lab3

October 26, 2023

0.1 Lab 3: Linear Regression

Jack Krebsbach Math 313

[ ]:
```
import numpy as np
import pandas as pd
from matplotlib.pyplot   import subplots
import statsmodels.api as sm
```

Here we import some submodules from the `statsmodels` packages. We rename the named exports to keep the ***namespace*** clean.

[ ]:
```
from statsmodels.stats.outliers_influence  import  variance_inflation_factor as␣
↪VIF
from statsmodels.stats.anova import anova_lm
```

We also use some functions from ISLP

[ ]:
```
from ISLP import load_data
from ISLP.models import (ModelSpec as MS, summarize , poly)
```

0.1.1 Objects & Namespaces

We can list the objects in the current namespace using `dir()` (the following is truncated).

[ ]:
```
dir()[:5]
```

[ ]:
```
['A', 'Boston', 'Carseats', 'In', 'MS']
```

Each python object has its own namespace. We can also access this namespace of specific objects in the following way. Notice that `numpy` arrays has a sum property!

[ ]:
```
A = np.arange(0,10,2)
dir(A)[:5]
```

[ ]:
```
['T', '__abs__', '__add__', '__and__', '__array__']
```

[ ]:
```
A.sum()
```

0.1.2 Simple Linear Regression

First we will create design/model matrices using `ModelSpec()` transform from `ISLP.models`.

The task will be to build a model using 13 predictors to predict `mev`.

[ ]:
```
Boston = load_data("Boston")
Boston.columns
```

We can learn more about the data by typing `Boston?`. Our first model will just use a single predictor to predict `medv`.

[ ]:
```
Boston?
```

[ ]:
```
# The model matrix is constructed by hand
X = pd.DataFrame({'intercept': np.ones(Boston.shape[0]), 'lstat':␣
↪Boston['lstat']})

X[:4]
```

[ ]:
```
   intercept  lstat
0        1.0   4.98
1        1.0   9.14
2        1.0   4.03
3        1.0   2.94
```

[ ]:
```
# Here we fit the model
y = Boston['medv']
model = sm.OLS(y, X)
results = model.fit()
```

To get more information about the fitted model we can use the `summarize` function. Which gives useful statistics like standard errors, t-statistics, and p-values.

[ ]:
```
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 In practice models usually contain more than one predictor. In addition, transformations of the variables and interactions terms can be added. We can rely on tools from `sklearn` to create these transforms.

[ ]:
```
design = MS(['lstat'])
design = design.fit(Boston)
X = design.transform(Boston)
X[:4]
```

[ ]:
```
   intercept  lstat
0        1.0   4.98
1        1.0   9.14
2        1.0   4.03
3        1.0   2.94
```

In our previous data set `fit()` does not do much, it just checks if the lsat variable exists in the data set.

While this processed was executed in two lines of code, the design object is changed after calling `fit()`.

[ ]:
```
results.summary()
results.params
```

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

We can also use `get_prediction` to obtain new labels with confidence intervals from data not in the training set.

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

[ ]:
```
   intercept  lstat
0        1.0      5
1        1.0     10
2        1.0     15
```

[ ]:
```
# Here we obtain predictions for the new data.
new_predictions = results.get_prediction(newX); new_predictions.predicted_mean
```

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

[ ]:
```
# Extract confidence intervals
new_predictions.conf_int(alpha=0.05)
```

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

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

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

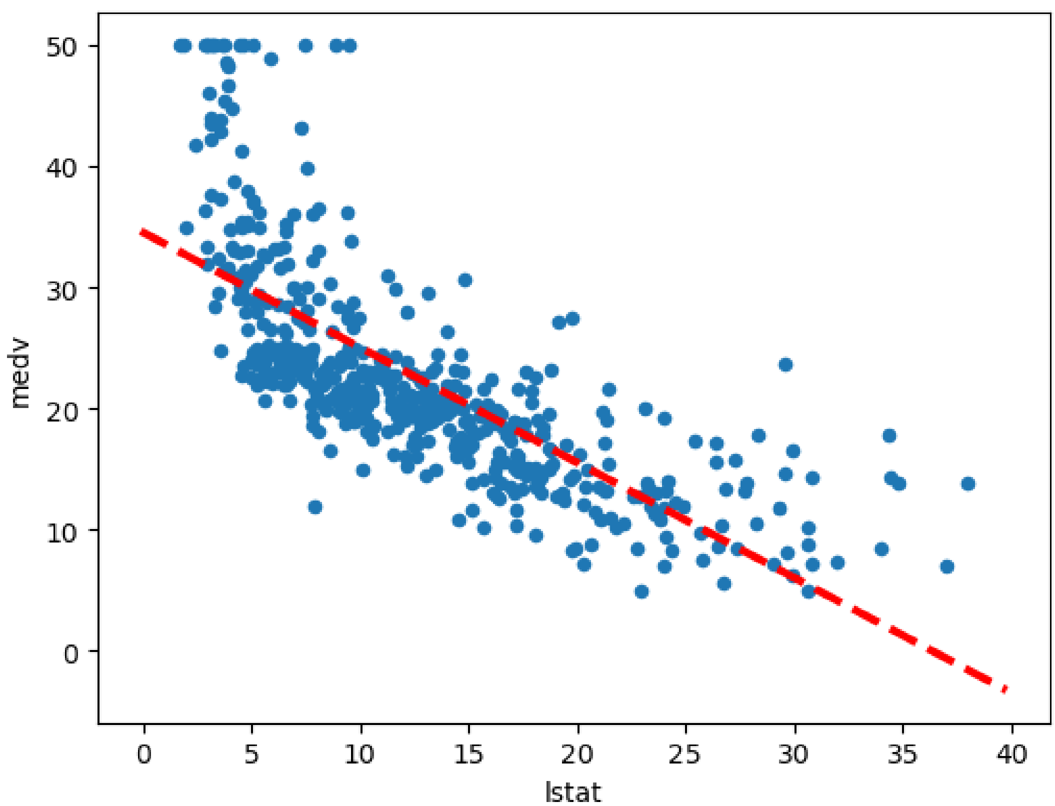
Defining functions Here we define some helper functions to visualize the fits and residuals.

