IST438-W2-Applications

3/6/23

Supervised Learning: Linear Regression Models

In this application, we will interest the regression problem under the following sections:

- Training model
- Measuring model performance
- Checking over and underfitting problem
- Checking model assumptions

Packages

We need to install `{DALEX}` package to use `apartments` data set in applications. Please use the two-step codes below: (1) install, (2) load the package.

```
# install.packages("DALEX")
# install.packages("ggplot2")
# install.packages("car")
library(DALEX)
library(ggplot2)
library(car)
```

The first line is `hashtag`ed to faster this process. Do not forget to un-`hashtag` it in your first run. Because any line, which is hashtaged, is not run in R. It is turned the command line!

Dataset

We use the `apartments` data set from `{DALEX}` package. It has some features of the apartments in Warsaw, Poland.

```
# calling apartments from {DALEX} package
data(apartments)
```

You can use `str()` function to take a look data set. It returns a data frame consists information about the data set such as:

- number of observation
- number of features (variables)
- name of features
- type of features
- a few observations of features

```
# Take a look to dataset
str(apartments)
```

```
'data.frame':   1000 obs. of  6 variables:
 $ m2.price         : num  5897 1818 3643 3517 3013 ...
 $ construction.year: num  1953 1992 1937 1995 1992 ...
 $ surface          : num  25 143 56 93 144 61 127 105 145 112 ...
 $ floor            : int  3 9 1 7 6 6 8 8 6 9 ...
 $ no.rooms         : num  1 5 2 3 5 2 5 4 6 4 ...
 $ district         : Factor w/ 10 levels "Bemowo","Bielany",..: 6 2 5 4 3 6 3 7 6 6 ...
```

Task: Regression

In this application, we try to train a linear regression model on the `apartments` data set to predict the m^2 price of an intended apartment.

Step 1 - Splitting the data set

We can use `sample()` function to split the data set as `test` and `train` set. Do not forget to set a seed to reproduce this process in future. It is important to get same observations in each run of these codes.

```r
set.seed(123) # for reproducibility
index <- sample(1 : nrow(apartments), round(nrow(apartments) * 0.80))
train <- apartments[index, ]
test  <- apartments[-index, ]
```

In the codes above, we get a sample from a vector has a length is equal to the number of observation in the data set. We must input a sequence as the first argument and the ratio of the train set as the second argument. Then we can save the randomly selected values of the sequence as `index` object. We can set the observations to `train` and `test` objects by using the `index` object.

Step 2 - Train a linear regression model

We can use the `lm()` function to train a linear regression model. It needs two obligatory arguments: (1) model formula and (2) data set that used to train the model. It is possible to define the model formula like `y ~ x1 + x2 + ...` where `y` is the target variable which is interested features to predict and `x`s are the features that used the information to predict the target variable.

There is a tip about the model formula: you can use `y ~.` instead of defining all features you want to input to the model.

```r
lrm_model <- lm(m2.price ~ ., data = train)
```

In above, we train a linear regression model by using the `train` data that we splited in previous step and the model formula. Then, we assigned the output of `lm()` function to the `lrm_model` object. Do not forget that you can give another name to the model object whatever you want, because it is just an object!

Let see the output of the model:

```r
lrm_model
```

```
Call:
lm(formula = m2.price ~ ., data = train)

Coefficients:
        (Intercept)    construction.year              surface
           5404.531              -0.437              -10.139
              floor            no.rooms      districtBielany
```

```
           -99.236             -35.862              14.553
districtMokotow      districtOchota       districtPraga
           928.000             938.294             -18.703
districtSrodmiescie       districtUrsus     districtUrsynow
          2077.514              38.646             -21.448
       districtWola    districtZoliborz
             5.154             890.673
```

Model output returns the information about model formula, train data, and the estimated values of model parameters under the `Coefficients` title. You can see the numeric (continuous) features with its name, but the categorical features with `its name + name of the category` such as `districtOchota`.

