Detailed resolution of percentage of defective radios prediction

Paulo S. A. Sousa

2023-03-21

Abstract

A prediction model based on the linear regression model will be developed, to predict the percentage of the defective radios.

To predict the percentage of defective radios will help the company to better manage the factors leading to higher percentage of defective radios.

1 Introduction

We have already explored the dataset of the de defective car radios, essential through plots. We have gained some intuitive understanding of important predictors. However, we have not yet developed a prediction model. That is what we are now going to do.

We will use, as prediction model, the linear regression model.

The code structure that we will create will be able to be used with other prediction models with minimal changes!

2 Dataset reading

To read the dataset, we will use `pandas` as usual. We first need to load `pandas`. Since we will probably use `numpy` as well later, we can also load `numpy` too:

```python
import pandas as pd
import numpy as np
```

Let us read the dataset into Python:

```python
df = pd.read_excel('datasets/data_carradios.xlsx')
print(df)
```

```
     perc_defec      bdate  team  training      datep  prizeq  prized
0          0.00 1981-07-02     8         1 2021-07-01       0       0
1         36.32 1992-06-14     6         0 2021-07-02       0       0
2         48.91 2003-05-28     7         0 2021-07-05     500       0
3         20.36 1992-06-14    10         0 2021-07-06       0       0
4         42.07 2003-05-28     7         0 2021-07-07     500       0
..          ...        ...   ...       ...        ...     ...     ...
995       21.50 1981-07-02     1         0 2021-12-13       0       0
996        7.53 1981-07-02     2         0 2021-12-14     500     600
997       27.70 1997-12-05    10         0 2021-12-15       0       0
998        0.00 1986-12-23     5         0 2021-12-16       0     600
999       38.39 1986-12-23     6         0 2021-12-17       0       0

[1000 rows x 7 columns]
```

To create dataframes `X` and `y`:

```
X = df.drop('perc_defec', axis=1)
y = df['perc_defec']
```

To split the dataset into train and test sets, we need to load `train_test_split` function:

```
from sklearn.model_selection import train_test_split
```

And now the split itself:

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
 ↪  test_size=0.2, random_state=45)
```

3 Pipelines creation

As already mentioned, we need to transform the birth date and the date of production to ages and weekdays, respectively. We can automate that transformation by using pipelines and column transformer.

We need first to load the needed functions:

```
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
```

Since we need to use function `now` of library `datetime`, we need to load that library:

```
import datetime
```

We now create a function that, given a column of dates, returns a column of ages. For the sake of a better integration with pipelines and column transformers, the column *is* expected to be a dataframe and the returned result *is* also expected to be a dataframe.

```
def get_ages(col):
  result = (datetime.datetime.now() - col).astype('<m8[Y]')
  result = pd.DataFrame(result)
  return result
```

Because `get_ages` is a user-defined function (and not an embedded Python function), we need to use `FunctionTransformer` function:

```
from sklearn.preprocessing import FunctionTransformer
```

To create a pipeline with `get_ages` function, we can do:

```
ager = Pipeline([
  ('ages', FunctionTransformer(get_ages,
↪  feature_names_out='one-to-one'))
  ])
```

By setting `feature_names_out='one-to-one'`, we are instructing Python to pass on the names of the columns to be transformed by our function `get_ages`. Since we will use `get_ages` only on column `bdate`, the `bdate` name will be passed on to the respective chained pipelines and transformers. (If we do not use `feature_names_out='one-to-one'`, we will not be able to get the names of the coefficients of the model, after the model is fit!)

Considering that we need to scale the column `bdate` after having it converted to ages, we need to add a scaler to the pipeline `ager` and beforehand load the respective function:

```
from sklearn.preprocessing import StandardScaler

ager = Pipeline([
  ('ages', FunctionTransformer(get_ages,
↪  feature_names_out='one-to-one')),
  ('scale', StandardScaler())
  ])
```

Now that we have the `ager` pipeline ready, we can insert it into the `ColumnTranformer`. We must be warned that we will only define the `ColumnTransformer`, but, for now, we will not run it – we run it only when we run `pipe.fit`. Consequently, no change in any column of the dataframe will take place until we run `pipe.fit`.

The `ColumnTransformer`:

```
preprocessor = ColumnTransformer([
  ('age_tr', ager, ['bdate'])],
  remainder='passthrough')
```

How can we be sure the column transformation is being done correctly? Well, we can

