Py_Hello-Word_Regression

January 19, 2021

1 Hello World Data Analysis in Python

In this notebook we will do a basic data analysis in python.

Our goal is to see how the price of a used car depends on characteristics of the car (features.)

We will:

- read in the data
- plot and tranform the data
- fit a simple multiple regression model, getting fits, predictions and standard inference.

1.1 Import Needed Modules

We need to import numpy, pandas, and matplot.pyplot (as np, pd, and plt).
numpy gives as vector/matrix/array operations, pandas gives us "data frames" data structures, and matplot.pyplot give us graphics.

We also import LinearRegression from sklearn.linear_model to run the multiple regression.

We also import statsmodels.api (as sm) to get inference and summaries (e.g. R-squared, t-stats, p-values) for multiple regression.

```
In [1]: import matplotlib.pyplot as plt
        import seaborn; seaborn.set()

        import numpy as np
        import pandas as pd
        from sklearn.linear_model import LinearRegression
        import statsmodels.api as sm


        #ipython magic function, helps display of plots in a notebook
        %matplotlib inline
```

1.2 Read in the Data and Get the Variables We Want

We will

- read in the data to a pandas data frame

- pull off price, mileage, and year
- divide price and mileage by 1,000
- do some simple summaries

First we will read in the data from the file susedcars.csv on Rob's data page.

```
In [2]: cd = pd.read_csv("http://www.rob-mcculloch.org/data/susedcars.csv") #cd for car data
        print("*** the type of cd is:")
        print(type(cd))
        print("***number number of rows and columns is: ",cd.shape)
        print("***the column names are:")
        print(cd.columns.values)
```

```
*** the type of cd is:
<class 'pandas.core.frame.DataFrame'>
***number number of rows and columns is:  (1000, 7)
***the column names are:
['price' 'trim' 'isOneOwner' 'mileage' 'year' 'color' 'displacement']
```

Each of the 1,000 rows corresponds to a used cars. price is what the car sold for. The other variables are *features* describing the car. Our goal is to relate the price to the other features.

We can pull one column (variable) out of the data frame by name.

```
In [3]: temp = cd['mileage']  # pull out the variable mileage
        temp[0:5]  # print out the mileage of the first 5 cars, note the indexing!! [a,b)
```

```
Out[3]: 0     36858.0
        1     46883.0
        2    108759.0
        3     35187.0
        4     48153.0
        Name: mileage, dtype: float64
```

The feature mileage is a numeric variable with units miles. We can summarize it using the usual descriptive summaries:

```
In [4]: print(cd['mileage'][0:5])  # first 5 values of variable mileage
        cd['mileage'].describe()   # summary statistics of variable mileage
```

```
0     36858.0
1     46883.0
2    108759.0
3     35187.0
4     48153.0
Name: mileage, dtype: float64
```

```
Out[4]: count      1000.000000
        mean      73652.408000
        std       42887.422189
        min        1997.000000
        25%       40132.750000
        50%       67919.500000
        75%      100138.250000
        max      255419.000000
        Name: mileage, dtype: float64
```

The feature color is a categorical variable. Each car is in one of the color categories. We can't summarize a categorical variable the same way that we summarize a numeric variable. There is no "average" color. To summarize a categorical variable we simply count how many observations are in each category.

```
In [5]: print(cd['color'][0:5])  # colors of first 5 cars
        cd['color'].value_counts() # how many cars have each color
```

```
0    Silver
1     Black
2     White
3     Black
4     Black
Name: color, dtype: object
```

```
Out[5]: Black     415
        other     227
        Silver    213
        White     145
        Name: color, dtype: int64
```

