Detailed resolution of the wine quality prediction

Paulo S. A. Sousa

2023-03-14

Abstract

A prediction model based on the linear regression model will be developed, to predict the quality of wines based on their chemical composition.

To predict the quality of a new product using the respective chemical composition may help companies to develop new successful products.

A detailed discussion about `sklearn` pipelines will presented.

1 Introduction

The Python code structure to create prediction models is similar across a great variety of prediction models – the procedure will be, in essence, repeated when other models, beyond the linear regression model, are used. Therefore, it is extremely important that students learn and understand all the involved details. That is the reason why I am writing these notes for you.

To create a prediction model, some steps need to be taken in proper sequence. Pipelines of `sklearn` can greatly help us, as, by using them, we will be able to automate a substantial part of the needed work, making our life a lot easier.

2 Reading the dataset to Python

To read the dataset to Python, we need to use `pandas`. Hence, we need to load `pandas` before reading the dataset:

```
import pandas as pd
```

After having loaded `pandas`, we can read the dataset to a `pandas` dataframe:

```
# You must use the appropriate path to your CSV file.
# We need to use sep=';', because this file uses ';' as column
 ↪  separator
```

```
# (and not ',' as usual).
df = pd.read_csv(
    'datasets/winequality-red.csv',
    sep=';')
```

```
      fixed acidity  volatile acidity  citric acid  residual sugar  chlorides  \
0               7.0              0.27         0.36            20.7      0.045
1               6.3              0.30         0.34             1.6      0.049
2               8.1              0.28         0.40             6.9      0.050
3               7.2              0.23         0.32             8.5      0.058
4               7.2              0.23         0.32             8.5      0.058
...             ...               ...          ...             ...        ...
4893            6.2              0.21         0.29             1.6      0.039
4894            6.6              0.32         0.36             8.0      0.047
4895            6.5              0.24         0.19             1.2      0.041
4896            5.5              0.29         0.30             1.1      0.022
4897            6.0              0.21         0.38             0.8      0.020

      free sulfur dioxide  total sulfur dioxide  density    pH  sulphates  \
0                    45.0                 170.0  1.00100  3.00       0.45
1                    14.0                 132.0  0.99400  3.30       0.49
2                    30.0                  97.0  0.99510  3.26       0.44
3                    47.0                 186.0  0.99560  3.19       0.40
4                    47.0                 186.0  0.99560  3.19       0.40
...                   ...                   ...      ...   ...        ...
4893                 24.0                  92.0  0.99114  3.27       0.50
4894                 57.0                 168.0  0.99490  3.15       0.46
4895                 30.0                 111.0  0.99254  2.99       0.46
4896                 20.0                 110.0  0.98869  3.34       0.38
4897                 22.0                  98.0  0.98941  3.26       0.32

      alcohol  quality
0         8.8        6
1         9.5        6
2        10.1        6
3         9.9        6
4         9.9        6
...       ...      ...
4893     11.2        6
4894      9.6        5
4895      9.4        6
4896     12.8        7
4897     11.8        6

[4898 rows x 12 columns]
```

3 Preprocessing of data

The preprocessing of data aims to prepare the data to be used by the model. In general, it involves multiple steps. To automate the preprocessing stage, we will use `sklearn` pipelines.

The `sklearn` functions need the original dataset separated into two dataframes:

- `X`, which must contain *all* the predictors.
- `y`, which must contain *only* the outcome variable.

Let us do that:

```
X = df.drop('quality', axis=1) # axis=1 is to mean we will drop a
 ↪  column and not a row
y = df['quality']
```

3.1 Split of data into train and test sets

Prediction is useless if aimed to predict the past! Prediction is needed to predict cases still unseen. However, we only have data from the past, and we do not know whether our prediction model will work well in unseen cases. Thus, to overcome this difficulty, we will *randomly* split the data into *train* and *test* sets. With the train set, we will create the model. To get an idea about how the model will work with unseen cases, we will use the test set.

It is usual to use 80% and 20% of the initial dataset as train and test sets, respectively.

The split can be easily done with the `sklearn` function `train_test_split`:

```
from sklearn.model_selection import train_test_split # to load the
 ↪  train_test_split function

X_train, X_test, y_train, y_test = train_test_split(X, y,
 ↪  test_size=0.2, random_state=45) # test_size is to use 20% of the
 ↪  initial dataset as test set
```

We do not really need to use `random_state=45`. However, to make our results reproducible, we do use it, given the random nature of the splitting process. In fact, the elements that go into train and test sets are randomly chosen, but the random choice of elements will be always the same, if we fix the `random_state` with an integer. We used the number 45, but we could have used any integer number.

