Machine Learning

Development project - December 2022

Regression - Boston housing and prostate cancer datasets

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Eustache LE BIHAN, Badr SOULAIMANI, Mohssine RAZOKI, Joel IRIE

Summary

I - Context and datasets

We have two provided datasets. Both of them are meant for regression purposes. The first dataset is the Boston housing dataset and the second one is the Prostate cancer dataset.

i) Boston housing dataset

The Boston housing dataset contains 506 observations and 14 variables. It is derived from information collected by the U.S. Census Service concerning housing in the area of Boston MA. The following describes the dataset features (columns):

- CRIM - per capita crime rate by town
- ZN - proportion of residential land zoned for lots over 25,000 sq.ft.
- INDUS - proportion of non-retail business acres per town.
- CHAS - Charles River dummy variable (1 if tract bounds river; 0 otherwise)
- NOX - nitric oxides concentration (parts per 10 million)
- RM - average number of rooms per dwelling
- AGE - proportion of owner-occupied units built prior to 1940
- DIS - weighted distances to five Boston employment centres
- RAD - index of accessibility to radial highways
- TAX - full-value property-tax rate per $10,000
- PTRATIO - pupil-teacher ratio by town
- B - 1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town
- LSTAT - % lower status of the population
- MEDV - Median value of owner-occupied homes in $1000's

The target variable we are trying to predict is the variable **MEDV** from the 13 others.
The dataset is noisy(it contains NaN values) and therefore needs cleaning.

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1	296	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2	242	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3	222	18.7	396.90	NaN	36.2

fig. 1: Sample of 5 raws of the dataset

Let's get the number of missing values per attributes:

```
(house_data.isna().sum()).sort_values(ascending=False)
```

```
CRIM       20
ZN         20
INDUS      20
CHAS       20
AGE        20
LSTAT      20
NOX         0
RM          0
DIS         0
RAD         0
TAX         0
PTRATIO     0
B           0
MEDV        0
dtype: int64
```

fig. 2: Number of NaN values per variable

As we can see, variables *CRIM*, *ZN*, *INDUS*, *CHAS*, *GE* and *LSTAT* have the most missing values. We will replace these missing values with the mean of the corresponding feature (columns).

Now we want to look at the spread and skewness of numerical data through their quartiles:

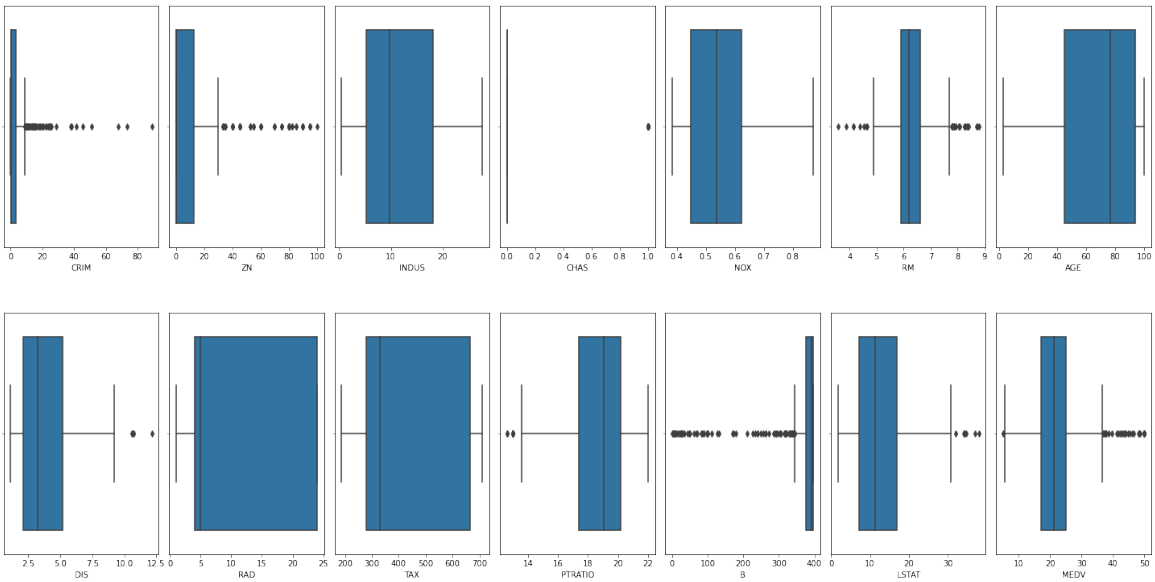

fig. 3: Boxplot of the dataset

This boxplot shows that Columns like *CRIM*, *ZN*, *RM*, and *B* seem to have outliers (high values) that can affect the model prediction. We will solve this by standardizing the dataset before applying regression models.

ii) Prostate cancer dataset

This dataset is composed of 97 observations of 9 variables and an extra one being a train/test indicator. There are 9 attributes:

- lcavol
- lweight
- age
- lbph
- svi
- lcp
- gleason
- pgg45
- lpsa

Here, we are trying to predict the variable **lpsa** from the 8 others.