Let's focus on the two numeric features mileage and year. Our goal will be to see how price relates to mileage and year.
We will divide both price and mileage by 1,000 to make the results easier to understand.

```
In [6]: cd = cd[['price','mileage','year']]
        cd['price'] = cd['price']/1000
        cd['mileage'] = cd['mileage']/1000
        print(cd.head()) # head just prints out the first few rows
```

```
    price  mileage  year
0  43.995   36.858  2008
1  44.995   46.883  2012
2  25.999  108.759  2007
3  33.880   35.187  2007
4  34.895   48.153  2007
```

```
In [7]: print(cd.describe()) #summarize each column
```

```
           price      mileage         year
count  1000.000000  1000.000000  1000.000000
mean     30.583318    73.652408  2006.939000
std      18.411018    42.887422     4.194624
min       0.995000     1.997000  1994.000000
25%      12.995000    40.132750  2004.000000
50%      29.800000    67.919500  2007.000000
75%      43.992000   100.138250  2010.000000
max      79.995000   255.419000  2013.000000
```

```
In [8]: print(cd.corr()) #compute the correlation between each column
```

```
            price   mileage      year
price    1.000000 -0.815246  0.880537
mileage -0.815246  1.000000 -0.744729
year     0.880537 -0.744729  1.000000
```

Remember, a correlation is between -1 and 1.

The closer the correlation is to 1, the stronger the linear relationship between the variables, with a positive slope.

The closer the correlation is to -1, the stronger the linear relationship between the variables, with a negative slope.

So it looks like the bigger the mileage is, the lower the price of the car.
The bigger the year is, the higher the price of the car.

Makes sense!!

1.3 Y and x, Features

We often use "y" to generically denote the variable we trying to predict and "x" to denote the variables we can use to predict y.

In our example y=price and x=(mileage,year).

x=(mileage,year) is the what we know about the car. Given this knowledge, what is our guess for the price of the car.

As we have done above, x is also often called the *features*.

1.4 Get y=price and X=(mileage,year) as Numpy ndarrays

Let's get a numpy array X whose 2 columns are the explanatory features *mileage* and *year*.

Let's also get a numpy array with just the target variable y = *price*.

```
In [9]: X = cd[['mileage','year']].to_numpy()  #mileage and year columns as a numpy array
        print("*** type of X is",type(X))
        print(X.shape) #number of rows and columns
        print(X[0:4,:]) #first 4 rows
        y = cd['price'].values #price as a numpy vector
        print(len(y))
        print(y[0:4])
```

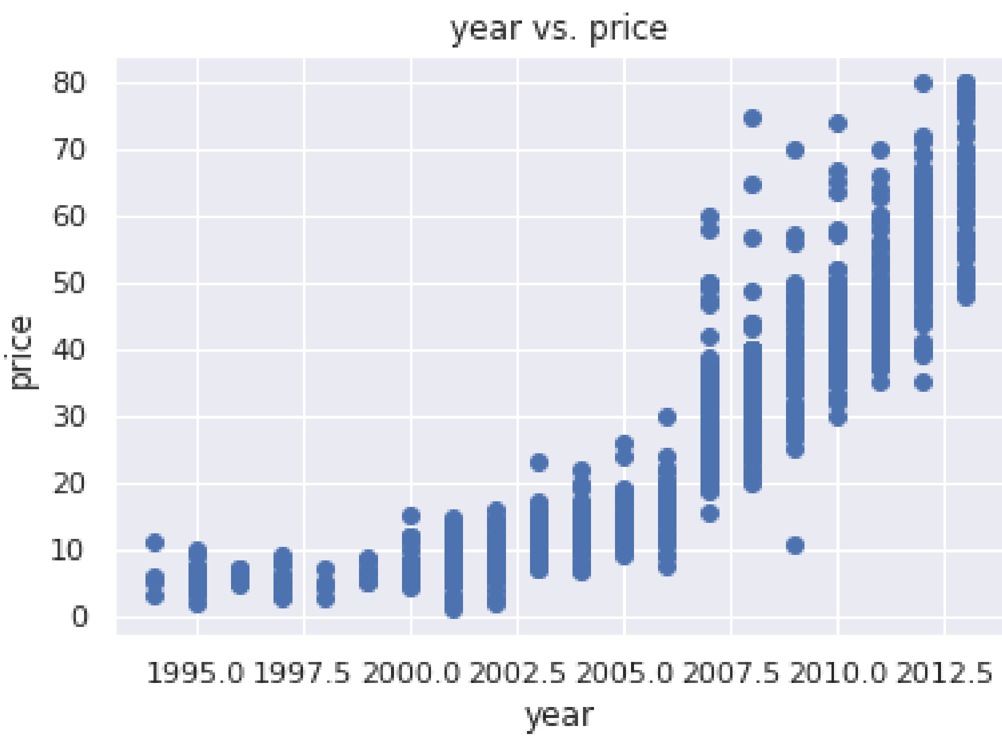
```
*** type of X is <class 'numpy.ndarray'>
(1000, 2)
[[  36.858 2008.   ]
 [  46.883 2012.   ]
 [ 108.759 2007.   ]
 [  35.187 2007.   ]]
1000
[43.995 44.995 25.999 33.88 ]
```

