Lab: Linear Regression

Set up IPython libraries for customizing notebook display

```
from notebookfuncs import *
```

Import standard libraries

```
import numpy as np
import pandas as pd
from matplotlib.pyplot import subplots
```

New imports

```
import statsmodels.api as sm
```

Import statsmodels objects

```
from statsmodels.stats.outliers_influence import variance_inflation_factor as VIF
from statsmodels.stats.anova import anova_lm
```

Import ISLP objects

```
from ISLP import load_data
from ISLP.models import ModelSpec as MS, summarize, poly
```

Import User Functions

```python
from userfuncs import *
```

Inspecting objects and namespaces

```python
dir()
```

```
['Audio',
 'In',
 'InteractiveShell',
 'MS',
 'Markdown',
 'Math',
 'Out',
 'VIF',
 '_',
 '__',
 '___',
 '__builtin__',
 '__builtins__',
 '__doc__',
 '__loader__',
 '__name__',
 '__package__',
 '__session__',
 '__spec__',
 '_dh',
 '_i',
 '_i1',
 '_i2',
 '_i3',
 '_i4',
 '_i5',
 '_i6',
 '_i7',
 '_ih',
 '_ii',
 '_iii',
```

```
 '_oh',
 'allDone',
 'anova_lm',
 'calculate_VIFs',
 'display',
 'display_DFFITS_plot',
 'display_cooks_distance_plot',
 'display_hat_leverage_cutoffs',
 'display_hat_leverage_plot',
 'display_residuals_plot',
 'display_studentized_residuals',
 'dmatrices',
 'exit',
 'get_influence_points',
 'get_ipython',
 'get_results_df',
 'identify_highest_VIF_feature',
 'identify_least_significant_feature',
 'influence_plot',
 'is_numeric_dtype',
 'load_data',
 'np',
 'open',
 'pd',
 'perform_analysis',
 'poly',
 'printmd',
 'px',
 'quit',
 'sm',
 'smf',
 'standardize',
 'stats',
 'subplots',
 'summarize']
```

```
A = np.array([3, 5, 11])
dir(A)
```

```
['T',
 '__abs__',
 '__add__',
```

```
'__and__',
'__array__',
'__array_finalize__',
'__array_function__',
'__array_interface__',
'__array_prepare__',
'__array_priority__',
'__array_struct__',
'__array_ufunc__',
'__array_wrap__',
'__bool__',
'__buffer__',
'__class__',
'__class_getitem__',
'__complex__',
'__contains__',
'__copy__',
'__deepcopy__',
'__delattr__',
'__delitem__',
'__dir__',
'__divmod__',
'__dlpack__',
'__dlpack_device__',
'__doc__',
'__eq__',
'__float__',
'__floordiv__',
'__format__',
'__ge__',
'__getattribute__',
'__getitem__',
'__getstate__',
'__gt__',
'__hash__',
'__iadd__',
'__iand__',
'__ifloordiv__',
'__ilshift__',
'__imatmul__',
'__imod__',
'__imul__',
'__index__',
```

```
'__init__',
'__init_subclass__',
'__int__',
'__invert__',
'__ior__',
'__ipow__',
'__irshift__',
'__isub__',
'__iter__',
'__itruediv__',
'__ixor__',
'__le__',
'__len__',
'__lshift__',
'__lt__',
'__matmul__',
'__mod__',
'__mul__',
'__ne__',
'__neg__',
'__new__',
'__or__',
'__pos__',
'__pow__',
'__radd__',
'__rand__',
'__rdivmod__',
'__reduce__',
'__reduce_ex__',
'__repr__',
'__rfloordiv__',
'__rlshift__',
'__rmatmul__',
'__rmod__',
'__rmul__',
'__ror__',
'__rpow__',
'__rrshift__',
'__rshift__',
'__rsub__',
'__rtruediv__',
'__rxor__',
'__setattr__',
```

```
'__setitem__',
'__setstate__',
'__sizeof__',
'__str__',
'__sub__',
'__subclasshook__',
'__truediv__',
'__xor__',
'all',
'any',
'argmax',
'argmin',
'argpartition',
'argsort',
'astype',
'base',
'byteswap',
'choose',
'clip',
'compress',
'conj',
'conjugate',
'copy',
'ctypes',
'cumprod',
'cumsum',
'data',
'diagonal',
'dot',
'dtype',
'dump',
'dumps',
'fill',
'flags',
'flat',
'flatten',
'getfield',
'imag',
'item',
'itemset',
'itemsize',
'max',
'mean',
```

```
 'min',
 'nbytes',
 'ndim',
 'newbyteorder',
 'nonzero',
 'partition',
 'prod',
 'ptp',
 'put',
 'ravel',
 'real',
 'repeat',
 'reshape',
 'resize',
 'round',
 'searchsorted',
 'setfield',
 'setflags',
 'shape',
 'size',
 'sort',
 'squeeze',
 'std',
 'strides',
 'sum',
 'swapaxes',
 'take',
 'tobytes',
 'tofile',
 'tolist',
 'tostring',
 'trace',
 'transpose',
 'var',
 'view']
```

```
A.sum()
```

```
19
```

Simple Linear Regression

We will use the Boston housing dataset which is in the package ISLP

```
Boston = load_data("Boston")
Boston.columns
```

```
Index(['crim', 'zn', 'indus', 'chas', 'nox', 'rm', 'age', 'dis', 'rad', 'tax',
       'ptratio', 'lstat', 'medv'],
      dtype='object')
```

```
len(Boston.columns)
```

```
13
```

```
Boston?
```

```
Type:        DataFrame
String form:
crim    zn  indus  chas   nox    rm   age    dis  rad  tax  \
           0    0.00632  18.0   2.3 <...> 0   5.64  23.9
           504     21.0   6.48  22.0
           505     21.0   7.88  11.9

           [506 rows x 13 columns]
Length:      506
File:        ~/ISLP/islpenv/lib/python3.12/site-packages/pandas/core/frame.py
Docstring:
Two-dimensional, size-mutable, potentially heterogeneous tabular data.

Data structure also contains labeled axes (rows and columns).
Arithmetic operations align on both row and column labels. Can be
thought of as a dict-like container for Series objects. The primary
pandas data structure.

Parameters
----------
data : ndarray (structured or homogeneous), Iterable, dict, or DataFrame
    Dict can contain Series, arrays, constants, dataclass or list-like objects. If
    data is a dict, column order follows insertion-order. If a dict contains Series
```

```
    which have an index defined, it is aligned by its index. This alignment also
    occurs if data is a Series or a DataFrame itself. Alignment is done on
    Series/DataFrame inputs.

    If data is a list of dicts, column order follows insertion-order.

index : Index or array-like
    Index to use for resulting frame. Will default to RangeIndex if
    no indexing information part of input data and no index provided.
columns : Index or array-like
    Column labels to use for resulting frame when data does not have them,
    defaulting to RangeIndex(0, 1, 2, ..., n). If data contains column labels,
    will perform column selection instead.
dtype : dtype, default None
    Data type to force. Only a single dtype is allowed. If None, infer.
copy : bool or None, default None
    Copy data from inputs.
    For dict data, the default of None behaves like ``copy=True``.  For DataFrame
    or 2d ndarray input, the default of None behaves like ``copy=False``.
    If data is a dict containing one or more Series (possibly of different dtypes),
    ``copy=False`` will ensure that these inputs are not copied.

    .. versionchanged:: 1.3.0

See Also
--------
DataFrame.from_records : Constructor from tuples, also record arrays.
DataFrame.from_dict : From dicts of Series, arrays, or dicts.
read_csv : Read a comma-separated values (csv) file into DataFrame.
read_table : Read general delimited file into DataFrame.
read_clipboard : Read text from clipboard into DataFrame.

Notes
-----
Please reference the :ref:`User Guide <basics.dataframe>` for more information.

Examples
--------
Constructing DataFrame from a dictionary.

>>> d = {'col1': [1, 2], 'col2': [3, 4]}
>>> df = pd.DataFrame(data=d)
>>> df
```

```
   col1  col2
0     1     3
1     2     4
```

Notice that the inferred dtype is int64.

```
>>> df.dtypes
col1    int64
col2    int64
dtype: object
```

To enforce a single dtype:

```
>>> df = pd.DataFrame(data=d, dtype=np.int8)
>>> df.dtypes
col1    int8
col2    int8
dtype: object
```

Constructing DataFrame from a dictionary including Series:

```
>>> d = {'col1': [0, 1, 2, 3], 'col2': pd.Series([2, 3], index=[2, 3])}
>>> pd.DataFrame(data=d, index=[0, 1, 2, 3])
   col1  col2
0     0   NaN
1     1   NaN
2     2   2.0
3     3   3.0
```

Constructing DataFrame from numpy ndarray:

```
>>> df2 = pd.DataFrame(np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]),
...                    columns=['a', 'b', 'c'])
>>> df2
   a  b  c
0  1  2  3
1  4  5  6
2  7  8  9
```

Constructing DataFrame from a numpy ndarray that has labeled columns:

```
>>> data = np.array([(1, 2, 3), (4, 5, 6), (7, 8, 9)],
...                 dtype=[("a", "i4"), ("b", "i4"), ("c", "i4")])
```

```
>>> df3 = pd.DataFrame(data, columns=['c', 'a'])
...
>>> df3
   c  a
0  3  1
1  6  4
2  9  7
```

Constructing DataFrame from dataclass:

```
>>> from dataclasses import make_dataclass
>>> Point = make_dataclass("Point", [("x", int), ("y", int)])
>>> pd.DataFrame([Point(0, 0), Point(0, 3), Point(2, 3)])
   x  y
0  0  0
1  0  3
2  2  3
```