```
[ ]: def abline(ax, b, m):
         "Add a line with slope m and intercept b to ax"
         xlim = ax.get_xlim()
         ylim = [m * xlim[0] + b, m * xlim[1] + b]
         ax.plot(xlim, ylim)
```

By adding in `*args` and `kwargs` we can any number of non-named arguments and any number of named arguments.

```
[ ]: def abline(ax, b, m, *args, **kwargs):
         "Add a line with slope m and intercept b to ax"
         xlim = ax.get_xlim()
         ylim = [m * xlim[0] + b, m * xlim[1] + b]
         ax.plot(xlim, ylim, *args, **kwargs)
```

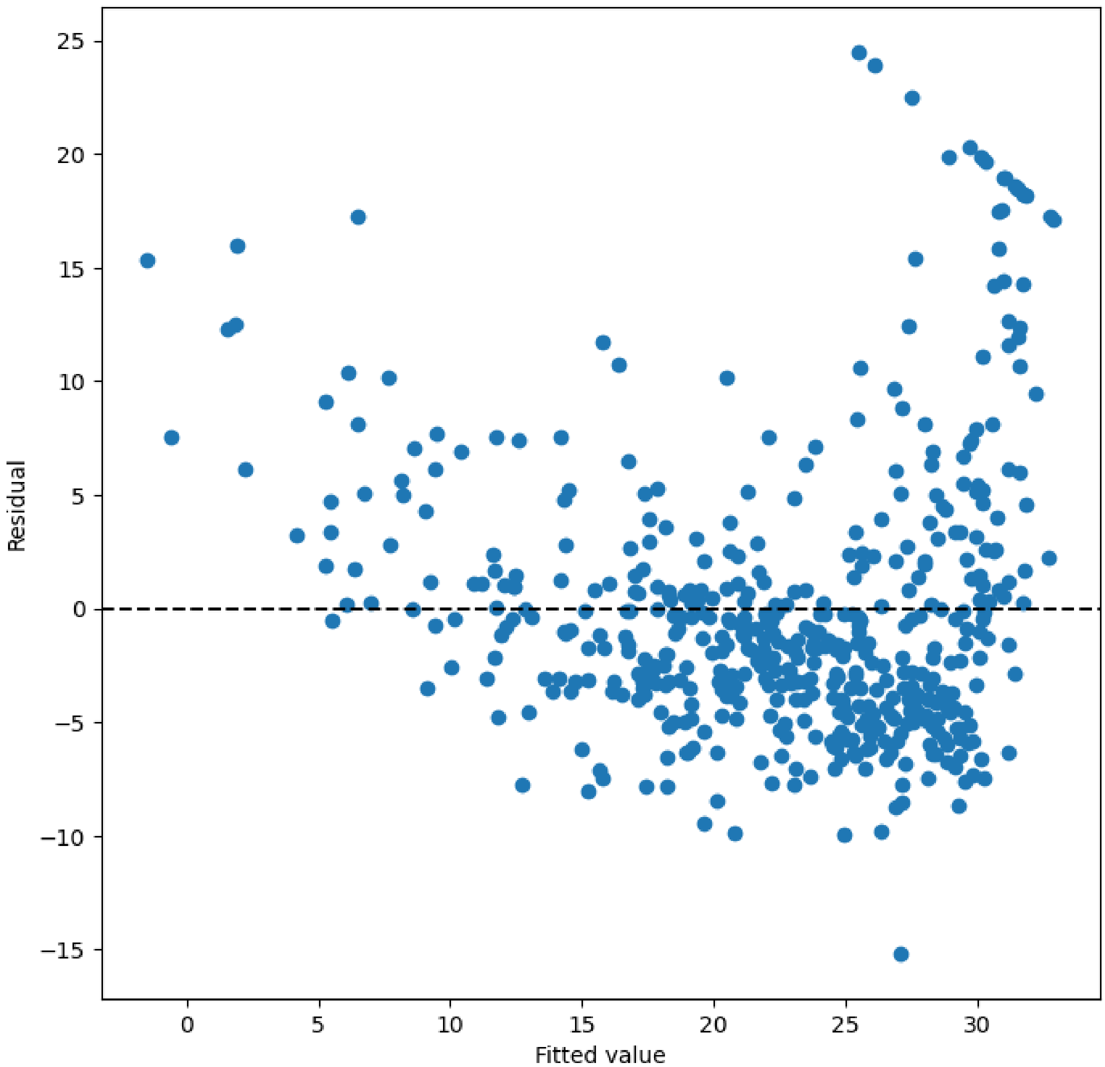
```
[ ]: ax = Boston.plot.scatter('lstat', 'medv')
     abline(ax,
         results.params[0],
         results.params[1],
         'r--', linewidth=3)
```



The figure above shows that there is some evidence of a non-linear relationship between `lsat` and `mev`.

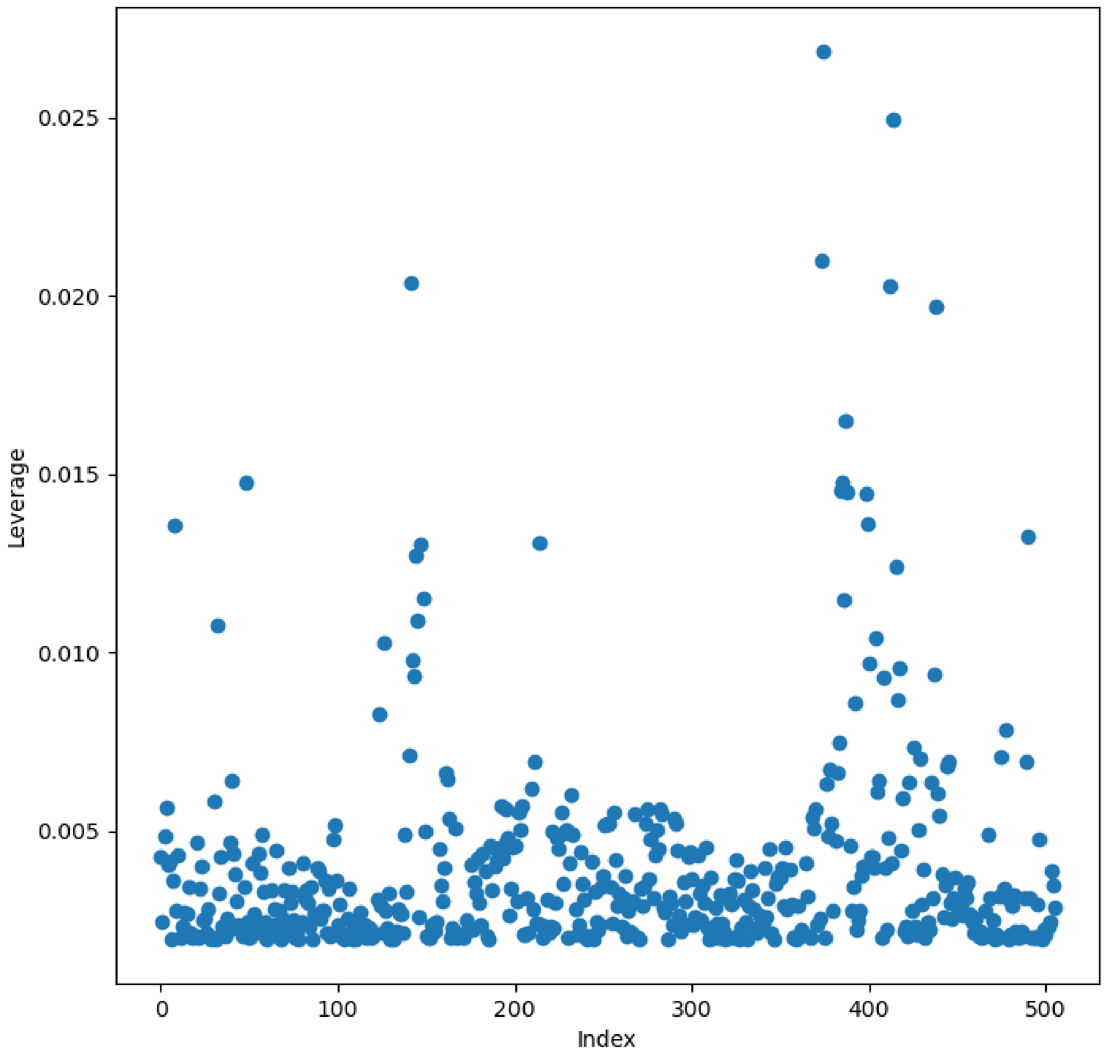
Here we analyze plots that visualize information of the fitted values and residuals. The residual plot also shows evidence of a non-linear relationship. We can also compute leverage statistics by using the `hat_matrix_diag` attribute from the object returned by `get_influence()`.

[ ]:
```
ax = subplots(figsize=(8,8))[1]
ax.scatter(results.fittedvalues , results.resid)
ax.set_xlabel('Fitted value')
ax.set_ylabel('Residual')
ax.axhline(0, c='k', ls='--');
```



[ ]:
```
infl = results.get_influence()
ax = subplots(figsize=(8,8))[1]
ax.scatter(np.arange(X.shape[0]), infl.hat_matrix_diag)
ax.set_xlabel('Index')
ax.set_ylabel('Leverage')
```

[ ]: 374



[ ]:
```
# This identifies the index of the largest element in the array which in turn
↪identifies which observation has the largest leverage statistic.
np.argmax(infl.hat_matrix_diag)
```

[ ]: 374

0.1.3 Multiple Linear Regression

To use multiple linear regression using least squares we can use `ModelSpec()` transform to construct the required model matrix. In the following we add `age` to the predictors.

[ ]:
```
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
```

We can easily create a list of all the predictors by dropping a single one instead of typing them all out.

[ ]:
```
terms = Boston.columns.drop('medv')
terms
```

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

[ ]:
```
# Fit the multiple linear regression model.
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
```

[ ]:
```
# Using all the variables as predictors except age
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
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