1.5 Plot y vs each x

Now let's plot year vs. price.

```
In [10]: plt.scatter(X[:,1],y)
         plt.xlabel("year")
         plt.ylabel("price")
         plt.title("year vs. price")
```

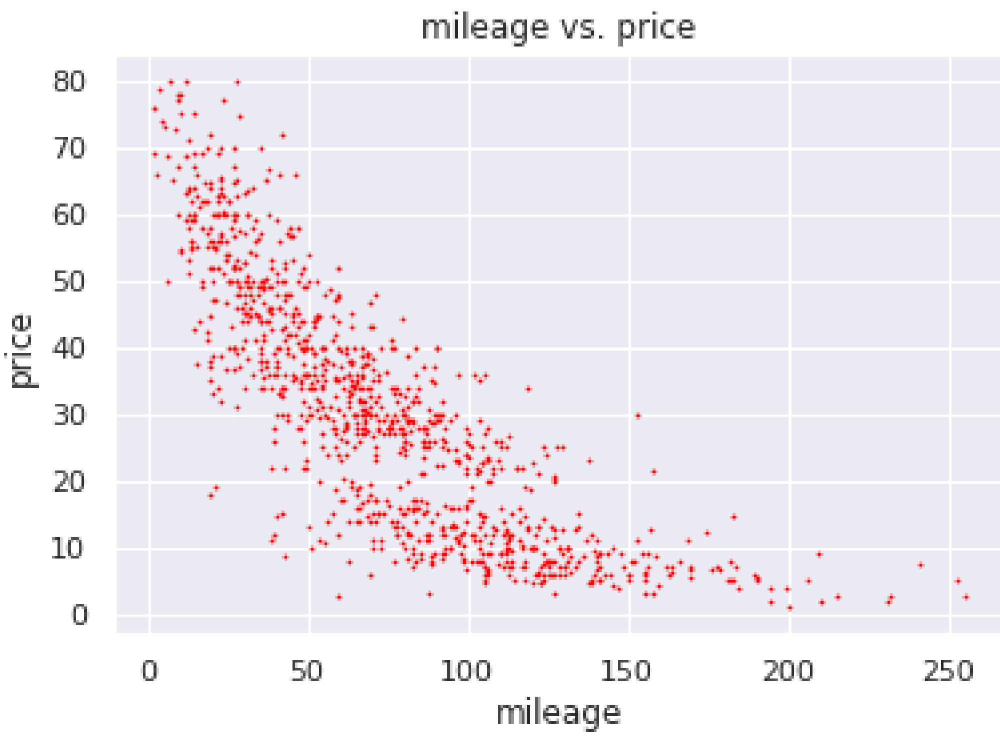
```
Out[10]: Text(0.5, 1.0, 'year vs. price')
```



And mileage vs. price.
Let's change the size of the plotted symbol and the color of the plotted symbol.

```
In [11]: plt.scatter(X[:,0],y,s=.5,c="red")
         plt.xlabel("mileage")
         plt.ylabel("price")
         plt.title("mileage vs. price")
```

```
Out[11]: Text(0.5, 1.0, 'mileage vs. price')
```



Clearly, price is related to both year and mileage.
Clearly, the relationship is not linear !!!

What we really want to **learn** is the joint relationship betwee *price* and the pair of variables (*mileage*,*year*) !!!

Essentially, the modern statistical tools or *Machine Learning* enables us to learn the relationships from data without making strong assumptions.

In the expression:

$$price = f(mileage, year)$$

we would like to know the function f.