Constructing DataFrame from Series/DataFrame:

```
>>> ser = pd.Series([1, 2, 3], index=["a", "b", "c"])
>>> df = pd.DataFrame(data=ser, index=["a", "c"])
>>> df
   0
a  1
c  3

>>> df1 = pd.DataFrame([1, 2, 3], index=["a", "b", "c"], columns=["x"])
>>> df2 = pd.DataFrame(data=df1, index=["a", "c"])
>>> df2
   x
a  1
c  3
```

Use sm.OLS to fit a simple linear regression

```
X = pd.DataFrame({"intercept": np.ones(Boston.shape[0]), "lstat": Boston["lstat"]})
X.head()
```

	intercept	lstat
0	1.0	4.98
1	1.0	9.14
2	1.0	4.03
3	1.0	2.94
4	1.0	5.33

Extract the response and fit the model.

```
y = Boston["medv"]
model = sm.OLS(y, X)
results = model.fit()
```

```
<statsmodels.regression.linear_model.RegressionResultsWrapper at 0x7b0e81f6ade0>
```

Summarize the results using the ISLP method summarize

```
summarize(results)
```

	coef	std err	t	P>|t|
intercept	34.5538	0.563	61.415	0.0
lstat	-0.9500	0.039	-24.528	0.0

Using Transformations: Fit and Transform

```
design = MS(["lstat"])
design = design.fit(Boston)
X = design.transform(Boston)
X.head()
```

	intercept	lstat
0	1.0	4.98
1	1.0	9.14

	intercept	lstat
2	1.0	4.03
3	1.0	2.94
4	1.0	5.33

```
design = MS(["lstat"])
design = design.fit_transform(Boston)
X.head()
```

	intercept	lstat
0	1.0	4.98
1	1.0	9.14
2	1.0	4.03
3	1.0	2.94
4	1.0	5.33

Full and exhaustive summary of the fit

```
results.summary()
```

Dep. Variable:	medv	**R-squared:**	0.544
Model:	OLS	**Adj. R-squared:**	0.543
Method:	Least Squares	**F-statistic:**	601.6
Date:	Tue, 22 Oct 2024	**Prob (F-statistic):**	5.08e-88
Time:	12:45:15	**Log-Likelihood:**	-1641.5
No. Observations:	506	**AIC:**	3287.
Df Residuals:	504	**BIC:**	3295.
Df Model:	1		
Covariance Type:	nonrobust		

	coef	**std err**	**t**	**P> |t|**	**[0.025**	**0.975]**
intercept	34.5538	0.563	61.415	0.000	33.448	35.659
lstat	-0.9500	0.039	-24.528	0.000	-1.026	-0.874

Omnibus:	137.043	**Durbin-Watson:**	0.892
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	291.373
Skew:	1.453	**Prob(JB):**	5.36e-64
Kurtosis:	5.319	**Cond. No.**	29.7

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

Fitted coefficients can be retrieved as the *params* attribute of results

```
results.params
```

```
intercept    34.553841
lstat        -0.950049
dtype: float64
```

Computing predictions

```
design = MS(["lstat"])
new_df = pd.DataFrame({"lstat": [5, 10, 15]})
print(new_df)
design = design.fit(new_df)
newX = design.transform(new_df)
newX
```

```
   lstat
0      5
1     10
2     15
```

	intercept	lstat
0	1.0	5
1	1.0	10
2	1.0	15

```
new_predictions = results.get_prediction(newX)
new_predictions.predicted_mean
```

```
array([29.80359411, 25.05334734, 20.30310057])
```

We can predict confidence intervals for the predicted values.

```
new_predictions.conf_int(alpha=0.05)
```

```
array([[29.00741194, 30.59977628],
       [24.47413202, 25.63256267],
       [19.73158815, 20.87461299]])
```

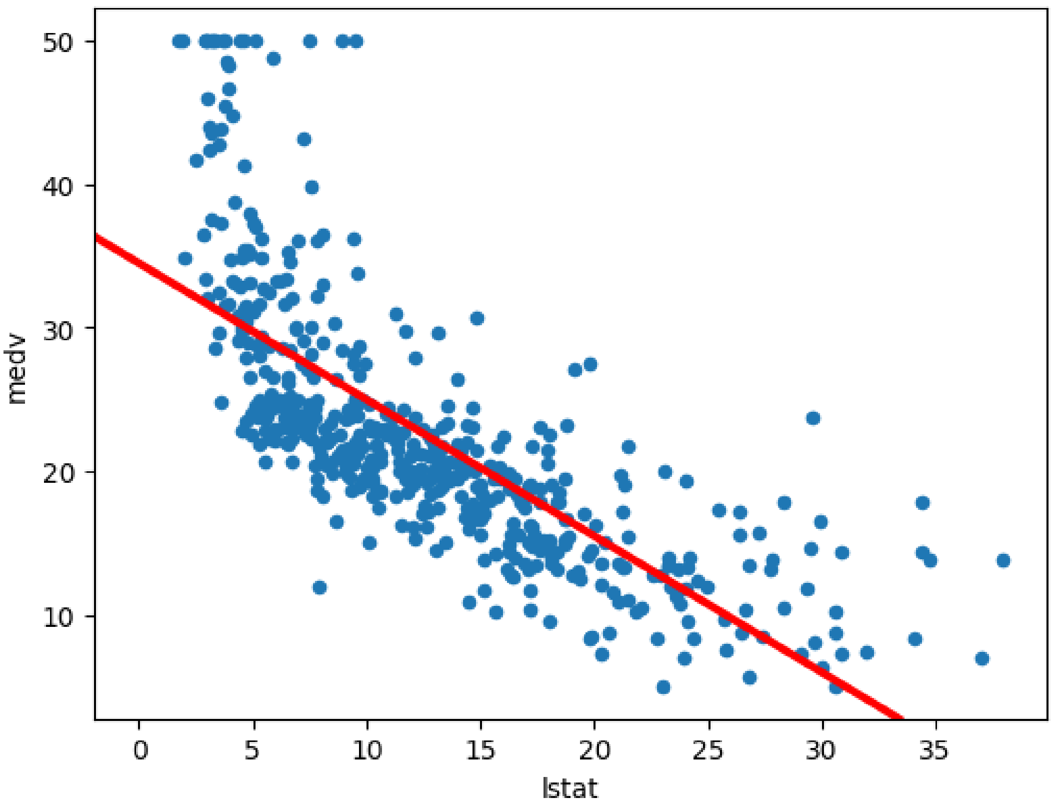
We can obtain prediction intervals for the values which are wider than the confidence intervals since they're for a specific instance of lstat by setting obs=True.

```
new_predictions.conf_int(obs=True, alpha=0.05)
```

```
array([[17.56567478, 42.04151344],
       [12.82762635, 37.27906833],
       [ 8.0777421 , 32.52845905]])
```

Plot medv and lstat using DataFrame.plot.scatter() and add the regression line to the resulting plot.

```
ax = Boston.plot.scatter("lstat", "medv")
ax.axline(
    (ax.get_xlim()[0], results.params.iloc[0]),
    slope=results.params.iloc[1],
    color="r",
    linewidth=3,
)
```

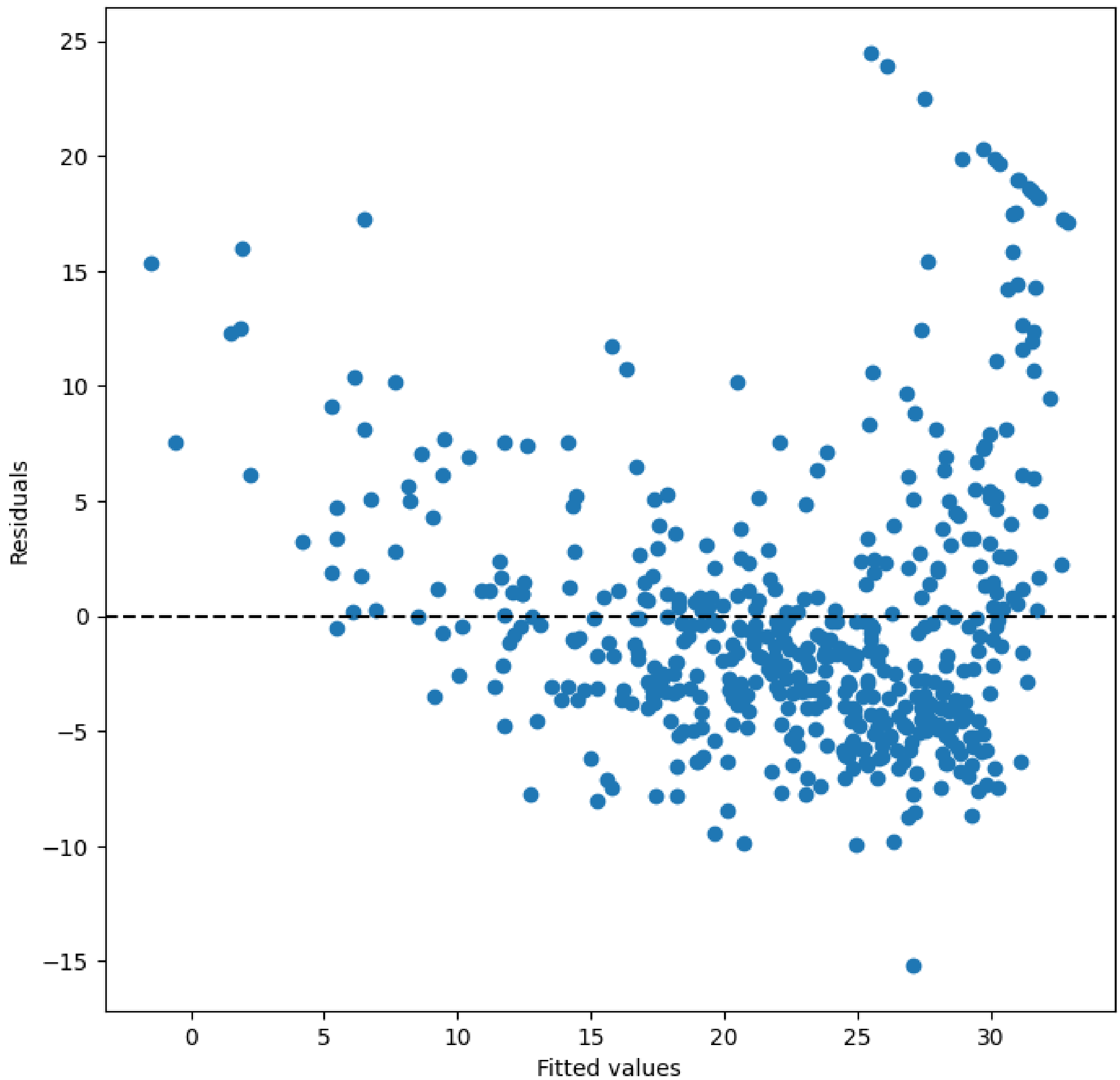


- There is some evidence of non-linearity in the relationship b/w lstat and medv.