Multivariable goodness of fit. The individual components of results can be accessed by name.

To access this information we can use list comprehension which is a simple and powerful way to form a list in python.

[ ]:

```
# Forming these Python objects within the list is called list comprehension.
vals = [VIF(X, i)
    for i in range(1, X.shape[1])]
vif = pd.DataFrame({'vif':vals},
index=X.columns[1:])
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

[ ]:

```
# We can use list comprehension to perform repetitive operations
vals = []
for i in range(1, X.values.shape[1]):
```

```
    vals.append(VIF(X.values, i))
```

Interaction terms To add an interaction terms we can include a tuple in the model matrix.

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

[ ]:

	coef	std err	t	P>|t|
intercept	36.0885	1.470	24.553	0.000
lstat	-1.3921	0.167	-8.313	0.000
age	-0.0007	0.020	-0.036	0.971
lstat:age	0.0042	0.002	2.244	0.025

Non-linear Transformation of the Predictors We can also include non-linear transformations of the predictors in addition to just the interaction terms and features themeselves.

[ ]:
```
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

We see that the p-value associated with the quadratic term is near zero. This means that the model was improved by adding that term.

[ ]:
```
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

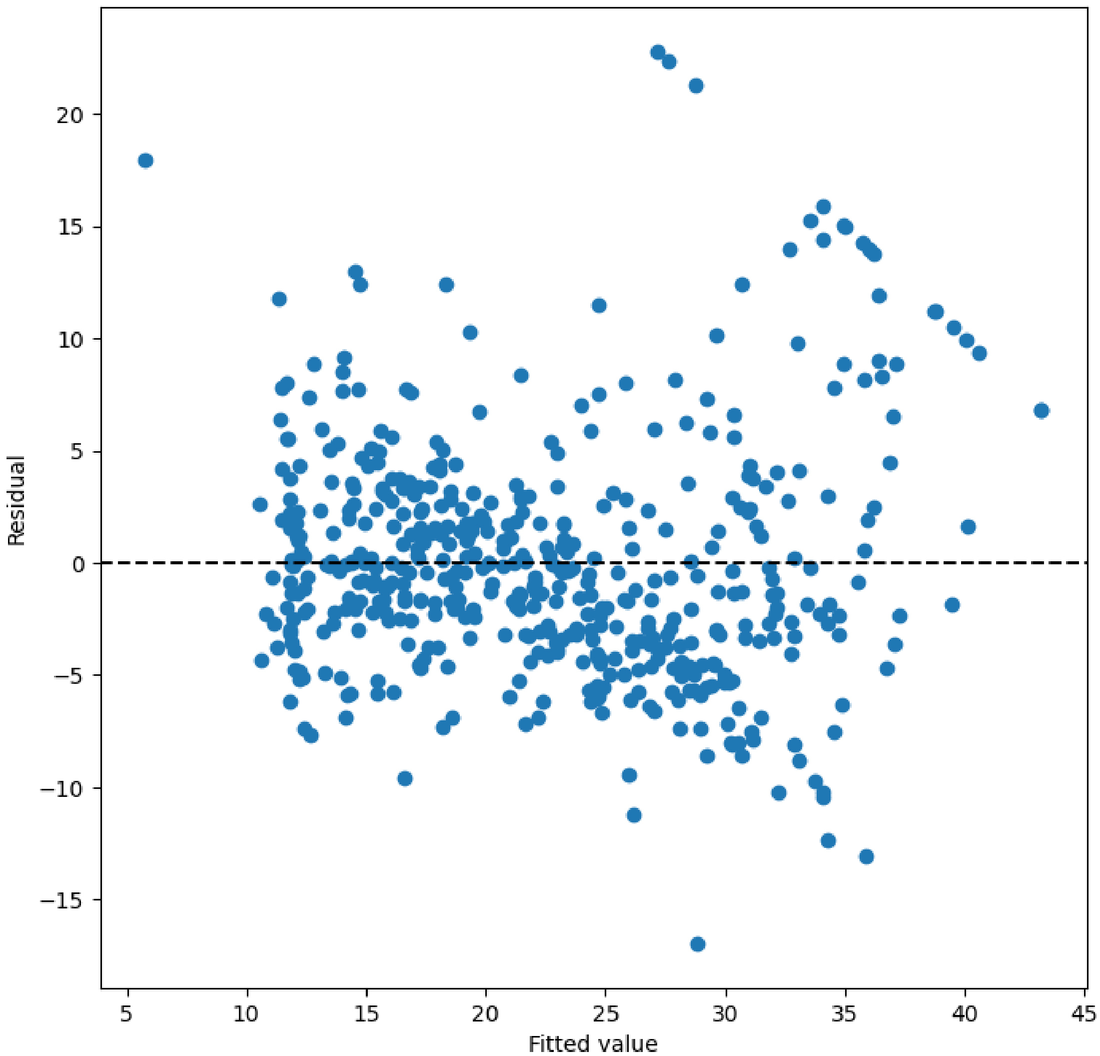
To get a better understanding of how the quadratic fit is superior to the linear fit we can use the `anova_lm` function, which performs a hypothesis test against the two models.

