01_linear_regression_intro

September 29, 2021

0.1 Linear Regression - Introduction

Linear regression relates a continuous response (dependent) variable to one or more predictors (features, independent variables), using the assumption that the relationship is linear in nature: - The relationship between each feature and the response is a straight line when we keep other features constant. - The slope of this line does not depend on the values of the other variables. - The effects of each variable on the response are additive (but we can include new variables that represent the interaction of two variables).

In other words, the model assumes that the response variable can be explained or predicted by a linear combination of the features, except for random deviations from this linear relationship.

0.2 Imports & Settings

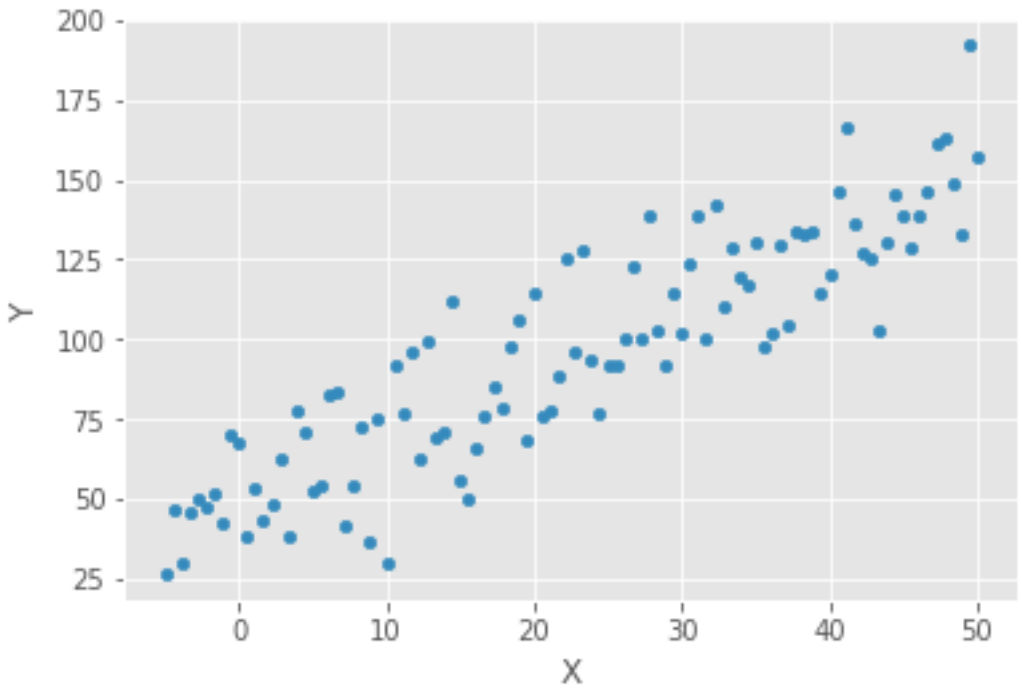
[1]:
```
%matplotlib inline
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import axes3d, Axes3D
import pandas as pd
import numpy as np
import statsmodels.api as sm
from sklearn.linear_model import SGDRegressor
from sklearn.preprocessing import StandardScaler
import warnings
```

[2]:
```
plt.style.use('ggplot')
pd.options.display.float_format = '{:,.2f}'.format
warnings.filterwarnings('ignore')
```

0.2.1 Simple Regression

Generate random data

[3]:
```
x = np.linspace(-5, 50, 100)
y = 50 + 2 * x  + np.random.normal(0, 20, size=len(x))
data = pd.DataFrame({'X': x, 'Y': y})
ax = data.plot.scatter(x='X', y='Y');
```



Our linear model with a single independent variable on the left-hand side assumes the following form:

$$y = \beta_0 + \beta_1 X_1 + \epsilon$$

ϵ accounts for the deviations or errors that we will encounter when our data do not actually fit a straight line. When ϵ materializes, that is when we run the model of this type on actual data, the errors are called **residuals**.

Estimate a simple regression with statsmodels The upper part of the summary displays the dataset characteristics, namely the estimation method, the number of observations and parameters, and indicates that standard error estimates do not account for heteroskedasticity.