Find the fitted values and residuals of the fit as attributes of the results object as *results.fittedvalues* and *results.resid*.

- The get_influence() method computes various influence measures of the regression.

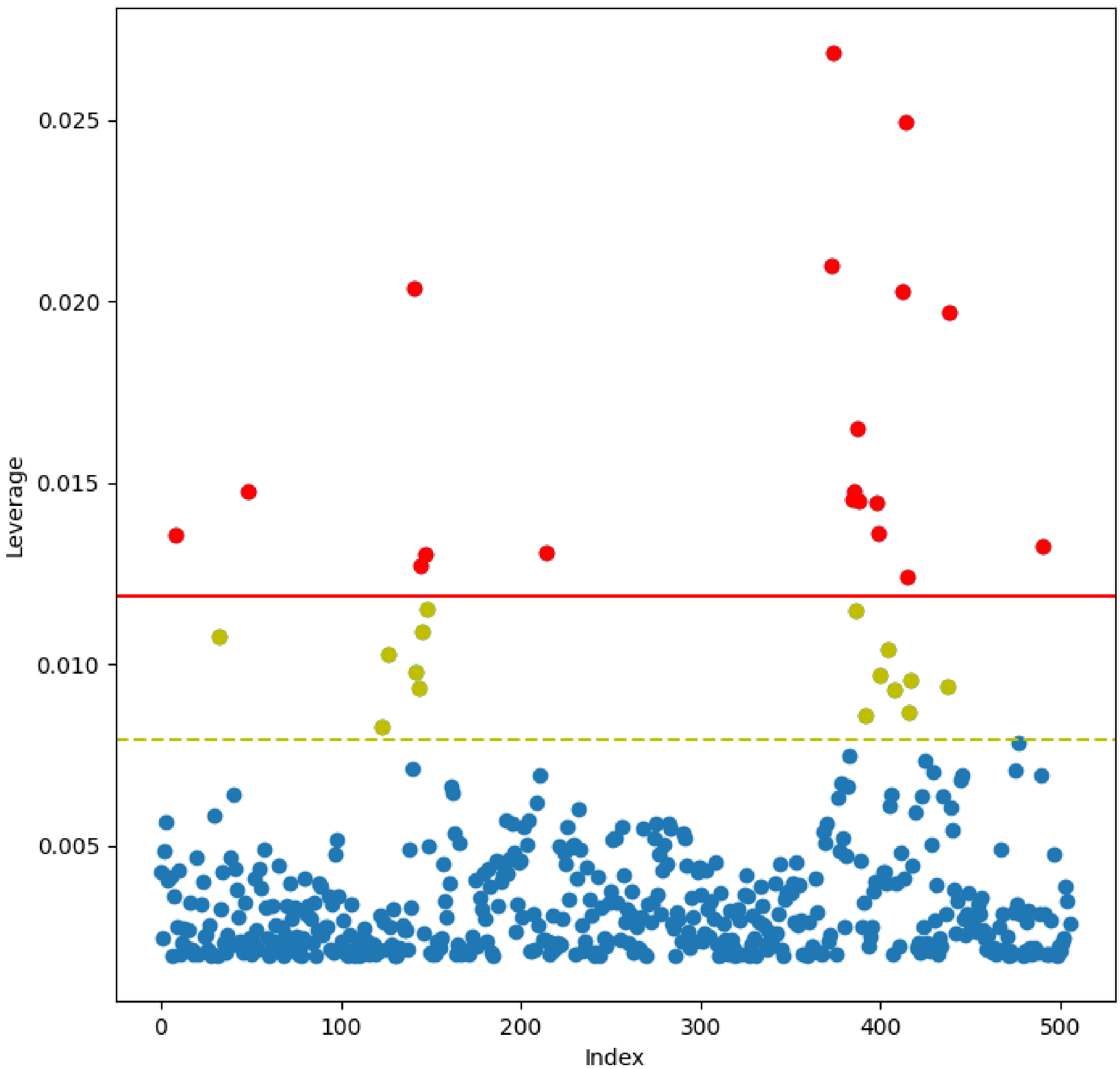
```
_, ax = subplots(figsize=(8, 8))
ax.scatter(results.fittedvalues, results.resid)
ax.set_xlabel("Fitted values")
ax.set_ylabel("Residuals")
ax.axhline(0, c="k", ls="--")
```



- On the basis of the residual plot, there is some evidence of non-linearity.

Leverage statistics can be computed for any number of predictors using the hat_matrix_diag attribute of the value returned by the get_influence() method.

```
display_hat_leverage_cutoffs(results)
```