If you want to see more detail about the model, use the `summary()` function.

```
summary(lrm_model)
```

```
Call:
lm(formula = m2.price ~ ., data = train)

Residuals:
   Min     1Q Median     3Q    Max
-249.0 -201.4 -167.7  377.7  476.0

Coefficients:
                    Estimate Std. Error t value Pr(>|t|)
(Intercept)         5404.5310   761.7231   7.095 2.89e-12 ***
construction.year     -0.4370     0.3880  -1.126   0.2604
surface              -10.1388     0.6474 -15.660  < 2e-16 ***
floor                -99.2361     3.4140 -29.067  < 2e-16 ***
no.rooms             -35.8617    17.6361  -2.033   0.0423 *
districtBielany       14.5529    44.4367   0.327   0.7434
districtMokotow      927.9999    43.1075  21.528  < 2e-16 ***
districtOchota       938.2940    44.3454  21.159  < 2e-16 ***
districtPraga        -18.7035    44.6213  -0.419   0.6752
districtSrodmiescie 2077.5145    44.9024  46.267  < 2e-16 ***
districtUrsus         38.6455    43.9379   0.880   0.3794
districtUrsynow      -21.4476    43.0660  -0.498   0.6186
districtWola           5.1541    44.1533   0.117   0.9071
districtZoliborz     890.6728    44.4854  20.022  < 2e-16 ***
---
```

```
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 280.6 on 786 degrees of freedom
Multiple R-squared:  0.9018,    Adjusted R-squared:  0.9002
F-statistic: 555.5 on 13 and 786 DF,  p-value: < 2.2e-16
```

It returns some statistics about the residuals, the model parameters, and also some performance metric values such as the multiple R^2 and the adjusted R^2. These values show the model performance on train data.

Step 3 - Measuring model performance

It is necessary to check the model performance of the model on test set. Because we are interested to train such a model has a good generalizability performance. To do this, we first calculate the predicted values of the target variable on test set.

Do not forget to exclude the true values of the target variable from the test set!

```
predicted_y <- predict(lrm_model, test[,-1])
head(predicted_y)
```

```
       1        3        7        9       12       22
6041.446 3800.527 3210.712 4358.694 2995.008 3149.241
```

Then, we can calculate some performance metrics of the trained model.

```
error <- test$m2.price - predicted_y
head(error)
```

```
        1         3         7         9        12        22
-144.4463 -157.5273 -227.7120  386.3061 -198.0076 -153.2411
```

Mean squared error (MSE), root mean squared error (RMSE), median absolute error (MAE) are the main performance metrics for the model used in regression task.

```
mse_model  <- mean(error ^ 2)
rmse_model <- sqrt(mean(error ^ 2))
mae_model  <- median(abs(error))
```

Let's compare the values of these metrics:

```
mse_model
```

```
[1] 81266.28
```

```
rmse_model
```

```
[1] 285.0724
```

```
mae_model
```

```
[1] 211.7335
```

Q: What are differences between these metrics? Which of these should be used in which situations?

Step X1 - Checking the possible over and underfitting problem

The way to check there is any problem related to over or underfitting in the model is to compare the model performance on train and test set. Let's use the RMSE for this:

```
rmse_train <- sqrt(mean((lrm_model$residuals) ^ 2))
rmse_test  <- rmse_model
```

Calculate the difference between th RMSE values:

```
rmse_train - rmse_test
```

```
[1] -6.892241
```

The difference is negative means that the performance of the model is better on test set than train set. This may be a sign for overfitting problem because the model performance is better on test set. So, we may say that the model learn more from the train set that cause the poorer performance on test set. But there is no threshold to conclude that there is any problem related to over or underfitting. We can use this difference just as a sign!