3.2 Data scaling

The predictor `total sulfur dioxide` assumes large values, while predictor `chlorides` assumes very tiny values. This asymmetry of scales may impact negatively on the predictive performance of our model. Consequently, we need to reduce all predictors to a similar scale. Basically, for each predictor, we need to subtract all values by its mean and, subsequently, divide the resulting values by the standard deviation of the predictor.

To give an example, we will consider predictor `total sulfur dioxide`:

```
scaled_predictor = (df['total sulfur dioxide'] - df['total sulfur
 ↪  dioxide'].mean()) / df['total sulfur dioxide'].std()
```

This procedure will scale the predictor `total sulfur dioxide`, which now has mean equal to 0 and standard deviation equal to 1.

If we apply this calculation to all columns, all columns will be equally scaled: All with mean equal to 0 and standard deviation equal 1.

`sklearn` has a function that can scale a predictor (in the just described way): `StandardScaler`. Although simplifying, function `StandardScaler` would need to be applied to *all* predictors – and that is not practical, as, usually, we have a lot of predictors!

The solution is to use `sklearn` pipelines.

A pipeline is a group of *sequential* steps that can be executed together by calling the pipeline name.

Each step of a pipeline, when we use `Pipeline` function to create the pipeline, *needs* a name. Why does a step need to be named? Because, often, we need to retrieve the results of a step and, therefore, we need to use the name of the step to get its results.

In the prediction model we are creating, we do not need named steps, because we do not need to retrieve the results of any step. Hence, we do not really need to use meaningful step names – we can use whatever names come to our mind!

There is another function to create pipelines: `make_pipeline`. This `make_pipeline` function creates pipelines with steps with *no* names. We could have used `make_pipeline` instead. However, since we will meet cases in which we need to retrieve the results of intermediate steps, I decided to introduce pipelines with named steps.

To use `StandardScaler`, we need to load it before using it:

```
from sklearn.preprocessing import StandardScaler
```

To use `Pipeline` function, we also need to load it:

```
from sklearn.pipeline import Pipeline
```

To create a pipeline with the scaler:

```
scaler = Pipeline([
    ('any_name1', StandardScaler())
    ])
```

Notice that the scaler, up to now, is just defined and *not* applied to any column!

Before the scaler is applied to any column, we need to tell Python which columns we want to be scaled. To accomplish that, we need to use `ColumnTransformer` and, therefore, we need to load it before using it:

```
from sklearn.compose import ColumnTransformer
```

As discussed before, we need to apply the scaler to all predictors. To get the names of all predictors, we can use the column names of `X`:

```
X.columns.to_list() # We use to_list() to get the names as a list
```

```
['fixed acidity',
 'volatile acidity',
 'citric acid',
 'residual sugar',
 'chlorides',
 'free sulfur dioxide',
 'total sulfur dioxide',
 'density',
 'pH',
 'sulphates',
 'alcohol']
```

Now, we can tell Python that we want all predictors to be scaled:

```
preprocessor = ColumnTransformer([
    ('any_name_does', scaler, X.columns.to_list())],
    remainder='passthrough')
```

The column transformation is just defined and still to be executed. *No* column has been transformed up to now!

Notice that we are using `scaler` pipeline inside column transformer `preprocessor`.

The parameter `remainder='passthrough'` is to tell the column transformer to pass through the transformed columns. Otherwise, the column would be transformed but not passed through to the model!

Finally, we need to create a final pipeline to run, in sequence, the `preprocessor` column transformer (that, in turn, uses the `scaler` pipeline) and the linear regression model.

To load the function that creates the linear regression model:

```
from sklearn.linear_model import LinearRegression
```

And the final pipeline can be created as follows:

```
pipe = Pipeline([
    ('pre', preprocessor),
    ('lm', LinearRegression())])
```

Again, the pipeline is created, but not executed! This means that no scaling has been done. Moreover, the linear regression model has not yet been applied. Why? Because all pipelines and transformers have just been defined, but not executed: The execution of the pipelines is *triggered* by function `fit` as follows:

```
pipe.fit(X_train, y_train)
```

Now, to make predictions for the outcome variable:

```
y_pred = pipe.predict(X_train)
y_pred
```

```
array([6.48553273, 4.72694379, 5.84088531, ..., 5.48660099, 6.34047368,
       5.4922784 ])
```

4 Predictive performance of the model

We have used the train set to make predictions. Now, we need to evaluate the quality of the predictions. To exemplify, we can use the first rows of the train set and the respective predictions:

	y_train	y_pred	absolute error
0	7	6.49	0.51
1	5	4.73	0.27
2	6	5.84	0.16
3	6	6.50	0.50
4	7	6.43	0.57