The F statistics is large and its corresponding p-value is near zero, suggesting ample evidence that the bigger model is better.

In the figure below we see that the residuals do not have a discernible structure or pattern. Therefore, we can be confident that our model can appropriately describe the underlying pattern of the data.

[ ]:
```
ax = subplots(figsize=(8,8))[1]
ax.scatter(results3.fittedvalues , results3.resid)
ax.set_xlabel('Fitted value')
ax.set_ylabel('Residual')
ax.axhline(0, c='k', ls='--')
```

[ ]: <matplotlib.lines.Line2D at 0x2835b7950>



Qualitative Predictors Instead of the previous data set we will work with the `Carseats` data which contains qualitative predictors in addition to quantitative predictors. We will work to predict `sales` in 400 locations.

These data are realized as one-hot-encoded variables.

[ ]:
```
Carseats = load_data('Carseats')
Carseats.columns
```

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

To avoid collinearity we drop the first column.

[ ]:
```
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
Price:Age	0.0001	0.000	0.801	0.424

We have added an interaction term between price and age. To encode the `ShelveLoc[Good]` dummy variable we insert a 1 to indicate a positive observation and a 0 otherwise.

We see that `ShelvLoc[Good]` has a positive value, indicating that good shelving location is associated with high sales. It has a higher coefficient than `ShelvLoc[Medium]` which means it leads to higher sales as well.