1.5.1 plot with pandas

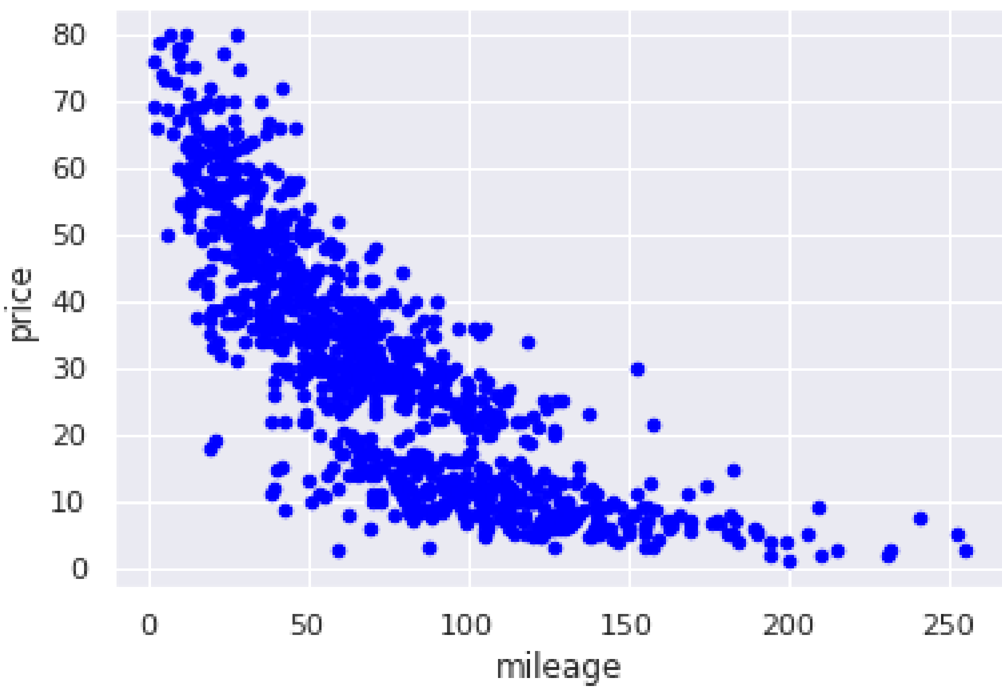
You can do a lot of the plotting directly in pandas (without getting a numpy array).

```
In [12]: Xdf = cd[['mileage','year','price']]
         Xdf.head()
```

```
Out[12]:    mileage  year   price
         0   36.858  2008  43.995
         1   46.883  2012  44.995
         2  108.759  2007  25.999
         3   35.187  2007  33.880
         4   48.153  2007  34.895
```

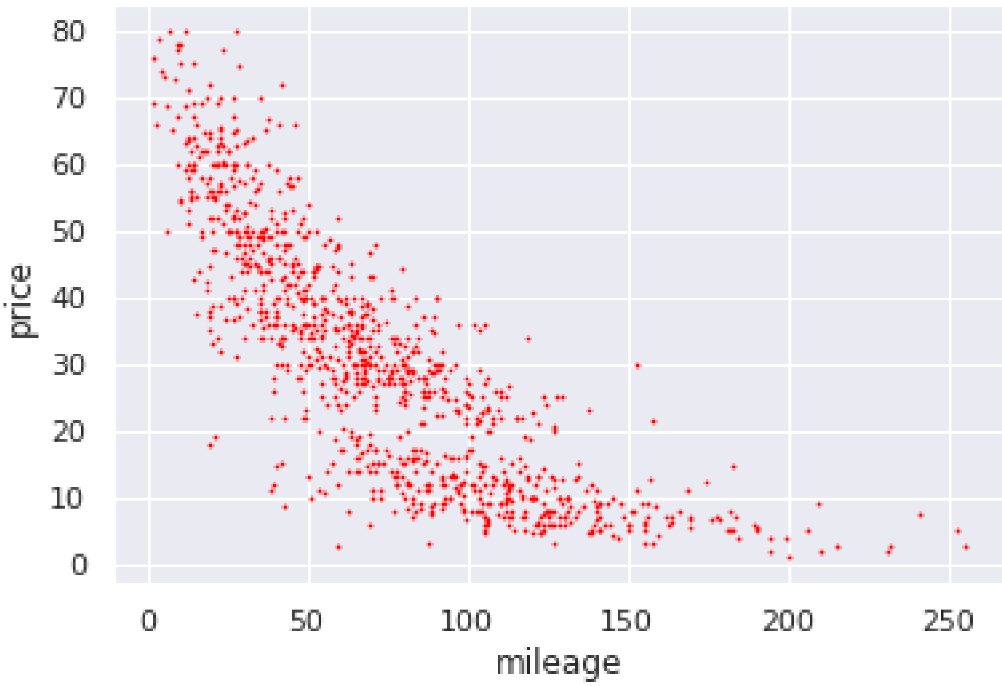
```
In [13]: Xdf.plot.scatter(0,2,c="blue") #access columns 0 and 2 = mileage and price
```

```
Out[13]: <AxesSubplot:xlabel='mileage', ylabel='price'>
```




```
In [14]: Xdf.plot.scatter('mileage','price',c="red",s=.5) # access columns using names
```

```
Out[14]: <AxesSubplot:xlabel='mileage', ylabel='price'>
```