The middle panel shows the coefficient values that closely reflect the artificial data generating process. We can confirm that the estimates displayed in the middle of the summary result can be obtained using the OLS formula derived previously:

[4]:
```
X = sm.add_constant(data['X'])
model = sm.OLS(data['Y'], X).fit()
print(model.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                      Y   R-squared:                       0.799
Model:                            OLS   Adj. R-squared:                  0.797
```

```
Method:                 Least Squares   F-statistic:                     388.8
Date:                Thu, 27 Jun 2019   Prob (F-statistic):           6.95e-36
Time:                        18:32:15   Log-Likelihood:                -422.48
No. Observations:                 100   AIC:                             849.0
Df Residuals:                      98   BIC:                             854.2
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const         49.3348      2.879     17.136      0.000      43.622      55.048
X              2.0546      0.104     19.719      0.000       1.848       2.261
==============================================================================
Omnibus:                        0.953   Durbin-Watson:                   2.199
Prob(Omnibus):                  0.621   Jarque-Bera (JB):                1.059
Skew:                           0.186   Prob(JB):                        0.589
Kurtosis:                       2.660   Cond. No.                         47.6
==============================================================================

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

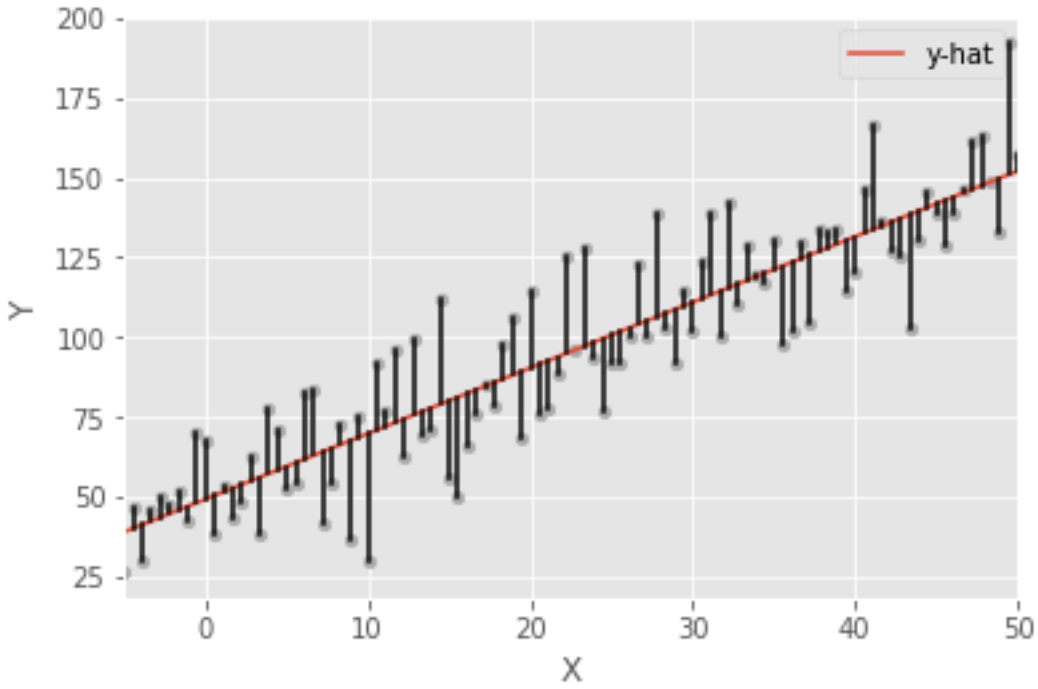
Verify calculation

[5]:
```
beta = np.linalg.inv(X.T.dot(X)).dot(X.T.dot(y))
pd.Series(beta, index=X.columns)
```

[5]:
```
const   49.33
X        2.05
dtype: float64
```

Display model & residuals

[6]:
```
data['y-hat'] = model.predict()
data['residuals'] = model.resid
ax = data.plot.scatter(x='X', y='Y', c='darkgrey')
data.plot.line(x='X', y='y-hat', ax=ax);
for _, row in data.iterrows():
    plt.plot((row.X, row.X), (row.Y, row['y-hat']), 'k-')
```



0.2.2 Multiple Regression

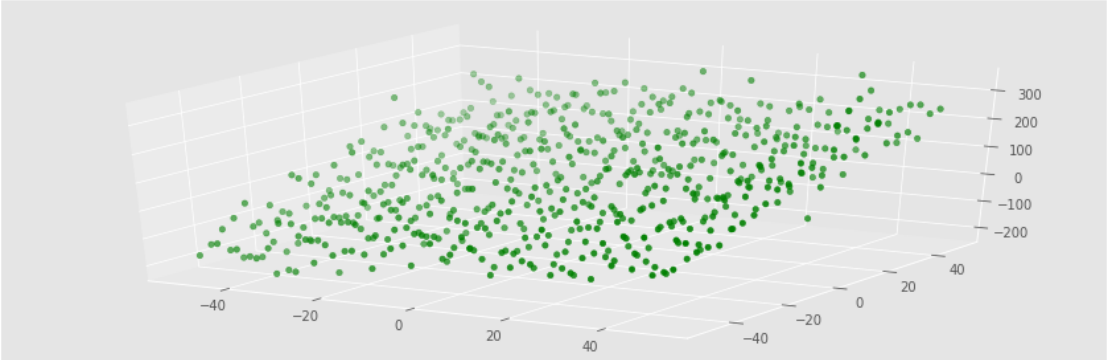
For two independent variables, the model simply changes as follows:

$$y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \epsilon$$

Generate new random data

[7]:
```
## Create data
size = 25
X_1, X_2 = np.meshgrid(np.linspace(-50, 50, size), np.linspace(-50, 50, size),␣
↪indexing='ij')
data = pd.DataFrame({'X_1': X_1.ravel(), 'X_2': X_2.ravel()})
data['Y'] = 50 + data.X_1 + 3 * data.X_2 + np.random.normal(0, 50, size=size**2)