As we can see, the absolute error is kind of acceptable! To have an overall perspective, we can take the mean of the absolute error corresponding to predictions for *all* elements of the train set. To do that, we could use the usual Python functions to calculate the mean absolute error. Fortunately, `sklearn` offers us a function that already computes the mean absolute error: Function `mean_absolute_error`. We can load this function with:

```
from sklearn.metrics import mean_absolute_error
```

To calculate the mean absolute error:

```
mae = mean_absolute_error(y_train, y_pred)
mae
```

```
0.5859156936493962
```

To print the mean absolute error with a nicer format:

```
print(f'MAE= {mae}')
```

```
MAE= 0.5859156936493962
```

With f-strings (strings like `f''`), the variables inside curly brackets will be replaced with their current values. Thus, when printing, the variable `mae` will be replaced with its current value.

To set the decimals to 4 digits, we can add `:0.04f`:

```
print(f'MAE= {mae:0.04f}')
```

```
MAE= 0.5859
```

If we wanted only 3 decimals, we could use `:0.03f`.

Since MAE is 0.5859, on average, the error is much below one quality unit. Not bad.

How can a company use this model to produce a new wine? Well, the estimates of the parameters can be obtained:

```
pipe.named_steps['lm'].coef_
```

```
array([ 0.05327132, -0.18799844, -0.00285321,  0.39110592, -0.00739712,
        0.07184577, -0.00867385, -0.41388714,  0.10288324,  0.06487277,
        0.25983798])
```

We can form a dataframe with the coefficients and their corresponding names:

```
results = pd.DataFrame({
  'Coefficients': X.columns,
  'Predictions': pipe.named_steps['lm'].coef_
```

```
})
```

	Coefficients	Predictions
0	fixed acidity	0.053271
1	volatile acidity	-0.187998
2	citric acid	-0.002853
3	residual sugar	0.391106
4	chlorides	-0.007397
5	free sulfur dioxide	0.071846
6	total sulfur dioxide	-0.008674
7	density	-0.413887
8	pH	0.102883
9	sulphates	0.064873
10	alcohol	0.259838

As can be seen, the most positive contributors to wine quality are:

- `residual sugar`, `alcohol` and `pH`.

The most negative contributors are:

- `density` and `volatile acidity`.

Thus, if a wine producer correctly manipulates those factors, while manufacturing a new wine, the new wine will probably be a high quality wine. This same methodology can be applied to develop not only new wines but many other products.

Another way to assess the predictive performance of a regression model is to use the root mean squared error (RMSE), which is the square root of the mean squared error. We can use `sklearn` to calculate. To do so, we first need to load the needed function:

```
from sklearn.metrics import mean_squared_error
```

To calculate the RMSE:

```
rmse = mean_squared_error(y_train, y_pred, squared=False)
rmse
```

```
0.7521265125434423
```

To set the decimals to 4 digits, we can add `:0.04f`:

```
print(f'RMSE= {rmse:0.04f}')
```

```
RMSE= 0.7521
```

Both MAE and RMSE are small compared the mean of the outcome variable:

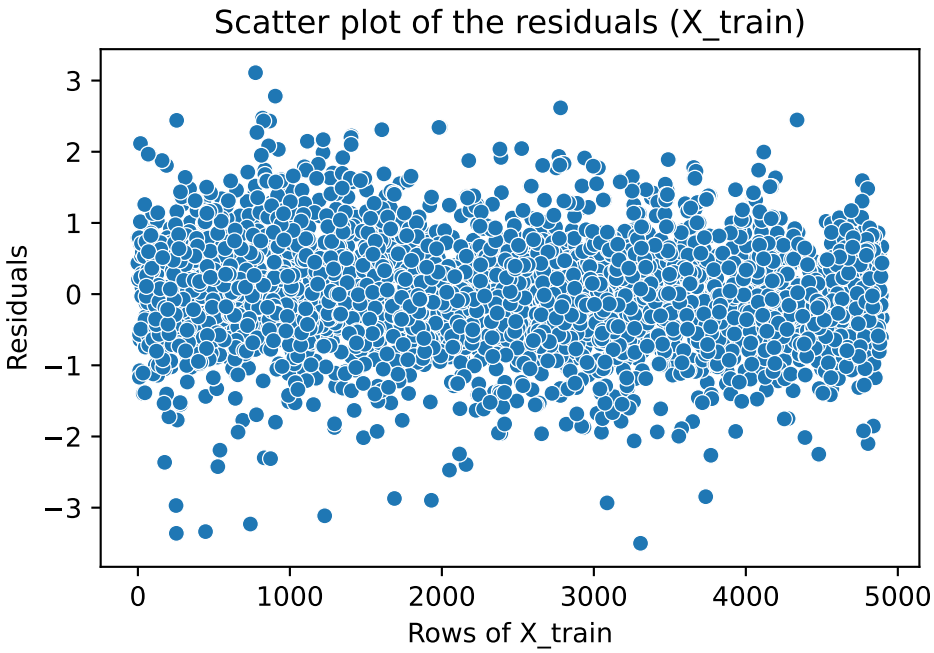
```
print(f'Ratio MAE / mean(y): {mae / np.mean(y_train):0.03f}')
```

```
Ratio MAE / mean(y): 0.100
```