1.6 Use iloc to subset a data frame

You can also use integers to pick off rows and columns using **iloc**.

```
In [15]: cd.columns.values
```

```
Out[15]: array(['price', 'mileage', 'year'], dtype=object)
```

```
In [16]: XXdf = cd.iloc[:,[2,0]] #year and price
         XXdf.head()
```

Out[16]:

	year	price
0	2008	43.995
1	2012	44.995
2	2007	25.999
3	2007	33.880
4	2007	34.895

```
In [17]: cd.iloc[0:3,[2,0]] #pick off rows and columns
```

Out[17]:

	year	price
0	2008	43.995
1	2012	44.995
2	2007	25.999

1.7 Run The Regression of y=price on X=(mileage,year)

Let's run a linear regression of *price* on *mileage* and *year*.
Our model is:

$$price = \beta_0 + \beta_1 mileage + \beta_2 year + \epsilon$$

This model assumes a linear relationship.
We already know this is a bad idea !!!
Let's go ahead and *fit* the model.
Fitting the model to data will give us estimates of the parameters $(\beta_0, \beta_1, \beta_2)$.
The error term ϵ represents the part of price we cannot know from (*mileage*,*year*).

```
In [18]: lmmod = LinearRegression(fit_intercept=True)
         lmmod.fit(X,y)
         print("Model Slopes:    ",lmmod.coef_)
         print("Model Intercept:",lmmod.intercept_)
```

```
Model Slopes:     [-0.1537219   2.69434954]
Model Intercept: -5365.489872256992
```

Note that there does not seem to be a simple regression summary in sklearn. Maybe that is a good thing !!!!.
So, the fitted relationship is

$$price = -5365.49 - 0.154\, mileage + 2.7\, year$$

1.7.1 Looking at the LinearRegression object lmmod

Let's have a quick look at the lmmod object.

```
In [19]: print(lmmod)
```

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normalize=False)
```

Above we see the basic attributed you can set when using LinearRegression.
Some are obvious, like fit_intercept controls whether or not an intercept is included in the regression.
For others you would try (i) ?lmmod (ii) Read the sklearn documentation (iii) google it, (iv) read a book.

```
In [20]: dir(lmmod) #you can always find out a lot about an object with dir() !!
```

```
Out[20]: ['__abstractmethods__',
          '__class__',
          '__delattr__',
          '__dict__',
          '__dir__',
```

```
'__doc__',
'__eq__',
'__format__',
'__ge__',
'__getattribute__',
'__getstate__',
'__gt__',
'__hash__',
'__init__',
'__init_subclass__',
'__le__',
'__lt__',
'__module__',
'__ne__',
'__new__',
'__reduce__',
'__reduce_ex__',
'__repr__',
'__setattr__',
'__setstate__',
'__sizeof__',
'__str__',
'__subclasshook__',
'__weakref__',
'_abc_cache',
'_abc_negative_cache',
'_abc_negative_cache_version',
'_abc_registry',
'_decision_function',
'_estimator_type',
'_get_param_names',
'_preprocess_data',
'_residues',
'_set_intercept',
'coef_',
'copy_X',
'fit',
'fit_intercept',
'get_params',
'intercept_',
'n_jobs',
'normalize',
'predict',
'rank_',
'score',
'set_params',
'singular_']
```

Some of the things in llmod are attributes (data structures) and some are methods (functions).