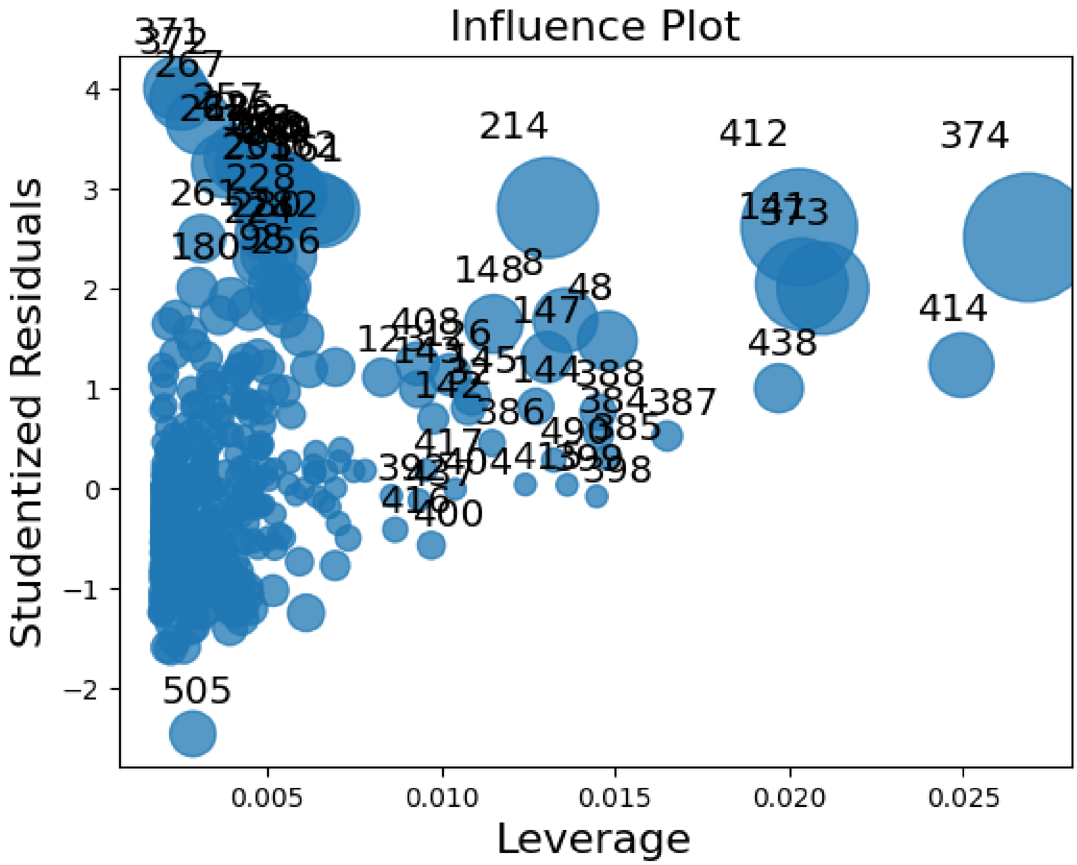


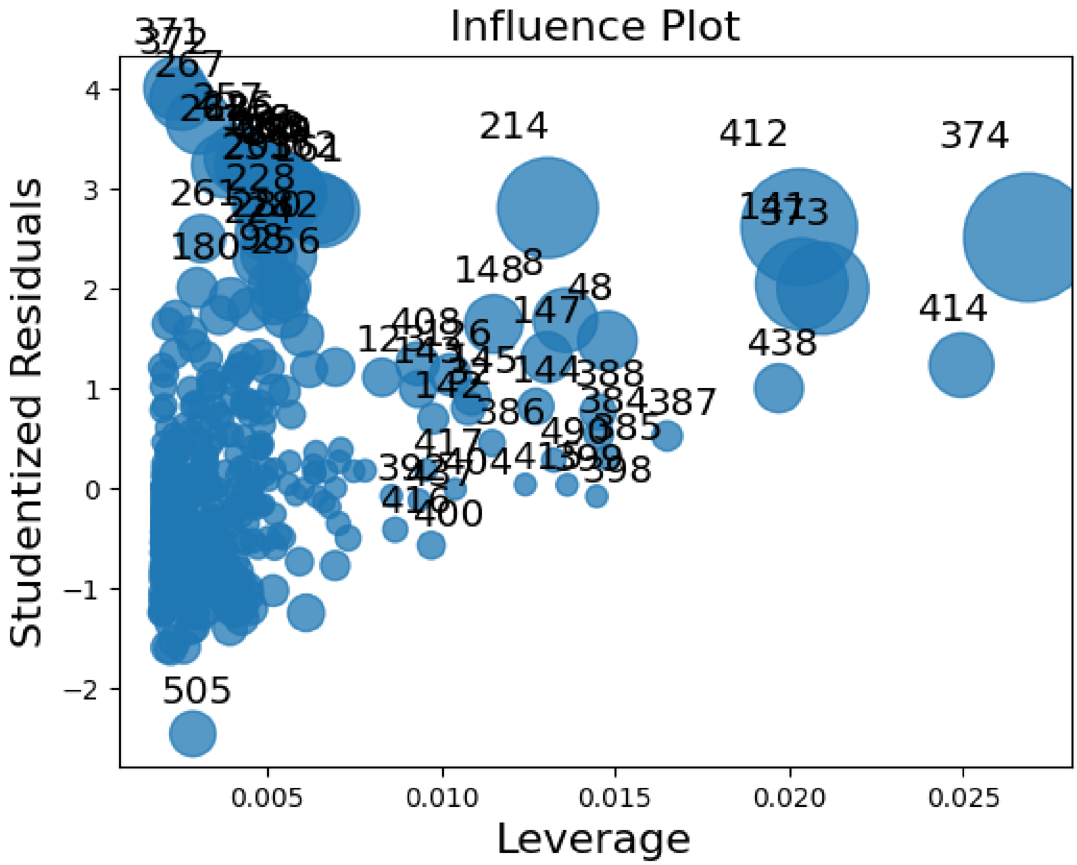

```
display_hat_leverage_plot(results)
```

Unable to display output for mime type(s): text/html

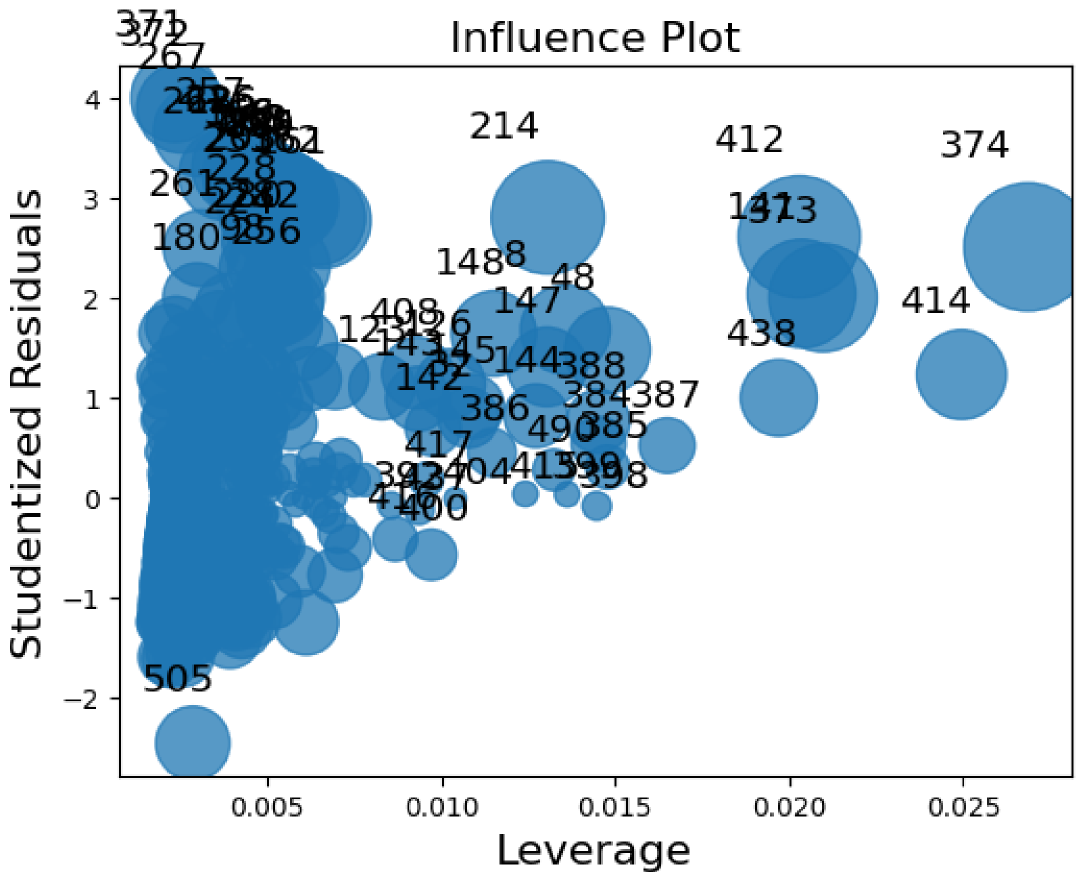
Unable to display output for mime type(s): application/vnd.plotly.v1+json, text/html

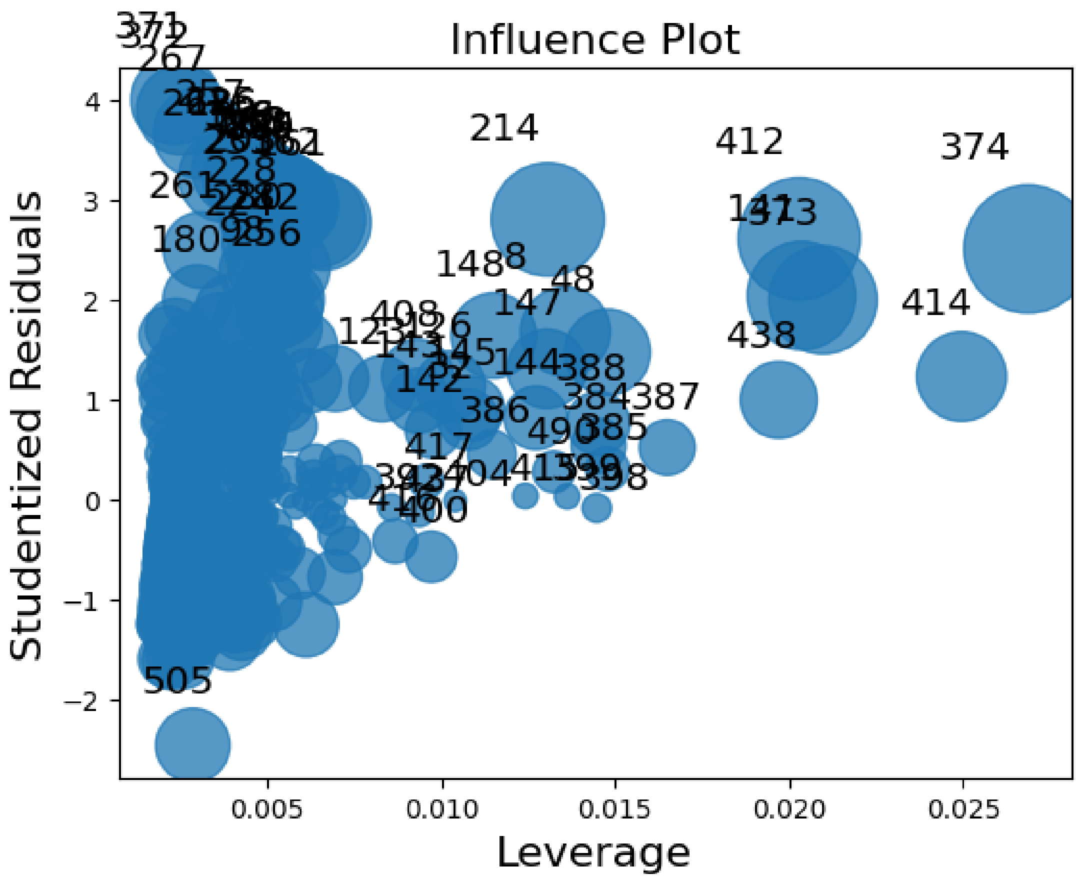
```
display_cooks_distance_plot(results)
```


Influence Plot
Studentized Residuals
Leverage
0.005
0.010
0.015
0.020
0.025
4
3
2
1
0
−1
−2
214
412
374
414
438
505
180
148
147
48
144
388
387
386
385
400




```
display_DFFITS_plot(results)
```


