```
    pipeline = Pipeline(
        steps=[
            ("min-max", preprocessing.MinMaxScaler()),
            ("lmax", preprocessing.Normalizer(norm="max")),
            ("lda", LinearDiscriminantAnalysis(solver="eigen", shrinkage=None,
    covariance_estimator=OAS())),
        ]
    )
```

Listing 6.6: Pipeline for TabNet

Hyperparameter	**Value**
learning_rate	0.005
batch_size	256
virtual_batch_size	128
n_epochs	200
n_d	64
n_a	16
n_steps	3
n_independent	2
n_shared	2
gamma	1.5

Table 6.5: TabNet Best configuration

The results obtained by this configuration of hyperparameters are:

- **MSE:** 70.756
- **R2 score:** 0.357

6.3.2 TabTransformer

The TabTransformer model was trained by searching for the best configuration by iterating over the possible values of the hyperparameters illustrated in section 6. Below we will illustrate the best combination and the results derived from the training of the model. In this case, the following pipeline was applied as preprocessing:

```
    pipeline = Pipeline(
        steps=[
            ("std", preprocessing.StandardScaler()),
            ("l2", preprocessing.Normalizer(norm="l2")),
        ]
    )
```

Listing 6.7: Pipeline for TabTransformer

Hyperparameter	**Value**
learning_rate	0.01
batch_size	512
virtual_batch_size	128
n_epochs	100
num_heads	8
num_attn_blocks	6
transformer_activation	ReGLU

Table 6.6: TabTransformer Best configuration

The results obtained by this configuration of hyperparameters are:

- **MSE:** 78.044
- **R2 score:** 0.291

6.4 Additional notes

PCA underperformed with respect to LDA, which even gives one less feature in the new data space (53 instead of 54 components). As mentioned in *4.2.1*, PCA works on the correlation between the various features and in our study case they are very low correlated, while LDA also considers the class separability of the data.

For all the models, both undersampling and oversampling (and the combination of the two) led to less accurate results. That is probably because the data within each class is not very similar: undersampling the most frequent classes resulted in less accurate models for predicting the test set (filled mostly of the same highly frequent classes), while oversampling the less frequent classes resulted in a probable overfitting on those classes, badly predicting the test samples. Here we remind that the "stratify" option of the train/val/test split was set, therefore in the test set are surely some samples of all the classes.

For all the models, detecting and removing the outliers underperformed the results. Were implemented two methods: removing the outliers by fitting the detection model on the entire train set and by fitting one model for each class (Year) of the train set. The latter was tried in order to prevent the samples of the least frequent classes from being identified as outliers, being of a much smaller number than the samples of the other classes.

Conclusions

The results obtained from various models demonstrate that, in general, all the models behave quite similarly. Among all, those that achieve the best results are Deep Neural Networks and TabNet, indicating that for the assigned task neural networks appear to perform the best. However, all other models have still shown results in line with the top performers, with a slight deterioration, but not with extreme variations.

In *7.1* there is a quick recap of all the results.

Model	**MSE**	**R2**
Linear Regression	73.598	0.331
Random Forest Regression	73.903	0.329
K-Nearest Neighbors Regression	73.656	0.331
Support Vector Regression	78.721	0.285
Deep Neural Networks	70.935	0.356
TabNet	70.756	0.357
TabTransformer	78.044	0.291

Table 7.1: Final best results