```
In [21]: print(type(lmmod.coef_))
         type(lmmod.set_params)

<class 'numpy.ndarray'>


Out[21]: method
```

So, you could do ?lmmod.set_params at a python prompt.

1.8 Get and Plot the Fits

Let's get the fitted values.

For each observation in our data set the fits are

$$\hat{price}_i = -5365.49 - 0.154\, mileage_i + 2.7\, year_i, \quad i = 1, 2, \ldots, n.$$

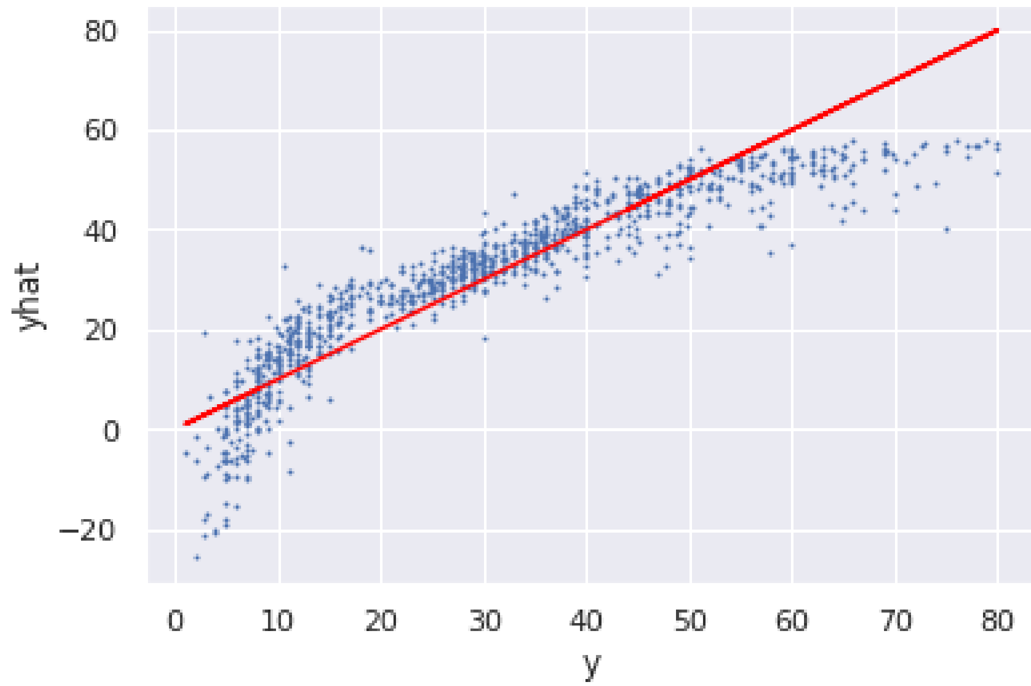
You can think of the fit as the predicted price give the values of *mileage* and *year* according to the model.

```
In [22]: yhat = lmmod.predict(X)
         print("the length of yhat is",len(yhat))
         print("the type of yhat is:")
         print(type(yhat))

the length of yhat is 1000
the type of yhat is:
<class 'numpy.ndarray'>


In [23]: plt.scatter(y,yhat,s=.8)
         plt.plot(y,y,c='red') #add the line
         plt.xlabel("y"); plt.ylabel("yhat")

Out[23]: Text(0, 0.5, 'yhat')
```



Clearly, it is really bad !!!
Machine Learning will enable us to get it right fairly automatically.

1.9 Predictions

Let's get predictions for x not in our training data.
We will make a numpy array whose rows have the x values we want to predict at.

