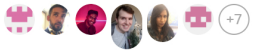
Model Fit in Linear Regression



Model Fit in Linear Regression

Introduction

In this lecture, you'll learn how to evaluate your model results, and you'll learn methods to select the appropriate features.

Objectives

You will be able to:

- Analyze the results of regression and R-squared and adjusted-R-squared
- Understand and apply forward and backward predictor selection

R-squared and adjusted-R-squared

Let's have another look at the output for our `auto-mpg` output.

The code below reiterates the steps we've taken before: we've created dummies for our categorical variables and have log-transformed some of our continuous predictors.

In [1]:
```
import pandas as pd
import numpy as np
data = pd.read_csv("auto-mpg.csv")

acc = data["acceleration"]
logdisp = np.log(data["displacement"])
loghorse = np.log(data["horsepower"])
logweight= np.log(data["weight"])

scaled_acc = (acc-min(acc))/(max(acc)-min(acc))
scaled_disp = (logdisp-np.mean(logdisp))/np.sqrt(np.var(logdisp))
scaled_horse = (loghorse-np.mean(loghorse))/(max(loghorse)-min(loghorse))
scaled_weight= (logweight-np.mean(logweight))/np.sqrt(np.var(logweight))

data_fin = pd.DataFrame([])
data_fin["acc"]= scaled_acc
data_fin["disp"]= scaled_disp
data_fin["horse"] = scaled_horse
data_fin["weight"] = scaled_weight
cyl_dummies = pd.get_dummies(data["cylinders"], prefix="cyl")
yr_dummies = pd.get_dummies(data["model year"], prefix="yr")
orig_dummies = pd.get_dummies(data["origin"], prefix="orig")
mpg = data["mpg"]
data_fin = pd.concat([mpg, data_fin, cyl_dummies, yr_dummies, orig_dummies], axis=1)
```

In [2]:
```
data_ols = pd.concat([mpg, scaled_acc, scaled_weight, orig_dummies], axis= 1)
data_ols.head(3)
```

Out[2]:

	mpg	acceleration	weight	orig_1	orig_2	orig_3
0	18.0	0.238095	0.720986	1	0	0
1	15.0	0.208333	0.908047	1	0	0
2	18.0	0.178571	0.651205	1	0	0

In [3]:
```
#import statsmodels.api as sm
from statsmodels.formula.api import ols
```

In [4]:
```
outcome = 'mpg'
predictors = data_ols.drop('mpg', axis=1)
predictors = predictors.drop("orig_3",axis=1)
pred_sum = "+".join(predictors.columns)
formula = outcome + "~" + pred_sum
```

In [5]:
```
model = ols(formula= formula, data=data_ols).fit()
model.summary()
```

Out[5]:

OLS Regression Results

Dep. Variable:	mpg	**R-squared:**	0.726
Model:	OLS	**Adj. R-squared:**	0.723
Method:	Least Squares	**F-statistic:**	256.7
Date:	Mon, 15 Oct 2018	**Prob (F-statistic):**	1.86e-107

Time:	20:19:20	**Log-Likelihood:**	-1107.2
No. Observations:	392	**AIC:**	2224.
Df Residuals:	387	**BIC:**	2244.
Df Model:	4		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	22.4826	0.789	28.504	0.000	20.932	24.033
acceleration	5.0494	1.389	3.634	0.000	2.318	7.781
weight	-5.8764	0.282	-20.831	0.000	-6.431	-5.322
orig_1	-1.7218	0.653	-2.638	0.009	-3.005	-0.438
orig_2	-1.3095	0.688	-1.903	0.058	-2.662	0.043

Omnibus:	37.427	**Durbin-Watson:**	0.840
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	55.989
Skew:	0.648	**Prob(JB):**	6.95e-13
Kurtosis:	4.322	**Cond. No.**	9.59

Let's discuss some key metrics in the light of our output:

- R-squared uses a baseline model which is the worst model. This baseline model does not make use of any independent variables to predict the value of dependent variable Y. Instead it uses the mean of the observed responses of dependent variable Y and always predicts this mean as the value of Y. The mathematical formula to calculate R-squared for a linear regression line is in terms of squared errors for the fitted model and the baseline model. In the formula below, SS_{RES} is the residual sum of squared errors or our model, also known as SSE, which is the error between the real and predicted values. SS_{TOT} is the difference between real and mean y values.

$$R^2 = 1 - \frac{SS_{RES}}{SS_{TOT}} = \frac{\sum_i y_i - \hat{y_i}}{\sum_i Y_i - \bar{y_i}}$$

- The problem with R^2 is that, whichever predictor you **add** to your model which will make your model more complex, will increase your R^2 value. It makes seem like adding a predictor is always better. The R^2_{adj} penalized for this. Make sure to read this blogpost on the difference between the two to get a better sense to why use R^2_{adj} !

The parameter estimates and p-values

Just like with single linear regression, the parameters or coefficients we're calculating have a p-value or *significance* attached to them. The interpretation of the p-value for each parameter is exactly the same as for single multiple regression:

> The p-value represents the probability that the coefficient is actually zero.

In the Statsmodels output, the p-value can be found in the column with name $P > |t|$. A popular threshols for the p-value is 0.05, where we $p < 0.05$ denotes that a certain parameter is significant, and $p > 0.05$ means that the parameter isn't significant.

The two columns right to the p-value column represent the bounds associated with the 95% confidence interval. What this means is that, after having run the model, we are 95% certain that our parameter value is withing the bounds of this interval. When you chose a p-value cut-off of 0.05, there is an interesting relationship between the 95% confidence interval and the p-value: If the 95% confidence does not include 0, the p-value will be smaller than 0.05, and the parameter estimate will be significant.

Which variables are most important when predicting the target?

Now that you know how predictors influence the target, let's talk about selecting the right predictors for your model. It is reasonable to think that when having many potential predictors (sometimes 100s or 1000s of predictors can be available!) not only will adding all of them in largely increase computation time, it might even lead to inferior R^2_{adj} or inferior predictions in general. There are several ways to approach variable selection, and we'll only touch upon this in an ad-hoc manner in this lesson. The most straightforward way is to run a model with each possible *combination* of variables and see which one results in the best metric of your choise, let's say, R^2_{adj}.

> This is where your combinatorics knowledge comes in handy!

Now, when you know about combinations, you know that the number of combinations can add up really quickly.

> Imagine you have 6 variables. How many models can you create with 6 variables and all subsets of variables?

You can create:

- 1 model with all 6 variables
- 6 models with just 5 variables: $\binom{6}{}$

- 6 models with just 5 variables: $\binom{6}{5}$
- 15 models with just 4 variables: $\binom{6}{4} = \frac{6!}{2! * 4!} = 15$
- 20 models with 3 variables: $\binom{6}{3} = \frac{6!}{3! * 3!} = 20$
- 15 models with 2 variables: $\binom{6}{2} = \frac{6!}{4! * 2!} = 15$
- 6 models with 1 variable: $\binom{6}{1}$
- And, technically, 1 model with no predictors: this will return a model that simply returns the mean of Y.

> This means that with just 6 variables, a so-called exhaustive search of all submodels results in having to evaluate *64 models*. Below, we'll describe 2 methods that can be use to select a submodel with the most appropriate features: **stepwise selection** on the one hand, and **feature ranking with recursive feature elimination** on the other hand.

Stepwise selection with p-values

In stepwise selection, you starts with and empty model (which only includes the intercept), and each time, the variable that has an associated parameter estimate with the lowest p-value is added to the model (forward step). After adding each new variable in the model, the algorithm will look at the p-values of all the other parameter estimates which were added to the model previously, and remove them if the p-value exceeds a certain value (backward step). The algorithm stops when no variables can be added or removed given the threshold values.

For more information, it's worth having a look at the wikipedia page on stepwise regression.

Unfortunately, stepwise selection is not readily available a Python library just yet. This stackexchange post, however, presents the code to do a stepwise selection in statsmodels.