```
preprocessor.fit_transform(X_train)
```

```
array([[-1.0050044058180092, 5, 0, Timestamp('2021-07-23 00:00:00'), 0,
        600],
       [0.07540346605801468, 10, 0, Timestamp('2021-10-06 00:00:00'), 0,
        0],
       [0.07540346605801468, 2, 0, Timestamp('2021-07-05 00:00:00'), 500,
        600],
       ...,
       [0.07540346605801468, 9, 0, Timestamp('2021-09-30 00:00:00'), 500,
        600],
       [-1.9053442990480292, 10, 0, Timestamp('2021-08-30 00:00:00'), 0,
        0],
       [0.07540346605801468, 2, 0, Timestamp('2021-11-09 00:00:00'), 500,
        600]], dtype=object)
```

It clear that the first column of the dataframe, `bdate`, was transformed.

It was left to do now the conversion of the column `datep` to weekdays.

We first create a function, `get_weekdays`, that, given a column of dates, returns the respective weekdays. As already discussed, for the sake of a better integration with pipelines and column transformers, the column is expected to be a dataframe and the returned result is also expected to be a dataframe.

```
def get_weekdays(col):
  result = col['datep'].dt.weekday # or result =
  ↪  col.iloc[:,0].dt.weekday
  result = pd.DataFrame(result)
  return result
```

To create a pipeline with `get_weekdays` function, we can do:

```
weeker = Pipeline([
  ('weekd', FunctionTransformer(get_weekdays,
  ↪  feature_names_out='one-to-one'))
  ])
```

We can now add this pipeline to the `ColumnTransformer`:

```python
preprocessor = ColumnTransformer([
  ('age_tr', ager, ['bdate']),
  ('weekd_tr', weeker, ['datep'])],
  remainder='passthrough')
```

The transformed column assumes values of 0, 1, 3 and 4. These numbers do not represent a numerical amount: They are instead categories. When a predictor is categorical, it is recommended to transform them to dummy variables, using an one-hot conversion.

To see how the one-hot conversion works, consider a categorical variable X, having three classes A, B and C. The one-hot conversion transforms the original categorical variable into as many variables as classes, one for each class, and, for each one of these newly created variables, it sets to 1 whether the value of X is of the respective class and 0 otherwise. Thus, an example of the one-hot conversion of a categorical variable, X, can be the following:

	X	X_A	X_B	X_C
0	B	0	1	0
1	A	1	0	0
2	B	0	1	0
3	B	0	1	0
4	C	0	0	1

Notice that: if X_A and X_B are both zero, then the class is C; if X_A and X_C are both zero, then the class is B; if X_B and X_C are both zero, then the class is A. In sum, each of these binary variables is *fully* determined by the other binary variables. Consequently, we can remove one of these binary variable, as being redundant.

Having us learned the one-hot encoding, we can add it to the pipeline weeker. But before, we need to load OneHotEncoder function:

```python
from sklearn.preprocessing import OneHotEncoder
```

Adding the one-hot encoding to the pipeline weeker:

```python
weeker = Pipeline([
  ('weekd', FunctionTransformer(get_weekdays,
↪   feature_names_out='one-to-one')),
  ('onehot', OneHotEncoder(drop='first'))
  ])
```

We are dropping the first binary variable (drop='first'), because of *multicolinearity*. Since one of the binary variables, resulting from the one-hot encoding, is fully determined by the other binary variables, one binary variable is redundant and must be removed from the linear regression model, as it is known that multicolinearity impacts negatively the predictive performance of the linear regression model.

To check that the one-hot encoding was done correctly, we can do:

```
preprocessor = ColumnTransformer([
  ('age_tr', ager, ['bdate']),
  ('weekd_tr', weeker, ['datep'])],
  remainder='passthrough')

preprocessor.fit_transform(X_train)[:3,:]
```

```
array([[-1.00500441e+00,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00,  1.00000000e+00,  5.00000000e+00,
         0.00000000e+00,  0.00000000e+00,  6.00000000e+02],
       [ 7.54034661e-02,  0.00000000e+00,  1.00000000e+00,
         0.00000000e+00,  0.00000000e+00,  1.00000000e+01,
         0.00000000e+00,  0.00000000e+00,  0.00000000e+00],
       [ 7.54034661e-02,  0.00000000e+00,  0.00000000e+00,
         0.00000000e+00,  0.00000000e+00,  2.00000000e+00,
         0.00000000e+00,  5.00000000e+02,  6.00000000e+02]])
```

Everything seems to be fine, as we see more columns than before, a sign that the one-hot encoding is being performed (it creates more columns, corresponding to the binary variables).

Since the `team` variable is also categorical, we need to one-hot encode it:

```
preprocessor = ColumnTransformer([
  ('age_tr', ager, ['bdate']),
  ('weekd_tr', weeker, ['datep']),
  ('team_tr', OneHotEncoder(drop='first'), ['team'])],
  remainder='passthrough')
```

Finally, we need to scale the prize columns:

```
from sklearn.preprocessing import StandardScaler

preprocessor = ColumnTransformer([
  ('age_tr', ager, ['bdate']),
  ('weekd_tr', weeker, ['datep']),
  ('team_tr', OneHotEncoder(drop='first'), ['team']),
  ('scale_tr', StandardScaler(), ['prizeq', 'prized'])],
  remainder='passthrough')
```

We may wonder about the question: Why have not we used pipelines to transform columns `team`, `prizeq` and `prized`? We could have used pipelines as well – it would also work fine. However, since the transformation of the mentioned columns consists only of a single step, we can avoid using pipelines.

4 Model creation and estimation

All preprocessing is defined. Now, we need to define the pipeline with the model:

```
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LinearRegression

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

We can now fit the pipeline `pipe`:

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

To get the predictions to the train set:

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

```
print(preprocessor.get_feature_names_out())
print(pipe.named_steps['lm'].coef_)
```

```
['age_tr__bdate' 'weekd_tr__datep_1' 'weekd_tr__datep_2'
 'weekd_tr__datep_3' 'weekd_tr__datep_4' 'team_tr__team_2'
 'team_tr__team_3' 'team_tr__team_4' 'team_tr__team_5' 'team_tr__team_6'
 'team_tr__team_7' 'team_tr__team_8' 'team_tr__team_9' 'team_tr__team_10'
 'scale_tr__prizeq' 'scale_tr__prized' 'remainder__training']
[  0.1524855   -1.01727536  -0.03583153  -0.25943286  18.1387667
  -2.70801008   9.70625074  -5.19140865   0.49258097  -0.53010065
   7.00396978  -8.35337177  -1.84173354  -0.58323912   5.37497188
  -8.87474267 -13.54478042]
```

5 Predictive performance of the model

To determine the predictive performance of the model:

```
from sklearn.metrics import r2_score, mean_absolute_error,
 ↪  mean_squared_error

print(f'R2= {r2_score(y_train, y_pred):0.03f}')
print(f'MAE= {mean_absolute_error(y_train, y_pred):0.03f}')
print(f'RMSE= {mean_squared_error(y_train, y_pred,
 ↪  squared=False):0.03f}')
```

```
R2= 0.917
MAE= 3.336
RMSE= 4.333
```

6 Lasso regression

When a predictor has little impact on the prediction, we should remove it from the model:

- A parsimonious model can be better and more easily interpreted;
- The larger the number of predictors, the more data we need to ensure good predictive performance of the models.

To remove weak predictors, Lasso regression is a great tool, as it removes them automatically.

To load the Lasso function:

```
from sklearn.linear_model import Lasso
```

To apply the Lasso regression, we just need to replace the linear regression model by the Lasso one.

```
pipe = Pipeline([
  ('pre', preprocessor),
  ('lasso', Lasso(alpha=0.1))
])

pipe.fit(X_train, y_train)
```

	Coefficients	Estimates
0	age_tr__bdate	0.03
1	weekd_tr__datep_1	-0.55
2	weekd_tr__datep_2	-0.00
3	weekd_tr__datep_3	-0.00
4	weekd_tr__datep_4	17.69
5	team_tr__team_2	-0.00
6	team_tr__team_3	9.01
7	team_tr__team_4	-0.00
8	team_tr__team_5	0.00
9	team_tr__team_6	-0.00
10	team_tr__team_7	8.56
11	team_tr__team_8	-2.30
12	team_tr__team_9	0.00
13	team_tr__team_10	-0.00
14	scale_tr__prizeq	4.32
15	scale_tr__prized	-8.60
16	remainder__training	-18.42

As discussed previously, some of the coefficients were forced down to zero, the weaker ones.

Let's us now evaluate the predictive performance of the model in the train set:

```
y_pred = pipe.predict(X_train)
print(f'R2= {r2_score(y_train, y_pred):0.03f}')
print(f'MAE= {mean_absolute_error(y_train, y_pred):0.03f}')
print(f'RMSE= {mean_squared_error(y_train, y_pred,
 ↪  squared=False):0.03f}')
```

```
R2= 0.915
MAE= 3.304
RMSE= 4.389
```

The predictive performance of our model seems to be quite good. Should we expect it to also perform good in the test set?

```
y_pred = pipe.predict(X_test)

print(f'R2= {r2_score(y_test, y_pred):0.03f}')
print(f'MAE= {mean_absolute_error(y_test, y_pred):0.03f}')
print(f'RMSE= {mean_squared_error(y_test, y_pred,
 ↪  squared=False):0.03f}')
```

```
R2= 0.892
MAE= 3.758
RMSE= 5.037
```

The answer is yes! The predictive performance of the model seems to be similar in both sets, train and test.