```
In [24]: Xp = np.array([[40,2010],[100,2004]],dtype=float)
         print(Xp)
         print(type(Xp))
         print(Xp.dtype)
```

```
[[  40. 2010.]
 [ 100. 2004.]]
<class 'numpy.ndarray'>
float64
```

So, the first car has 40 (thousand) miles on it and is a 2010, while the second car has 100 (thousand) miles on it and is a 2004.
Clearly, we expect the second car to sell for less!

```
In [25]: ypred = lmmod.predict(Xp)
         print(ypred)
```

```
[44.00383414 18.61442272]
```

So we predict (based on the linear model) that the first car will sell for 44 (thou) and the second car will sell for 18.6.

Let's check the first one "by hand".

Model Slopes: [-0.1537219 2.69434954]
Model Intercept: -5365.489872256993

So the prediction for the first car in `Xp` should be:

```
In [26]: -5365.49 - .1537*40 + 2.69434954*2010

Out[26]: 44.00457540000025
```

which is correct.

1.10 In-sample/out of sample, training data

The data we used to "fit" our model, is called the *training data*.
When we look at predictions for observations in the training data (as we did for `yhat`) we say we are looking at *in-sample* fits.

When we predict at observations not in the training data (as we did for `ypred`), then we are predicting *out of sample*.

Out of sample prediction is always a more interesting test since you have not seen an example. When you predict in-sample, the training data shows the model an example of what can happen at the feature values.

1.11 scikit-learn

Linear Regression is a basic model.
There are many modeling approaches in Machine Learning !!
scikit-learn has a nice general approach to working with models: * a model will have a set of hyperparameters (e.g. lmmod.fit_intercept) * given the hyparameters, the model can learn from training data (e.g lmmod.fit(X,y)) * given a model has learned, it can make predictions (e.g. lmmod.predict(Xp))

All the predictive models in scikit-learn use the basic setup.

1.12 Standard Regression Output

From our linear regression fit using sklearn, we got estimates for the parameters.

Often we want to know a lot more about the model fit.

In particular, we might want to know the *standard errors* associated with the parameter estimates.

To get the usual *regression ouput* we can us the python package statsmodels, imported above as sm.

```
In [27]: X = sm.add_constant(X) #appends 1 to beginning of each row for the intercept
         print(X[0:3,:]) # you can see the 1's
         results = sm.OLS(y, X).fit() #run the regression
         print(results.summary()) # print out the usual summaries
```

```
[[1.00000e+00 3.68580e+01 2.00800e+03]
 [1.00000e+00 4.68830e+01 2.01200e+03]
 [1.00000e+00 1.08759e+02 2.00700e+03]]
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.832
Model:                            OLS   Adj. R-squared:                  0.832
Method:                 Least Squares   F-statistic:                     2477.
Date:                Tue, 19 Jan 2021   Prob (F-statistic):               0.00
Time:                        06:14:52   Log-Likelihood:                -3438.1
No. Observations:                1000   AIC:                             6882.
Df Residuals:                     997   BIC:                             6897.
Df Model:                           2
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const      -5365.4899    171.567    -31.273      0.000   -5702.164   -5028.816
x1            -0.1537      0.008    -18.435      0.000      -0.170      -0.137
x2             2.6943      0.085     31.602      0.000       2.527       2.862
==============================================================================
Omnibus:                      171.937   Durbin-Watson:                   2.021
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              294.618
Skew:                           1.076   Prob(JB):                     1.06e-64
Kurtosis:                       4.562   Cond. No.                     1.44e+06
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 1.44e+06. This might indicate that there are
strong multicollinearity or other numerical problems.
```

Lot's of junk!!
In particular, the standard error associate with the estimate of the slope for *mileage* is .008.
The confidence interval for β_1, the *mileage* slope is:

```
In [28]: -0.1537 + np.array([-2,2])*0.008
```

```
Out[28]: array([-0.1697, -0.1377])
```

Recall that R^2 is the square of the correlation between y and $\hat{y}$:

```
In [29]: yyhat = np.column_stack([y,yhat])
         print(yyhat.shape)
         pd.DataFrame(yyhat).corr()
```

```
(1000, 2)
```

```
Out[29]:          0        1
         0  1.00000  0.91239
         1  0.91239  1.00000
```