We can plot the residuals (that is, the prediction errors), to have a broader understanding of the structure of the prediction errors:

```
from matplotlib import pyplot as plt # needed for adding the labels
import seaborn as sns
residuals = y_train - y_pred

sns.scatterplot(residuals)
plt.xlabel('Rows of X_train')
plt.ylabel('Residuals')
plt.title('Scatter plot of the residuals (X_train)')
plt.show()
```



As we can see, almost all errors of prediction are between -3 and 3, but most of them are between -2 and 2. Taking into account our goal, many prediction fail largely their targets! The initial bright picture is becoming somber! This justifies the fact that R2 is low, which expresses poor predictive performance. Recall that, regarding R2: values close to 0 mean low predictive performance, while values close to 1 mean high predictive performance.

Let us try to find examples where the prediction error is extreme:

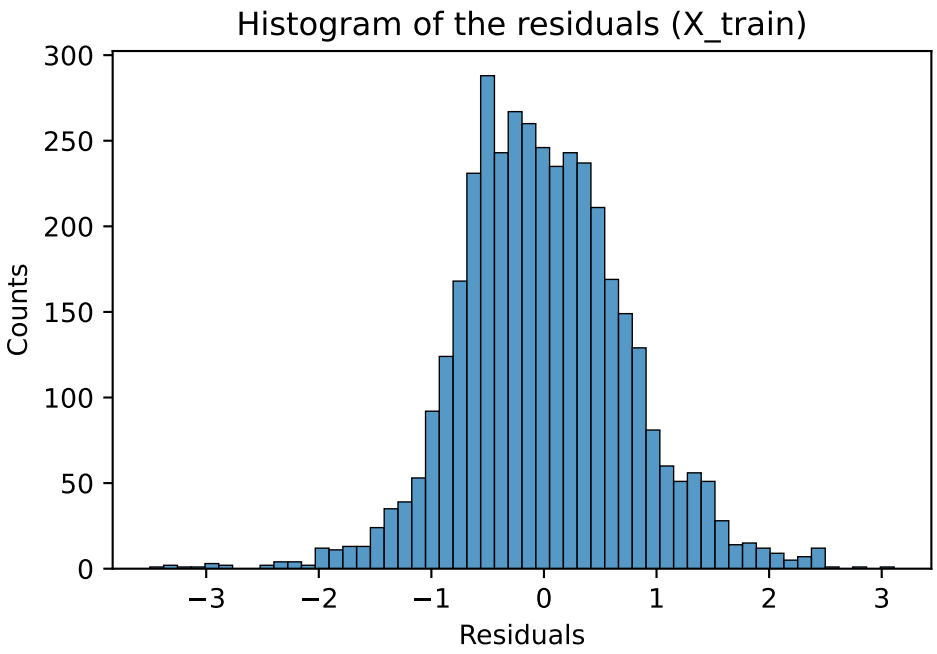
```
extreme_e = pd.DataFrame({
  'y_train': y_train[np.abs(residuals > 2.5)].values,
  'y_pred': np.round(y_pred[np.abs(residuals > 2.5)], 2)})

extreme_e
```

	y_train	y_pred
0	9	5.89
1	6	3.38
2	8	5.22

To plot the histogram of the residuals will help to further understand the structure of the prediction errors:

```
sns.histplot(residuals)
plt.xlabel('Residuals')
plt.ylabel('Counts')
plt.title('Histogram of the residuals (X_train)')
plt.show()
```



From the histogram, we can conclude that most of the residuals are concentrated between -1 and 1. Moreover, there is a tendency to underpredict the outcome variable when the prediction error is above 1, since the histogram is pronouncedly fatter than when the error is below -1.

The fact that our model is not so great (considering the company's goal) may be due to:

- Important predictors are missing in the dataset.
- The used model is not the most appropriate.
- Some tuning of model is still needed.

The reason because there are so many types of models is that some models perform better than others in each dataset. In sum, there is no universally better model!