Influence Plot
Studentized Residuals
Leverage
4
3
2
1
0
−1
−2
0.005
0.010
0.015
0.020
0.025
267
214
412
374
180
256
148
8
48
147
408
438
414
144
388
387
386
385
417
400
398
505



```
inf_df, _ = get_influence_points(results)
inf_df
```

```
n = 506.0, p = 2
Average Hat Leverage: 0.003952569169960474
Hat Leverage  Cutoff = 2 * Average Hat Leverage = 0.007905138339920948
DFBetas Cutoff = 3 / sqrt(n) = 0.1333662673423161
DFFITS Cutoff = 2 * sqrt(p/n) = 0.1257389226923863
Cooks Distance Cutoff = 1.0
Cooks Distance p-value Cutoff = 0.05
Studentized Residuals Cutoff = 3.0
Studentized Residuals p-value Cutoff = 0.01
```

	dfb_intercept	dfb_lstat	cooks_d	hat_diag	student_resid	dffits	student_resid_pvalue	ha
161	0.226022	-0.189640	0.025315	0.006609	2.776857	0.226503	0.002847	0.
162	0.225506	-0.188310	0.025219	0.006450	2.806416	0.226112	0.002602	0.
163	0.219862	-0.176384	0.024252	0.005359	3.024654	0.222011	0.001308	0.
166	0.217770	-0.172500	0.023921	0.005089	3.084034	0.220566	0.001077	0.
186	0.212939	-0.164025	0.023201	0.004589	3.201426	0.217377	0.000727	0.
195	0.221577	-0.179716	0.024534	0.005617	2.970018	0.223224	0.001560	0.
203	0.199749	-0.157618	0.020220	0.005013	2.853126	0.202515	0.002254	0.
204	0.221985	-0.180535	0.024602	0.005685	2.955977	0.223517	0.001632	0.
214	-0.197509	0.297576	0.051465	0.013063	2.807647	0.323011	0.002592	0.
225	0.211641	-0.161830	0.023017	0.004476	3.229640	0.216554	0.000660	0.
228	0.178647	-0.140414	0.016255	0.004938	2.573814	0.181309	0.005172	0.
233	0.196791	-0.154507	0.019680	0.004918	2.841931	0.199782	0.002333	0.
257	0.207834	-0.155541	0.022502	0.004180	3.306527	0.214221	0.000506	0.
261	0.128039	-0.084194	0.009646	0.003106	2.501385	0.139616	0.006344	0.
262	0.189166	-0.136023	0.019246	0.003742	3.231093	0.198020	0.000657	0.
267	0.184377	-0.119477	0.020008	0.003032	3.672331	0.202505	0.000133	0.
280	0.164168	-0.129770	0.013717	0.005047	2.335776	0.166365	0.009947	0.
283	0.220671	-0.177936	0.024384	0.005476	2.999671	0.222580	0.001418	0.
368	0.220170	-0.176971	0.024302	0.005402	3.015284	0.222227	0.001349	0.
369	0.217594	-0.172182	0.023894	0.005068	3.088724	0.220447	0.001061	0.
370	0.221623	-0.179808	0.024542	0.005625	2.968458	0.223257	0.001568	0.
371	0.155509	-0.078029	0.018381	0.002355	4.004703	0.194572	0.000036	0.
372	0.165278	-0.091836	0.018763	0.002529	3.901020	0.196431	0.000054	0.
374	-0.294291	0.401657	0.086162	0.026865	2.511537	0.417300	0.006166	0.
412	-0.253809	0.357605	0.070029	0.020290	2.615542	0.376405	0.004588	0.

For a more conservative cutoff values for hat_diag, we have the following infuence point(s):

```
inf_df[inf_df["hat_diag"] > (3 * np.mean(inf_df["hat_diag"]))]
```

	dfb_intercept	dfb_lstat	cooks_d	hat_diag	student_resid	dffits	student_resid_pvalue	ha
374	-0.294291	0.401657	0.086162	0.026865	2.511537	0.417300	0.006166	0.
412	-0.253809	0.357605	0.070029	0.020290	2.615542	0.376405	0.004588	0.

Using DFFITS cutoff, we have the following influential points

```
inf_df[inf_df["dffits"] > 2 * np.sqrt(len(results.params) / results.nobs)]
```

	dfb_intercept	dfb_lstat	cooks_d	hat_diag	student_resid	dffits	student_resid_pvalue	ha
161	0.226022	-0.189640	0.025315	0.006609	2.776857	0.226503	0.002847	0.
162	0.225506	-0.188310	0.025219	0.006450	2.806416	0.226112	0.002602	0.
163	0.219862	-0.176384	0.024252	0.005359	3.024654	0.222011	0.001308	0.
166	0.217770	-0.172500	0.023921	0.005089	3.084034	0.220566	0.001077	0.
186	0.212939	-0.164025	0.023201	0.004589	3.201426	0.217377	0.000727	0.
195	0.221577	-0.179716	0.024534	0.005617	2.970018	0.223224	0.001560	0.
203	0.199749	-0.157618	0.020220	0.005013	2.853126	0.202515	0.002254	0.
204	0.221985	-0.180535	0.024602	0.005685	2.955977	0.223517	0.001632	0.
214	-0.197509	0.297576	0.051465	0.013063	2.807647	0.323011	0.002592	0.
225	0.211641	-0.161830	0.023017	0.004476	3.229640	0.216554	0.000660	0.
228	0.178647	-0.140414	0.016255	0.004938	2.573814	0.181309	0.005172	0.
233	0.196791	-0.154507	0.019680	0.004918	2.841931	0.199782	0.002333	0.
257	0.207834	-0.155541	0.022502	0.004180	3.306527	0.214221	0.000506	0.
261	0.128039	-0.084194	0.009646	0.003106	2.501385	0.139616	0.006344	0.
262	0.189166	-0.136023	0.019246	0.003742	3.231093	0.198020	0.000657	0.
267	0.184377	-0.119477	0.020008	0.003032	3.672331	0.202505	0.000133	0.
280	0.164168	-0.129770	0.013717	0.005047	2.335776	0.166365	0.009947	0.
283	0.220671	-0.177936	0.024384	0.005476	2.999671	0.222580	0.001418	0.
368	0.220170	-0.176971	0.024302	0.005402	3.015284	0.222227	0.001349	0.
369	0.217594	-0.172182	0.023894	0.005068	3.088724	0.220447	0.001061	0.
370	0.221623	-0.179808	0.024542	0.005625	2.968458	0.223257	0.001568	0.
371	0.155509	-0.078029	0.018381	0.002355	4.004703	0.194572	0.000036	0.
372	0.165278	-0.091836	0.018763	0.002529	3.901020	0.196431	0.000054	0.
374	-0.294291	0.401657	0.086162	0.026865	2.511537	0.417300	0.006166	0.
412	-0.253809	0.357605	0.070029	0.020290	2.615542	0.376405	0.004588	0.

Using Cooks Distance, we have the following influential points

```
inf_df[inf_df["cooks_d"] > 1.0]
```

	dfb_intercept	dfb_lstat	cooks_d	hat_diag	student_resid	dffits	student_resid_pvalue	hat_influ

Using Cooks Distance p-values, we have the following influential points

```python
inf_df[inf_df["cooks_d_pvalue"] < 0.05]
```

	dfb_intercept	dfb_lstat	cooks_d	hat_diag	student_resid	dffits	student_resid_pvalue	hat_influ

Using DFBeta for intercept, we have the following influential points

```python
inf_df[inf_df["dfb_intercept"] > (3 / np.sqrt(results.nobs))]
```

	dfb_intercept	dfb_lstat	cooks_d	hat_diag	student_resid	dffits	student_resid_pvalue	ha
161	0.226022	-0.189640	0.025315	0.006609	2.776857	0.226503	0.002847	0.
162	0.225506	-0.188310	0.025219	0.006450	2.806416	0.226112	0.002602	0.
163	0.219862	-0.176384	0.024252	0.005359	3.024654	0.222011	0.001308	0.
166	0.217770	-0.172500	0.023921	0.005089	3.084034	0.220566	0.001077	0.
186	0.212939	-0.164025	0.023201	0.004589	3.201426	0.217377	0.000727	0.
195	0.221577	-0.179716	0.024534	0.005617	2.970018	0.223224	0.001560	0.
203	0.199749	-0.157618	0.020220	0.005013	2.853126	0.202515	0.002254	0.
204	0.221985	-0.180535	0.024602	0.005685	2.955977	0.223517	0.001632	0.
225	0.211641	-0.161830	0.023017	0.004476	3.229640	0.216554	0.000660	0.
228	0.178647	-0.140414	0.016255	0.004938	2.573814	0.181309	0.005172	0.
233	0.196791	-0.154507	0.019680	0.004918	2.841931	0.199782	0.002333	0.
257	0.207834	-0.155541	0.022502	0.004180	3.306527	0.214221	0.000506	0.
262	0.189166	-0.136023	0.019246	0.003742	3.231093	0.198020	0.000657	0.
267	0.184377	-0.119477	0.020008	0.003032	3.672331	0.202505	0.000133	0.
280	0.164168	-0.129770	0.013717	0.005047	2.335776	0.166365	0.009947	0.
283	0.220671	-0.177936	0.024384	0.005476	2.999671	0.222580	0.001418	0.
368	0.220170	-0.176971	0.024302	0.005402	3.015284	0.222227	0.001349	0.
369	0.217594	-0.172182	0.023894	0.005068	3.088724	0.220447	0.001061	0.
370	0.221623	-0.179808	0.024542	0.005625	2.968458	0.223257	0.001568	0.
371	0.155509	-0.078029	0.018381	0.002355	4.004703	0.194572	0.000036	0.
372	0.165278	-0.091836	0.018763	0.002529	3.901020	0.196431	0.000054	0.