## Plot
three_dee = plt.figure(figsize=(15, 5)).gca(projection='3d')
three_dee.scatter(data.X_1, data.X_2, data.Y, c='g');
```



[8]:
```
X = data[['X_1', 'X_2']]
y = data['Y']
```

Estimate multiple regression model with statsmodels The upper right part of the panel displays the goodness-of-fit measures just discussed, alongside the F-test that rejects the hypothesis that all coefficients are zero and irrelevant. Similarly, the t-statistics indicate that intercept and both slope coefficients are, unsurprisingly, highly significant.

The bottom part of the summary contains the residual diagnostics. The left panel displays skew and kurtosis that are used to test the normality hypothesis. Both the Omnibus and the Jarque—Bera test fails to reject the null hypothesis that the residuals are normally distributed. The Durbin—Watson statistic tests for serial correlation in the residuals and has a value near 2 which, given 2 parameters and 625 observations, fails to reject the hypothesis of no serial correlation.

Lastly, the condition number provides evidence about multicollinearity: it is the ratio of the square roots of the largest and the smallest eigenvalue of the design matrix that contains the input data. A value above 30 suggests that the regression may have significant multicollinearity.

[9]:
```
X_ols = sm.add_constant(X)
model = sm.OLS(y, X_ols).fit()
print(model.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                      Y   R-squared:                       0.800
Model:                            OLS   Adj. R-squared:                  0.799
Method:                 Least Squares   F-statistic:                     1245.
Date:                Thu, 27 Jun 2019   Prob (F-statistic):          3.70e-218
Time:                        18:32:17   Log-Likelihood:                -3304.8
No. Observations:                 625   AIC:                             6616.
Df Residuals:                     622   BIC:                             6629.
Df Model:                           2
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
```

```
------------------------------------------------------------------------------
const         50.8057      1.920     26.463      0.000      47.035      54.576
X_1            0.9997      0.064     15.646      0.000       0.874       1.125
X_2            3.0271      0.064     47.375      0.000       2.902       3.153
==============================================================================
Omnibus:                        0.617   Durbin-Watson:                   2.042
Prob(Omnibus):                  0.735   Jarque-Bera (JB):                0.707
Skew:                           0.008   Prob(JB):                        0.702
Kurtosis:                       2.836   Cond. No.                         30.0
==============================================================================

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

Verify computation

[10]:
```
beta = np.linalg.inv(X_ols.T.dot(X_ols)).dot(X_ols.T.dot(y))
pd.Series(beta, index=X_ols.columns)
```

[10]:
```
const   50.81
X_1      1.00
X_2      3.03
dtype: float64
```

Save output as image

[11]:
```
plt.rc('figure', figsize=(12, 7))
plt.text(0.01, 0.05, str(model.summary()), {'fontsize': 14}, fontproperties =␣
 ↪'monospace')
plt.axis('off')
plt.tight_layout()
plt.subplots_adjust(left=0.2, right=0.8, top=0.8, bottom=0.1)
# plt.savefig('multiple_regression_summary.png', bbox_inches='tight', dpi=300);
```