	Unnamed: 0	lcavol	lweight	age	lbph	svi	lcp	gleason	pgg45	lpsa	train
0	1	-0.579818	2.769459	50	-1.386294	0	-1.386294	6	0	-0.430783	T
1	2	-0.994252	3.319626	58	-1.386294	0	-1.386294	6	0	-0.162519	T
2	3	-0.510826	2.691243	74	-1.386294	0	-1.386294	7	20	-0.162519	T
3	4	-1.203973	3.282789	58	-1.386294	0	-1.386294	6	0	-0.162519	T
4	5	0.751416	3.432373	62	-1.386294	0	-1.386294	6	0	0.371564	T

fig. 4: Sample of 5 raws of the dataset

iii) Data preprocessing

For each dataset, the first steps to do were there cleaning and preprocessing. It consists of making the data exploitable to train the model on. In the preprocessing of the data, the goal was to transform the data into a shape that a model can train on. To do so we need to replace missing values, and centre and normalize the data. We began by replacing the missing values with the mean of the corresponding feature as all the dataset contains numerical values.

CHAS	NOX	RM
0.0	0.458	7.147
0.0	0.458	6.430
NaN	0.524	6.012
0.0	0.524	6.172
0.0	0.524	5.631
NaN	0.524	6.004

➡

CHAS	NOX	RM
0.000000	0.458	7.147
0.000000	0.458	6.430
0.069959	0.524	6.012
0.000000	0.524	6.172
0.000000	0.524	5.631
0.069959	0.524	6.004

fig. 5: Replacing missing values (NaN)

For the normalization step, we used the zero mean normalization method, which means values were normalized based on the mean and standard deviation of the data.

	CRIM	ZN	INDUS
count	486.000000	486.000000	486.000000
mean	3.611874	11.211934	11.083992
std	8.720192	23.388876	6.835896
min	0.006320	0.000000	0.460000
25%	0.081900	0.000000	5.190000
50%	0.253715	0.000000	9.690000
75%	3.560263	12.500000	18.100000
max	88.976200	100.000000	27.740000

	CRIM	ZN	INDUS
count	5.060000e+02	5.060000e+02	5.060000e+02
mean	2.896234e-17	-4.234645e-17	5.803439e-17
std	1.000000e+00	1.000000e+00	1.000000e+00
min	-4.219109e-01	-4.891544e-01	-1.585868e+00
25%	-4.129106e-01	-4.891544e-01	-8.798099e-01
50%	-3.886863e-01	-4.891544e-01	-1.767372e-01
75%	7.494596e-18	4.085718e-17	1.047296e+00
max	9.989073e+00	3.873647e+00	2.486281e+00

fig. 6: Data normalization

iv) Train/test split

To evaluate the performance of a training model, we need to test it on a test dataset different from the one it has been trained on. For the Boston housing Dataset, we split randomly the dataset with 70% of the observation for the training and 30% for testing. For the Prostate Cancer dataset, the train/test split is already set by the feature « train ».

v) Feature selection

We perform feature selection to compress our data and remove any sort of linearity between the different components. To do so, we use the Principal Component Analysis (*PCA*) algorithm to protect our data in another space. We then choose the number of components based on a set percentage of explained variance of the data. It is what we call threshold variance. We leave the choice of that parameter to the user. By default, the parameter is set to 0.95, meaning that we will choose the number of components that will explain 95% or more of the data's variance.

Results: Running *PCA* on our two datasets results in using only 9 out of 13 features for the housing data set and 6 out of the 8 features for the prostate dataset. Both of these results are with the variance threshold set to 95%.

II - Models: choice, parameters and training

i) Models

Since we have a regression problem, many models can do this task in the *scikit-learn* library. We decided to implement a number of them, notably those that we have seen during lectures and labs, and give the choice to the user to select what model should be used for the regression.

There are 6 implemented models in the project: **Linear Regression - Support Vector Machines Regression - Stochastic Gradient Descent Regression - Random Forest Regression - AdaBoost Regression - Multi-Layer Perceptron Regression**.
However, we made sure our code is scalable. So, adding another model and implementing it should be easy as long as it is available in the *scikit-learn* library.

ii) parameters

As for the parameters to use for the selected model, we let the user decide by passing them in a grid-like dictionary depending on the model used. The passed dictionary should look something like this for a Support Vector Machine Regression for example:

```
{'kernel':['linear', 'rbf', 'sigmoid', 'poly'],
                    'C'     :[1, 10],
                    'degree': [2, 3],
                    'gamma' : ['scale', 'auto']
                    }
```

fig. 7: Model parameters, an example

If only one value is passed for each parameter, the model returned uses the specified values. If multiple values are specified, we run a grid search to determine which are the most optimal parameters from the possible combinations. We rank the different models based on the mean squared error (*MSE*) between the predictions and the ground truth.