Using DFBeta for lstat, we have the following influential points

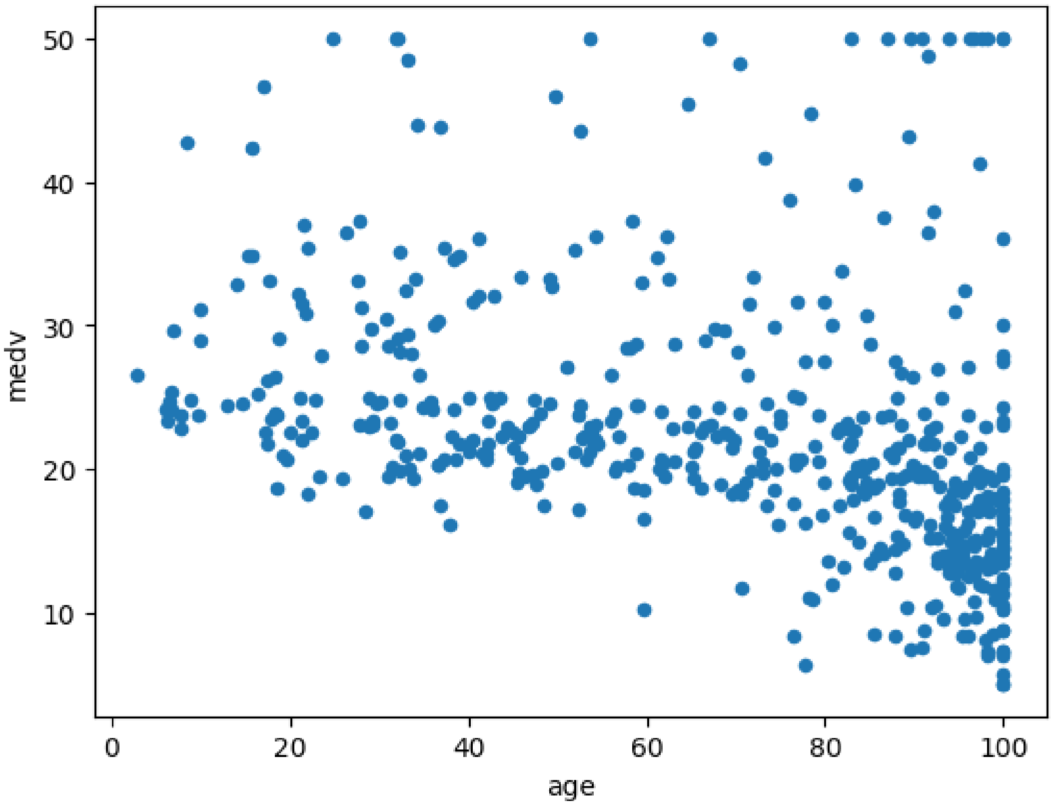
```
inf_df[inf_df["dfb_lstat"] > (3 / np.sqrt(results.nobs))]
```

	dfb_intercept	dfb_lstat	cooks_d	hat_diag	student_resid	dffits	student_resid_pvalue	ha
214	-0.197509	0.297576	0.051465	0.013063	2.807647	0.323011	0.002592	0.
374	-0.294291	0.401657	0.086162	0.026865	2.511537	0.417300	0.006166	0.
412	-0.253809	0.357605	0.070029	0.020290	2.615542	0.376405	0.004588	0.

Multiple linear regression

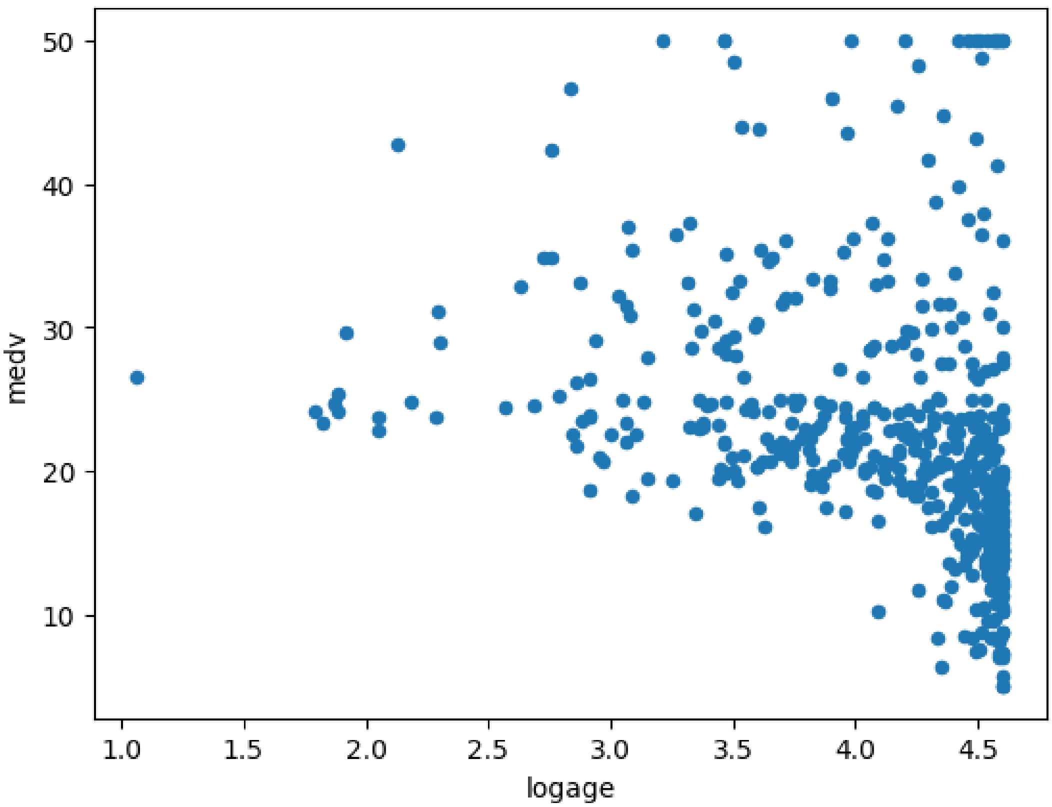
```
Boston.plot.scatter("age", "medv")
X = MS(["lstat", "age"]).fit_transform(Boston)
model1 = sm.OLS(y, X)
results1 = model1.fit()
summarize(results1)
```

	coef	std err	t	P>|t|
intercept	33.2228	0.731	45.458	0.000
lstat	-1.0321	0.048	-21.416	0.000
age	0.0345	0.012	2.826	0.005



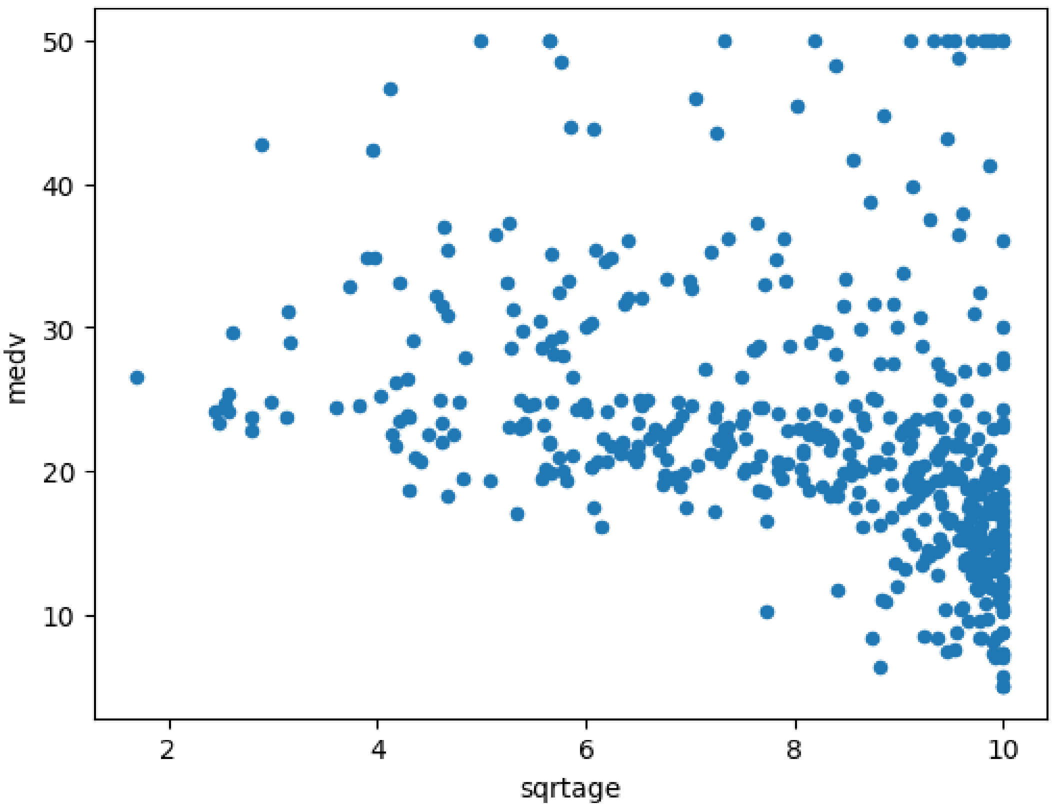

```
Boston["logage"] = np.log(Boston["age"])
Boston.plot.scatter("logage", "medv")
X = MS(["lstat", "logage"]).fit_transform(Boston)
model1 = sm.OLS(y, X)
resultslog = model1.fit()
print(summarize(resultslog))
```

```
              coef  std err       t  P>|t|
intercept  30.2143    1.947  15.517   0.00
lstat      -1.0051    0.045 -22.213   0.00
logage      1.2312    0.529   2.327   0.02
```




```
Boston["sqrtage"] = np.sqrt(Boston["age"])
Boston.plot.scatter("sqrtage", "medv")
X = MS(["lstat", "sqrtage"]).fit_transform(Boston)
model1 = sm.OLS(y, X)
resultssqrt = model1.fit()
summarize(resultssqrt)
```

	coef	std err	t	P>|t|
intercept	31.8635	1.174	27.139	0.000
lstat	-1.0203	0.047	-21.703	0.000
sqrtage	0.4450	0.171	2.606	0.009




```
Boston = Boston.drop(columns=["logage", "sqrtage"])
```