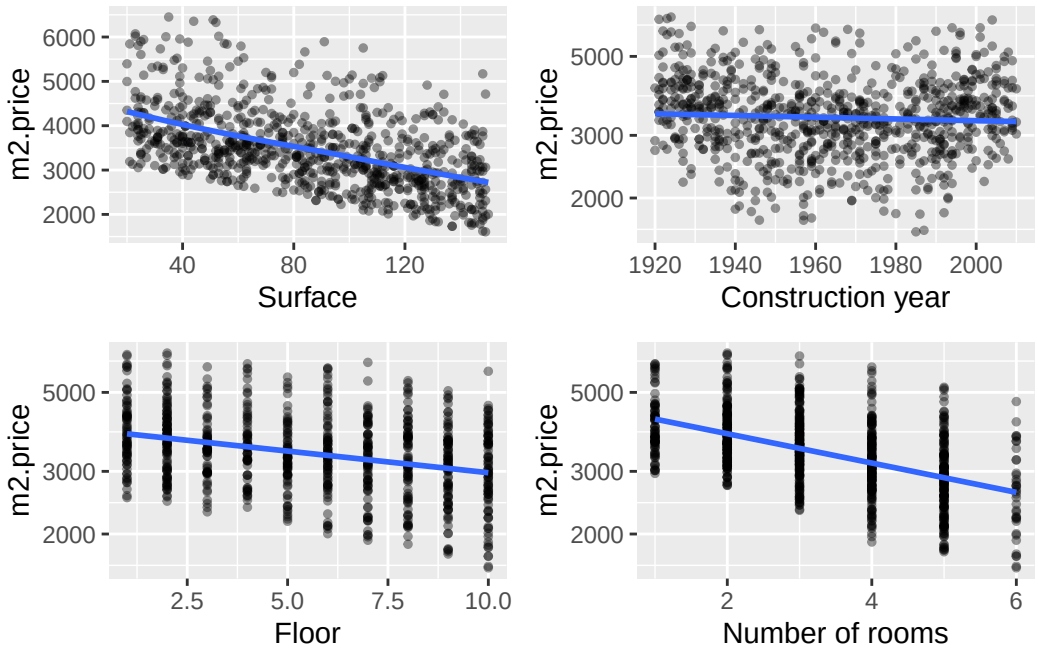
Step X2 - Checking the bias and variance of the model

Next week or later!

Step X3 - Checking the model assumptions

1. Linear relationship between features and target variable:

```
p1 <- ggplot(train, aes(surface, m2.price)) +
  geom_point(size = 1, alpha = .4) +
  geom_smooth(se = FALSE) +
  scale_y_continuous("m2.price") +
  xlab("Surface")

p2 <- ggplot(train, aes(construction.year, m2.price)) +
  geom_point(size = 1, alpha = .4) +
  geom_smooth(method = "lm", se = FALSE) +
  scale_y_log10("m2.price") +
  xlab("Construction year")

p3 <- ggplot(train, aes(floor, m2.price)) +
  geom_point(size = 1, alpha = .4) +
  geom_smooth(method = "lm", se = FALSE) +
  scale_y_log10("m2.price") +
  xlab("Floor")

p4 <- ggplot(train, aes(no.rooms, m2.price)) +
  geom_point(size = 1, alpha = .4) +
  geom_smooth(method = "lm", se = FALSE) +
  scale_y_log10("m2.price") +
  xlab("Number of rooms")

gridExtra::grid.arrange(p1, p2, p3, p4, nrow = 2)
```



Solutions for non-linearity:

- Use non-linear models (nonlinear regression or other ML models)
- Transform X and/or Y to obtain a linear relationship using Box-Cox, inverse, or etc.

2. Constant variance among residuals (errors)

The errors (residuals) must follows normal distribution with a constant variance. We can check this assumption by using the Shapiro-Wilk test or visual diagnostics.

Let's try to use Shapiro-Wilk test first:

Null hypothesis: errors follow normal the distribution Alternative hypothesis: errors do not follow the normal distribution