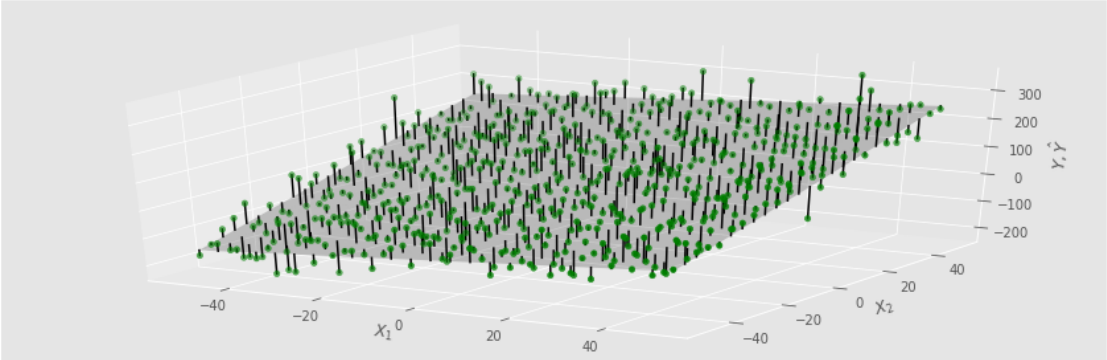
```
                            OLS Regression Results
==============================================================================
Dep. Variable:                      Y   R-squared:                       0.800
Model:                            OLS   Adj. R-squared:                  0.799
Method:                 Least Squares   F-statistic:                     1245.
Date:                Thu, 27 Jun 2019   Prob (F-statistic):          3.70e-218
Time:                        18:32:18   Log-Likelihood:                -3304.8
No. Observations:                 625   AIC:                             6616.
Df Residuals:                     622   BIC:                             6629.
Df Model:                           2
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const         50.8057      1.920     26.463      0.000      47.035      54.576
X_1            0.9997      0.064     15.646      0.000       0.874       1.125
X_2            3.0271      0.064     47.375      0.000       2.902       3.153
==============================================================================
Omnibus:                        0.617   Durbin-Watson:                   2.042
Prob(Omnibus):                  0.735   Jarque-Bera (JB):                0.707
Skew:                           0.008   Prob(JB):                        0.702
Kurtosis:                       2.836   Cond. No.                         30.0
==============================================================================

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

Display model & residuals The following diagram illustrates the hyperplane fitted by the model to the randomly generated data points

[12]:
```
three_dee = plt.figure(figsize=(15, 5)).gca(projection='3d')
three_dee.scatter(data.X_1, data.X_2, data.Y, c='g')
data['y-hat'] = model.predict()
to_plot = data.set_index(['X_1', 'X_2']).unstack().loc[:, 'y-hat']
three_dee.plot_surface(X_1, X_2, to_plot.values, color='black', alpha=0.2,␣
 ↪linewidth=1, antialiased=True)
for _, row in data.iterrows():
    plt.plot((row.X_1, row.X_1), (row.X_2, row.X_2), (row.Y, row['y-hat']),␣
 ↪'k-');
three_dee.set_xlabel('$X_1$');three_dee.set_ylabel('$X_2$');three_dee.
 ↪set_zlabel('$Y, \hat{Y}$')
# plt.savefig('multiple_regression_plot.png', dpi=300);
```

[12]:
```
Text(0.5, 0, '$Y, \\hat{Y}$')
```



Additional diagnostic tests

0.3 Stochastic Gradient Descent Regression