	crim	zn	indus	chas	nox	rm	age	dis	rad	tax	ptratio	lstat	medv
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	4.98	24.0
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	9.14	21.6
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	4.03	34.7
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	2.94	33.4
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	5.33	36.2
...	...	...	...	...	...	...	...	...	...	...	...	...	...
501	0.06263	0.0	11.93	0	0.573	6.593	69.1	2.4786	1	273	21.0	9.67	22.4
502	0.04527	0.0	11.93	0	0.573	6.120	76.7	2.2875	1	273	21.0	9.08	20.6
503	0.06076	0.0	11.93	0	0.573	6.976	91.0	2.1675	1	273	21.0	5.64	23.9
504	0.10959	0.0	11.93	0	0.573	6.794	89.3	2.3889	1	273	21.0	6.48	22.0
505	0.04741	0.0	11.93	0	0.573	6.030	80.8	2.5050	1	273	21.0	7.88	11.9

```python
terms = Boston.columns.drop("medv")
terms
```

```
Index(['crim', 'zn', 'indus', 'chas', 'nox', 'rm', 'age', 'dis', 'rad', 'tax',
       'ptratio', 'lstat'],
      dtype='object')
```

```python
X = MS(terms).fit_transform(Boston)
model = sm.OLS(y, X)
results = model.fit()
summarize(results)
```

	coef	std err	t	P>|t|
intercept	41.6173	4.936	8.431	0.000
crim	-0.1214	0.033	-3.678	0.000
zn	0.0470	0.014	3.384	0.001
indus	0.0135	0.062	0.217	0.829
chas	2.8400	0.870	3.264	0.001
nox	-18.7580	3.851	-4.870	0.000
rm	3.6581	0.420	8.705	0.000
age	0.0036	0.013	0.271	0.787
dis	-1.4908	0.202	-7.394	0.000
rad	0.2894	0.067	4.325	0.000
tax	-0.0127	0.004	-3.337	0.001
ptratio	-0.9375	0.132	-7.091	0.000
lstat	-0.5520	0.051	-10.897	0.000

- Age has a high p-value. So how about we drop it from the predictors?

```python
minus_age = Boston.columns.drop(["medv", "age"])
Xma = MS(minus_age).fit_transform(Boston)
model1 = sm.OLS(y, Xma)
summarize(model1.fit())
```

	coef	std err	t	P>|t|
intercept	41.5251	4.920	8.441	0.000
crim	-0.1214	0.033	-3.683	0.000
zn	0.0465	0.014	3.379	0.001

	coef	std err	t	P>|t|
indus	0.0135	0.062	0.217	0.829
chas	2.8528	0.868	3.287	0.001
nox	-18.4851	3.714	-4.978	0.000
rm	3.6811	0.411	8.951	0.000
dis	-1.5068	0.193	-7.825	0.000
rad	0.2879	0.067	4.322	0.000
tax	-0.0127	0.004	-3.333	0.001
ptratio	-0.9346	0.132	-7.099	0.000
lstat	-0.5474	0.048	-11.483	0.000

```
np.unique(Boston["indus"])
```

```
array([ 0.46,  0.74,  1.21,  1.22,  1.25,  1.32,  1.38,  1.47,  1.52,
        1.69,  1.76,  1.89,  1.91,  2.01,  2.02,  2.03,  2.18,  2.24,
        2.25,  2.31,  2.46,  2.68,  2.89,  2.93,  2.95,  2.97,  3.24,
        3.33,  3.37,  3.41,  3.44,  3.64,  3.75,  3.78,  3.97,  4.  ,
        4.05,  4.15,  4.39,  4.49,  4.86,  4.93,  4.95,  5.13,  5.19,
        5.32,  5.64,  5.86,  5.96,  6.06,  6.07,  6.09,  6.2 ,  6.41,
        6.91,  6.96,  7.07,  7.38,  7.87,  8.14,  8.56,  9.69,  9.9 ,
       10.01, 10.59, 10.81, 11.93, 12.83, 13.89, 13.92, 15.04, 18.1 ,
       19.58, 21.89, 25.65, 27.74])
```

Similarly, indus has a high p-value. Let’s drop it as well.

minus_age_indus = Boston.columns.drop([“medv”, “age”, “indus”]) Xmai = MS(minus_age_indus).fit_transfo model1 = sm.OLS(y, Xmai) results1 = model1.fit() summarize(results1)

We can also observe the F-statistic for the regression.

```
(results1.fvalue, results1.f_pvalue)
```

```
(308.9693351215988, 2.9820335524722154e-88)
```

Multivariate Goodness of Fit

We can access the individual components of results by name.

```python
dir(results1)
```

```
['HC0_se',
 'HC1_se',
 'HC2_se',
 'HC3_se',
 '_HCCM',
 '__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__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 '_abat_diagonal',
 '_cache',
 '_data_attr',
 '_data_in_cache',
 '_get_robustcov_results',
 '_get_wald_nonlinear',
 '_is_nested',
```

```
'_transform_predict_exog',
'_use_t',
'_wexog_singular_values',
'aic',
'bic',
'bse',
'centered_tss',
'compare_f_test',
'compare_lm_test',
'compare_lr_test',
'condition_number',
'conf_int',
'conf_int_el',
'cov_HC0',
'cov_HC1',
'cov_HC2',
'cov_HC3',
'cov_kwds',
'cov_params',
'cov_type',
'df_model',
'df_resid',
'diagn',
'eigenvals',
'el_test',
'ess',
'f_pvalue',
'f_test',
'fittedvalues',
'fvalue',
'get_influence',
'get_prediction',
'get_robustcov_results',
'info_criteria',
'initialize',
'k_constant',
'llf',
'load',
'model',
'mse_model',
'mse_resid',
'mse_total',
'nobs',
```

```
 'normalized_cov_params',
 'outlier_test',
 'params',
 'predict',
 'pvalues',
 'remove_data',
 'resid',
 'resid_pearson',
 'rsquared',
 'rsquared_adj',
 'save',
 'scale',
 'ssr',
 'summary',
 'summary2',
 't_test',
 't_test_pairwise',
 'tvalues',
 'uncentered_tss',
 'use_t',
 'wald_test',
 'wald_test_terms',
 'wresid']
```

- results.rsquared gives us the R2 and np.sqrt(results.scale) gives us the RSE.

```
print("RSE", np.sqrt(results1.scale))
```

```
RSE 6.173136281359115
```

```
("R", results1.rsquared)
```

```
('R', 0.5512689379421002)
```

- Variance Inflation Factors are sometimes useful to assess the collinearity effect in our regression model.

Compute VIFs and List Comprehension

```python
vals = [VIF(X, i) for i in range(1, X.shape[1])]
print(vals)
```

```
[1.7674859154310127, 2.2984589077358097, 3.9871806307570994, 1.071167773758404, 4.3690926228
```

```python
vif = pd.DataFrame({"vif": vals}, index=X.columns[1:])
print(vif)
("VIF Range:", np.min(vif), np.max(vif))
```

```
              vif
crim     1.767486
zn       2.298459
indus    3.987181
chas     1.071168
nox      4.369093
rm       1.912532
age      3.088232
dis      3.954037
rad      7.445301
tax      9.002158
ptratio  1.797060
lstat    2.870777
```

```
('VIF Range:', 1.071167773758404, 9.002157663471797)
```

- The VIFs are not very large.

Interaction terms

```python
X = MS(["lstat", "age", ("lstat", "age")]).fit_transform(Boston)
model2 = sm.OLS(y, X)
results2 = model2.fit()
summarize(results2)
```

	coef	std err	t	P>|t|
intercept	36.0885	1.470	24.553	0.000
lstat	-1.3921	0.167	-8.313	0.000

	coef	std err	t	P>|t|
age	-0.0007	0.020	-0.036	0.971
lstat:age	0.0042	0.002	2.244	0.025

```
(results2.rsquared, " > ", results1.rsquared)
```

```
(0.5557265450993936, ' > ', 0.5512689379421002)
```

- The interaction terms lstat:age are not statistically significant at 0.01 level of significance, and R2 does not significantly explain the variation in the model. Suffice to say, the interaction term can be dropped.

Non-linear transformation of the predictors

- The poly() function specifies the first argument term to be added to the model matrix

```
X = MS([poly("lstat", degree=2), "age"]).fit_transform(Boston)
model3 = sm.OLS(y, X)
results3 = model3.fit()
summarize(results3)
```

	coef	std err	t	P>|t|
intercept	17.7151	0.781	22.681	0.0
poly(lstat, degree=2)[0]	-179.2279	6.733	-26.620	0.0
poly(lstat, degree=2)[1]	72.9908	5.482	13.315	0.0
age	0.0703	0.011	6.471	0.0

The effectively 0 p-value associated with the quadratic term suggests an improved model. The R2 confirms it

```
print(results3.rsquared, " > ", results2.rsquared)
```

```
0.6683791720749932  >  0.5557265450993936
```

- By default, poly() creates a basis matrix for inclusion in the model matrix whose columns are orthogonal polynomials which are designed for stable least squares computations. If we had included another argument, raw = True , the basis matrix would consist of lstat and lstat ** 2. Both represent quadratic polynomials. The fitted values would not change. Just the polynomial coefficients. The columns created by poly() do not include an intercept column. These are provided by MS().