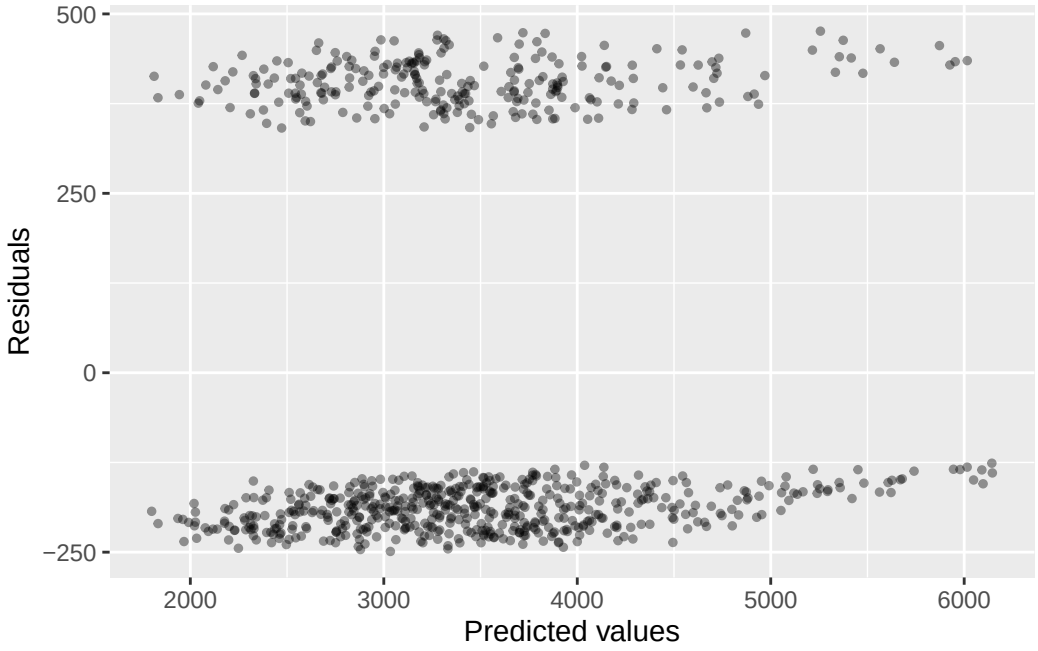
```
shapiro.test(lrm_model$residuals)
```

```
    Shapiro-Wilk normality test

data:  lrm_model$residuals
W = 0.68126, p-value < 2.2e-16
```

Because of the p-value is close to 0, and less than 0.05, there is enough evidence to reject the null hypothesis. This means that the errors do not follow the normal distribution.

We may see the visual diagnostics to check the constant variance, but in this case it is not necessary because error do not follow normal distribution.

```
con_var <- data.frame(fitted = lrm_model$fitted.values,
                      resid  = lrm_model$residuals)

ggplot(con_var, aes(fitted, resid)) +
  geom_point(size = 1, alpha = .4) +
  xlab("Predicted values") +
  ylab("Residuals")
```



In the plot above, the points must be distributed along the line $y = 0$ with a constant pattern in ideal case.

3. No autocorrelation between features (independent variables)

Linear regression assumes the errors are independent and uncorrelated. If in fact, there is correlation among the errors, then the estimated standard errors of the coefficients will be biased leading to prediction intervals being narrower than they should be.

To test the autocorrelation of the residuals, you can use the Durbin-Watson test with the following hypotheses:

Null hypothesis: $\rho_r = 0$ Alternative hypothesis: $\rho_r \neq 0$

```
durbinWatsonTest(lrm_model)
```

```
 lag Autocorrelation D-W Statistic p-value
   1     -0.02788732      2.054854   0.456
 Alternative hypothesis: rho != 0
```

Because of the p-value is $0.418 > 0.05$, there is no evidence to reject the null hypothesis. This means that the residuals are not autocorrelated at 95% significance level.

4. More observation than predictors (n > p)

```
# n: number of observations
# p: number of predictors
n <- dim(train)[1]
p <- dim(train)[2] - 1 # Do not forget to minus the target features because it is not a pr

n > p
```

```
[1] TRUE
```

5. No multicollinearity between features (independent variables)

Collinearity refers to the situation in which two or more predictor variables are closely related to one another. The presence of collinearity can pose problems in the estimation of model parameters, since it can be difficult to separate out the individual effects of collinear variables on the response. In fact, collinearity can cause predictor variables to appear as statistically insignificant when in fact they are significant. This obviously leads to an inaccurate interpretation of coefficients and makes it difficult to identify influential predictors.

We can use the Variance Inflation Factors (VIFs) to measure the collinearity between features. Let's calculate the VIFs by using the `vif()` function in `{car}` package:

```
vif(lrm_model)
```

```
                     GVIF Df GVIF^(1/(2*Df))
construction.year 1.016111  1        1.008023
surface           5.989268  1        2.447298
floor             1.015405  1        1.007673
no.rooms          5.986822  1        2.446798
district          1.037010  9        1.002021
```

If there is any feature has a GVIF value higher than 5 or 10, it may be correlated with the any of other features in the model. There is no consensus about the threshold for multicollinearity, but it is accepted that there is a multicollinearity problem if the VIF/GVIF values are higher than 10 To make the valid this assumption, the easiest solution is to exclude this feature from the model, or you may use more complex regression model instead of linear regression model.