iii) training

To train the model, we use a cross-validation technique with the number of splits being set by the user. We also perform a grid search if there is more than one value for a parameter. In the end, we return the training time (with grid search), the model with the best results, its score and its parameters.

```
Dataset :  HousingData.csv
Model :  SVMRegressor
Training time:  5.9029412269592285      best score:   0.17842193710013873
  best params:  {'C': 1, 'degree': 2, 'gamma': 'scale', 'kernel': 'rbf'}
```

fig. 8: an example of training result

III - Results and good programming practices

To do this group project we used the *git* version control system via *GitHub*.Moreover, we followed deep learning good practices to structure our project (see resources in the *readme*). The project file structure is the following :

```
ml-project
├── data
│   └── raw                   <- Raw datasets are stored here.
│
├── src
│   ├── __init__.py           <- Makes src a python module.
│   │
│   ├── data
│   │   └── make_dataset.py   <- Script to preprocess datasets.
│   │
│   ├── models
│   │   └── train_model.py    <- Script to train a model.
│   │
│   └── visualization
│       └── performances.py    <- Script to train evaluate linear regression performances.
│
├── tests
│   └── make_dataset_test.py  <- Unit test to check data preprocessing.
│
├── definitions.py            <- Define project variables such as ROOT_DIR.
│
├── main.ipynb                <- Notebook to run the code.
│
├── main.py
│
└── README.md
```

fig. 9: project struture

Another good programming practice is **documentation** which is very important to both be able to understand the code, use it and also maintain it if any features want to be added or if a problem has to be fixed. This includes the documentation of the different functions (Description of the function - inputs - outputs) as well as the commentaries inside each function.

Another aspect we focused on is code **scalability**. It is very important that our code can be scalable and allow for new features to be added easily, especially that with a regression problem, many models can be used as a solution. The code is not implemented for the use of a single model and any new model can be added by adding it to the *build_model* function in *train.py* as long as it implements the estimator interface in *scikit-learn*.

i) unit test

We wrote a test to check that our data preprocessing was working as expected, which means verifying that after storing data in a data frame, there are no remaining empty values but also that the mean of each feature is 0 and the standard deviation is 1.
Here's the output of this test (make_dataset_test.py available in ./tests). We can see that the former conditions are matched.

```
--------------------------------------------------------------------------------
prostate.data.txt
Number of NaN values (should be 0) : 0
            lcavol       lweight  ...         pgg45          lpsa
mean  3.433679e-17  6.947478e-16  ...  1.630998e-17  4.669804e-16
std   1.000000e+00  1.000000e+00  ...  1.000000e+00  1.000000e+00

[2 rows x 9 columns]
--------------------------------------------------------------------------------
HousingData.csv
Number of NaN values (should be 0) : 0
              CRIM            ZN  ...         LSTAT          MEDV
mean -5.616939e-17 -5.792468e-17  ... -1.404235e-16 -5.476515e-16
std   1.000000e+00  1.000000e+00  ...  1.000000e+00  1.000000e+00
```

fig. 10: unit test for import_clean_data and tandardizer functions

ii) results

The implementations of the training and the validation of the different models can be found in the main.ipynb notebook.

- First of all, we tried a basic linear regression. Such a model has a low coefficient of determination meaning (approximately 0.5 for the prostate cancer dataset, 0.65 for the Boston housing dataset) which means that the relationship between variables does not seem to be linear.
- Then, we trained different regression models implemented in the scikit-learn library: SVMRegressor, SGDRegressor, RandomForestRegressor, AdaBoostRegressor, and an MLPRegressor. All these models were trained using cross-validation and a grid search is also performed on the different values given as parameters to return the model with the most optimal parameters.

- Finally, we run a validation on the model with the most optimal parameters for each type of regression and show the results to the user. The score used is the mean squared error between the predictions and the ground truth.

```
Model name :  LinearRegressor     Score :  0.3458320392206669
Model name :  SVMRegressor     Score :  0.5862111924436271
Model name :  SGDRegressor     Score :  0.3555097947304952
Model name :  RandomForestRegressor     Score :  0.4608076463033218
Model name :  AdaBoostRegressor     Score :  0.4282445405740059
Model name :  MLPRegressor     Score :  0.3465805868306671
```

fig. 11: Results on the test set for the Boston housing dataset

```
Model name :  LinearRegressor     Score :  0.3079402633320044
Model name :  SVMRegressor     Score :  0.172301175691453
Model name :  SGDRegressor     Score :  0.305394546888522
Model name :  RandomForestRegressor     Score :  0.22333824800001228
Model name :  AdaBoostRegressor     Score :  0.2841535220885037
Model name :  MLPRegressor     Score :  0.1594640236929301
```

fig. 12: Results on the test set for the prostate cancer dataset

We can see that:

- For the prostate cancer dataset, the most performant model is surprisingly the linear one. Nevertheless, the MLP regressor reaches the same result but with a longer training time (see notebook main. ipynb)
- For the Boston Housing dataset, the most performant model is the MLP regressor, but also the longest to train.

III - Conclusion

To conclude, we developed a project to train a model/several models and test it/them on the two given datasets. We also implement good coding practices such as a project versioning thanks to git, code documentation and machine learning-oriented project structure.