Questions:

- What are orthogonal polynomials?
- http://home.iitk.ac.in/~shalab/regression/Chapter12-Regression-PolynomialRegression.pdf
- https://stats.stackexchange.com/questions/258307/raw-or-orthogonal-polynomial-regression

```
X = MS([poly("lstat", degree=2, raw=True), "age"]).fit_transform(Boston)
model3 = sm.OLS(y, X)
results3 = model3.fit()
summarize(results3)
```

	coef	std err	t	P>|t|
intercept	41.2885	0.873	47.284	0.0
poly(lstat, degree=2, raw=True)[0]	-2.6883	0.131	-20.502	0.0
poly(lstat, degree=2, raw=True)[1]	0.0495	0.004	13.315	0.0
age	0.0703	0.011	6.471	0.0

```
print(results3.rsquared, " > ", results1.rsquared)
```

```
0.6683791720749932  >  0.5512689379421002
```

- Use the anova_lm() function to further quantify the superiority of the quadratic fit.

```
anova_lm(results1, results3)
```

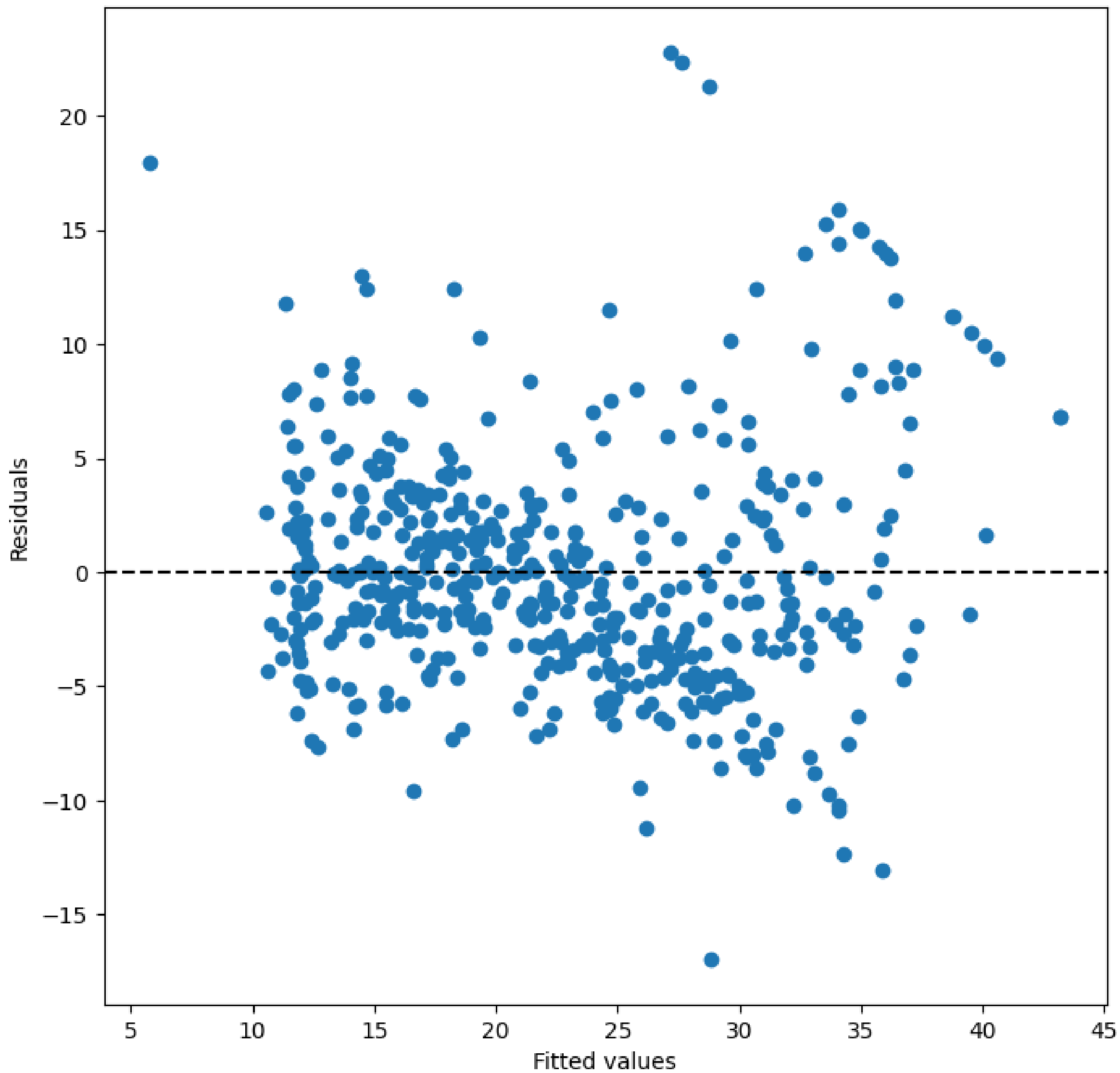
	df_resid	ssr	df_diff	ss_diff	F	Pr(>F)
0	503.0	19168.128609	0.0	NaN	NaN	NaN
1	502.0	14165.613251	1.0	5002.515357	177.278785	7.468491e-35

- results1 corresponds to the linear model containing predictors lstat and age only.
- results3 includes the quadratic term in lstat.
- The anova_lm() function performs a hypothesis test on the two models.
- H0: The quadratic term in the model is not needed.
- Ha: The larger model including the quadratic term is superior.
- Here, the F-statistic is 177.28 and the associated p-value is 0.
- The F-statistic is the t-statistic squared for the quadratic term in results3.

- These nested models differ by 1 degree of freedom.
- This provides very clear evidence that the quadratic term improves the model.
- The anova_lm() function can take more than two models as input.
- The comparison is successive pair-wise.
- That explains the NaNs in the first row of the output above, since there is no previous model with which to compare the output.

We can further plot the residuals of the regression against the fitted values to check of there still is a pattern discernible.

```
_, ax = subplots(figsize=(8, 8))
ax.scatter(results3.fittedvalues, results3.resid)
ax.set_xlabel("Fitted values")
ax.set_ylabel("Residuals")
ax.axhline(0, c="k", ls="--")
```



We can also try and add the interaction term (lstat, age) to the regression and check the results.

```
X = MS([poly("lstat", degree=2, raw=True), "age", ("lstat", "age")]).fit_transform(
    Boston
)
model4 = sm.OLS(y, X)
```

```python
results4 = model4.fit()
summarize(results4)
```

	coef	std err	t	P>|t|
intercept	37.2658	1.250	29.816	0.0
poly(lstat, degree=2, raw=True)[0]	-2.2980	0.156	-14.723	0.0
poly(lstat, degree=2, raw=True)[1]	0.0584	0.004	14.015	0.0
age	0.1439	0.020	7.279	0.0
lstat:age	-0.0079	0.002	-4.424	0.0

```python
print(results4.rsquared, " > ", results3.rsquared)
```

```
0.6808467217930462  >  0.6683791720749932
```

- The R2 in the interaction model again does not exceedingly explain the variance in the model compared to simply having the quadratic term.

Qualitative Predictors

Carseats data

```python
Carseats = load_data("Carseats")
Carseats.columns
```

```
Index(['Sales', 'CompPrice', 'Income', 'Advertising', 'Population', 'Price',
       'ShelveLoc', 'Age', 'Education', 'Urban', 'US'],
      dtype='object')
```

```python
Carseats.shape
```

```
(400, 11)
```

```python
Carseats.describe()
```

	Sales	CompPrice	Income	Advertising	Population	Price	Age	Educati
count	400.000000	400.000000	400.000000	400.000000	400.000000	400.000000	400.000000	400.0000
mean	7.496325	124.975000	68.657500	6.635000	264.840000	115.795000	53.322500	13.90000
std	2.824115	15.334512	27.986037	6.650364	147.376436	23.676664	16.200297	2.620528
min	0.000000	77.000000	21.000000	0.000000	10.000000	24.000000	25.000000	10.00000
25%	5.390000	115.000000	42.750000	0.000000	139.000000	100.000000	39.750000	12.00000
50%	7.490000	125.000000	69.000000	5.000000	272.000000	117.000000	54.500000	14.00000
75%	9.320000	135.000000	91.000000	12.000000	398.500000	131.000000	66.000000	16.00000
max	16.270000	175.000000	120.000000	29.000000	509.000000	191.000000	80.000000	18.00000

- ModelSpec() generates dummy variables for categorical columns automatically. This is termed a one-hot encoding of the categorical feature.
- Their columns sum to one. To avoid collinearity with the intercept, the first column is dropped.

Below we fit a multiple regression model with interaction terms.

```
allvars = list(Carseats.columns.drop("Sales"))
y = Carseats["Sales"]
final = allvars + [("Income", "Advertising"), ("Price", "Age")]
X = MS(final).fit_transform(Carseats)
model = sm.OLS(y, X)
summarize(model.fit())
```

	coef	std err	t	P>|t|
intercept	6.5756	1.009	6.519	0.000
CompPrice	0.0929	0.004	22.567	0.000
Income	0.0109	0.003	4.183	0.000
Advertising	0.0702	0.023	3.107	0.002
Population	0.0002	0.000	0.433	0.665
Price	-0.1008	0.007	-13.549	0.000
ShelveLoc[Good]	4.8487	0.153	31.724	0.000
ShelveLoc[Medium]	1.9533	0.126	15.531	0.000
Age	-0.0579	0.016	-3.633	0.000
Education	-0.0209	0.020	-1.063	0.288
Urban[Yes]	0.1402	0.112	1.247	0.213
US[Yes]	-0.1576	0.149	-1.058	0.291
Income:Advertising	0.0008	0.000	2.698	0.007

	coef	std err	t	P>|t|
Price:Age	0.0001	0.000	0.801	0.424

- It can be seen that ShelvLoc is significant and a good shelving location is associated with high sales (relative to a bad location). Medium has a smaller coefficient than Good leading us to believe that it leads to higher sales than a bad location, but lesser than a good location.

```
allDone()
```

```
<IPython.lib.display.Audio object>
```