In [6]:
```
import statsmodels.api as sm

def stepwise_selection(X, y,
                       initial_list=[],
                       threshold_in=0.01,
                       threshold_out = 0.05,
                       verbose=True):
    """ Perform a forward-backward feature selection
    based on p-value from statsmodels.api.OLS
    Arguments:
        X - pandas.DataFrame with candidate features
        y - list-like with the target
        initial_list - list of features to start with (column names of X)
        threshold_in - include a feature if its p-value < threshold_in
        threshold_out - exclude a feature if its p-value > threshold_out
        verbose - whether to print the sequence of inclusions and exclusions
    Returns: list of selected features
    Always set threshold_in < threshold_out to avoid infinite looping.
    See https://en.wikipedia.org/wiki/Stepwise_regression for the details
    """
    included = list(initial_list)
    while True:
        changed=False
        # forward step
        excluded = list(set(X.columns)-set(included))
        new_pval = pd.Series(index=excluded)
        for new_column in excluded:
            model = sm.OLS(y, sm.add_constant(pd.DataFrame(X[included+[new_column]]))).fi
            new_pval[new_column] = model.pvalues[new_column]
        best_pval = new_pval.min()
        if best_pval < threshold_in:
            best_feature = new_pval.idxmin()
            included.append(best_feature)
            changed=True
            if verbose:
                print('Add  {:30} with p-value {:.6}'.format(best_feature, best_pval))

        # backward step
        model = sm.OLS(y, sm.add_constant(pd.DataFrame(X[included]))).fit()
        # use all coefs except intercept
        pvalues = model.pvalues.iloc[1:]
        worst_pval = pvalues.max() # null if pvalues is empty
        if worst_pval > threshold_out:
            changed=True
            worst_feature = pvalues.argmax()
            included.remove(worst_feature)
            if verbose:
                print('Drop {:30} with p-value {:.6}'.format(worst_feature, worst_pval))
        if not changed:
            break
    return included
```

```
/Users/lore.dirick/anaconda3/lib/python3.6/site-packages/statsmodels/compat/pandas.py:5
6: FutureWarning: The pandas.core.datetools module is deprecated and will be removed in
a future version. Please use the pandas.tseries module instead.
  from pandas.core import datetools
```

In [7]:
```
result = stepwise_selection(predictors, data_fin["mpg"], verbose = True)
print('resulting features:')
print(result)
```

```
Add  weight                          with p-value 1.16293e-107
Add  acceleration                    with p-value 0.000646572
resulting features:
['weight', 'acceleration']
```

Applying the stepwise selection on our small auto-mpg example (just starting with the predictors weight, acceleration and the origin categorical levels), we end up with a model that just keeps the continuous variables in.

Feature ranking with recursive feature elimination

Scikit-learn also provides a few functionalities for feature selection. Their `Feature Ranking with Recursive Feature Elimination` selects the pre-specified n most important features. See here for more information on how the algorithm works.

In [12]:
```
#from sklearn.datasets import make_friedman1
from sklearn.feature_selection import RFE
from sklearn.linear_model import LinearRegression

linreg = LinearRegression()
selector = RFE(linreg, n_features_to_select = 2)
selector = selector.fit(predictors, data_fin["mpg"])
```

Calling the `.support_` attribute tells you which variables are selected

In [9]:
```
selector.support_
```

Out[9]:
```
array([ True,  True, False, False])
```

Calling `.ranking_` shows the ranking of the features, selected features are assigned rank 1

In [10]:
```
selector.ranking_
```

Out[10]:
```
array([1, 1, 2, 3])
```

By calling `.estimator_` on the RFE object, you can get access to the parameter estimates through `.coef_` and `.intercept`.

In [11]:
```
estimators = selector.estimator_
print(estimators.coef_)
print(estimators.intercept_)
```

```
[ 4.77870536 -6.26580459]
21.300812564199767
```

Forward selection using adjusted R-squared

This resource provides code for a forward selection procedure (much like stepwise selection, buth without a backward pass), but this time looking at the adjusted-R-squared to make decisions on which variable to add in the mode.

Summary

Congrats! In this lecture, you learned about how you can perform selection methods using p-values and adjusted-R-squared.