```
In [30]: .91239**2
```

```
Out[30]: 0.8324555121
```

Which is the same as the `R-squared` in the regression output.

1.13 Regression In Matrix Notation

Let's write our multiple regression model using vector/matrix notation and use basic matrix operations to check the predicted and fitted values.

The general multiple regression model is written:

$$Y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \dots \beta_p x_{ip} + \epsilon_i, \; i = 1, 2, \dots, n,$$

where i indexes observations and x_{ij} is the value of the j^{th} x in the i^{th} observation.

If we let

$$x_i = \begin{bmatrix} 1 \\ x_{i1} \\ x_{i2} \\ \vdots \\ x_{ip} \end{bmatrix}, \; X = \begin{bmatrix} x_1' \\ x_2' \\ \vdots \\ x_n' \end{bmatrix}, \; y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}, \; \epsilon = \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_n \end{bmatrix}, \; \beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \vdots \\ \beta_p \end{bmatrix} \tag{1}$$

,

then we can write the model in matrix form:

$$y = X\beta + \epsilon.$$

In our data, the first three rows of X are

```
In [31]: X[0:3,:]
```

```
Out[31]: array([[1.00000e+00, 3.68580e+01, 2.00800e+03],
                [1.00000e+00, 4.68830e+01, 2.01200e+03],
                [1.00000e+00, 1.08759e+02, 2.00700e+03]])
```

Which correspond to the the first three rows of our data frame cd:

```
In [32]: cd.iloc[0:3,1:3]
```

```
Out[32]:    mileage  year
         0   36.858  2008
         1   46.883  2012
         2  108.759  2007
```

Given our estimates:

$$\hat{\beta} = \begin{bmatrix} \hat{\beta}_0 \\ \hat{\beta}_1 \\ \vdots \\ \hat{\beta}_p \end{bmatrix} \tag{2}$$

We can get fitted values or predictions by matrix multiplication:

$$\hat{y} = X\,\hat{\beta}, \text{ or, } \hat{y}_p = X_p\,\hat{\beta}.$$

In our example,

```
In [33]: bhat = np.hstack([lmmod.intercept_,lmmod.coef_])[:,np.newaxis]
         print(bhat.shape)
         bhat

(3, 1)


Out[33]: array([[-5.36548987e+03],
                [-1.53721903e-01],
                [ 2.69434954e+00]])
```

So we can get our predictions by multiplying X_p times $\hat{\beta}$.
But first we have to add the column of ones:

```
In [34]: Xpp = np.hstack([np.ones((2,1)),Xp])
         print("Xp:\n",Xp)
         print("Xpp:\n",Xpp)

Xp:
 [[  40. 2010.]
 [ 100. 2004.]]
Xpp:
 [[1.000e+00 4.000e+01 2.010e+03]
 [1.000e+00 1.000e+02 2.004e+03]]
```

Now we can matrix multiply X_{pp} times $\hat{\beta}$:

```
In [35]: yhatp = Xpp @ bhat # Xpp * bhat, matrix multiplication
         yhatp

Out[35]: array([[44.00383414],
                [18.61442272]])
```

This is the same as what we got using the predict method on the lmmod object.
Let's get the in-sample fitted values by multiplying $X\hat{\beta}$:

```
In [36]: yhatm = X @ bhat
         print(yhatm[0:3,:])
         print(yhat[0:3]) #got these ones using the predict method
```

```
[[39.09812927]
 [48.33446537]
 [25.3510212 ]]
[39.09812927 48.33446537 25.3510212 ]
```

```
In [37]: dyhat = yhatm.flatten() - yhat
         dyhat.shape
```

```
Out[37]: (1000,)
```

```
In [38]: dyhat.mean()
```

```
Out[38]: 0.0
```

```
In [39]: dyhat.var()
```

```
Out[39]: 0.0
```

dyhat has 0 mean and variance, so it must be all zeros.
Just for fun we can plot yhat vs yhatm:

```
In [40]: plt.scatter(yhat,yhatm)
         plt.scatter(yhat,yhat,color='red',s=.5)
         plt.show()
```