The sklearn library includes an SGDRegressor model in its linear_models module. To learn the parameters for the same model using this method, we need to first standardize the data because the gradient is sensitive to the scale.

0.3.1 Prepare data

The gradient is sensitive to scale and so is SGDRegressor. Use the `StandardScaler` or `scale` to adjust the features.

We use StandardScaler() for this purpose that computes the mean and the standard deviation for each input variable during the fit step, and then subtracts the mean and divides by the standard deviation during the transform step that we can conveniently conduct in a single fit_transform() command:

[13]:
```
scaler = StandardScaler()
X_ = scaler.fit_transform(X)
```

0.3.2 Configure SGDRegressor

Then we instantiate the SGDRegressor using the default values except for a random_state setting to facilitate replication:

[14]:
```
sgd = SGDRegressor(loss='squared_loss', fit_intercept=True,
                   shuffle=True, random_state=42,
                   learning_rate='invscaling',
                   eta0=0.01, power_t=0.25)
```

0.3.3 Fit Model

Now we can fit the sgd model, create the in-sample predictions for both the OLS and the sgd models, and compute the root mean squared error for each:

[15]:
```
# sgd.n_iter = np.ceil(10**6 / len(y))
sgd.fit(X=X_, y=y)
```

[15]:
```
SGDRegressor(alpha=0.0001, average=False, early_stopping=False, epsilon=0.1,
             eta0=0.01, fit_intercept=True, l1_ratio=0.15,
             learning_rate='invscaling', loss='squared_loss', max_iter=1000,
             n_iter_no_change=5, penalty='l2', power_t=0.25, random_state=42,
             shuffle=True, tol=0.001, validation_fraction=0.1, verbose=0,
             warm_start=False)
```

As expected, both models yield the same result. We will now take on a more ambitious project using linear regression to estimate a multi-factor asset pricing model.

[16]:
```
coeffs = (sgd.coef_ * scaler.scale_) + scaler.mean_
pd.Series(coeffs, index=X.columns)
```

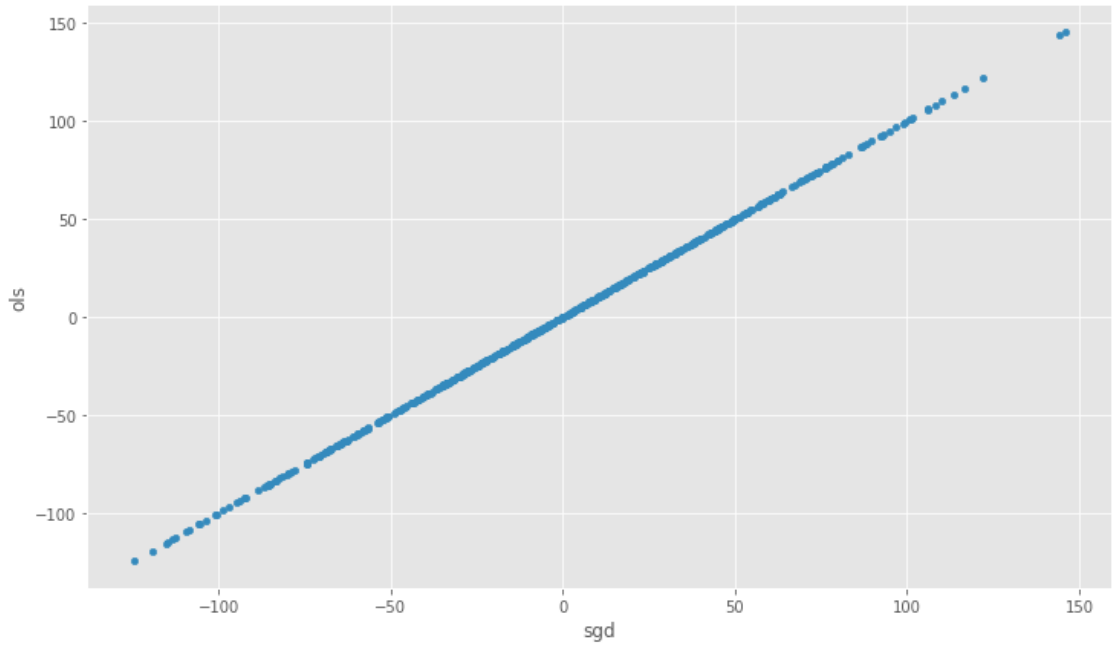
[16]:
```
X_1     902.64
X_2   2,733.89
dtype: float64
```

[17]:
```
resids = pd.DataFrame({'sgd': y - sgd.predict(X_),
                       'ols': y - model.predict(sm.add_constant(X))})
```

[18]:
```
resids.pow(2).sum().div(len(y)).pow(.5)
```

[18]:
```
sgd   47.88
ols   47.88
dtype: float64
```

[19]:
```
resids.plot.scatter(x='sgd', y='ols');
```


150
100
50
0
−50
−100
ols
−100
−50
0
50
100
150
sgd