Linear regression in depth

Fraida Fund

```
from sklearn import metrics
from sklearn.linear_model import LinearRegression


import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
sns.set()

# for 3d interactive plots
from ipywidgets import interact, fixed, widgets
from mpl_toolkits import mplot3d

from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

Data generated by a linear function

Suppose we have a process that generates data as

$$y_i = w_0 + w_1 x_{i,1} + ... + w_d x_{i,d} + \epsilon_i$$

where $\epsilon_i \sim N(0, \sigma^2)$.

Note: in this example, we use a "stochastic error" term. This is not to be confused with a residual term which can include systematic, non-random error.

- stochastic error: difference between observed value and "true" value. These random errors are independent, not systematic, and cannot be "learned" by any machine learning model.
- residual: difference between observed value and estimated value. These errors are typical *not* independent, and they can be systematic.

Here's a function to generate this kind of data

```
def generate_linear_regression_data(n=100, d=1, coef=[5], intercept=1, sigma=0):
  x = np.random.randn(n,d)
  y = (np.dot(x, coef) + intercept).squeeze() + sigma * np.random.randn(n)
  return x, y
```

and some default values we'll use:

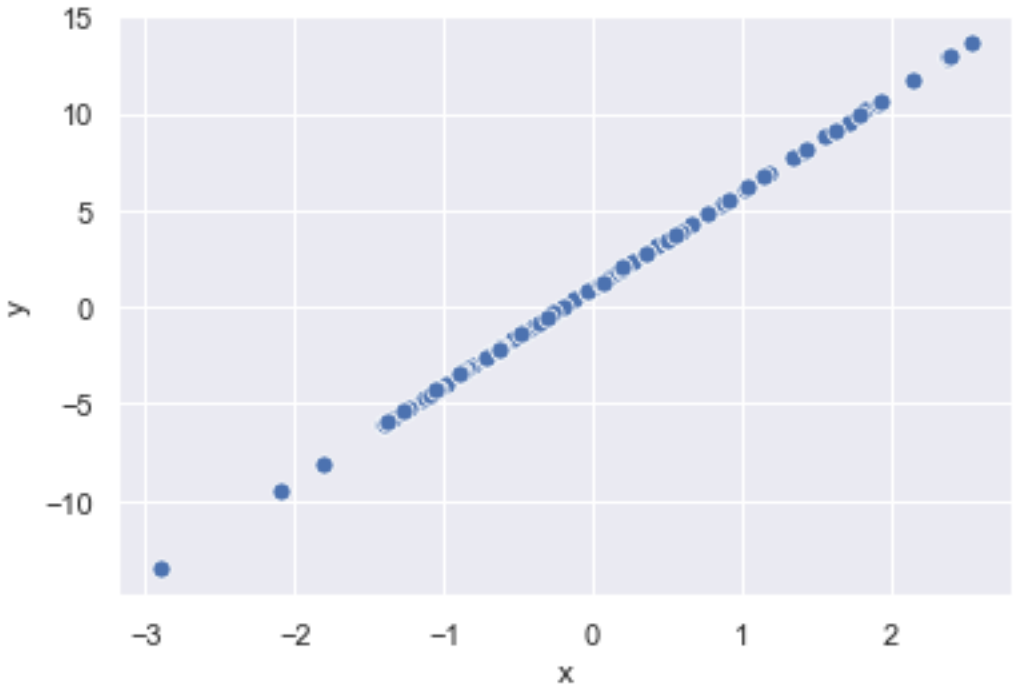
```
n_samples = 100
coef = [5]
intercept = 1
```

Simple linear regression

Generate some data

```
x_train, y_train = generate_linear_regression_data(n=n_samples, d=1, coef=coef,
    intercept=intercept)
x_test,  y_test  = generate_linear_regression_data(n=50, d=1, coef=coef, intercept=intercept)
```

```
sns.scatterplot(x=x_train.squeeze(), y=y_train, s=50);
plt.xlabel('x');
plt.ylabel('y');
```



Note: we generated `x` as a 2D array with `n_samples` rows and 1 column, but to plot it we need a 1D array. In our "crash course" lecture, we introduced the `squeeze()` function that removes any dimension with size 1, so the result here is a 1D array. We could also have used `x_train.reshape(-1,)`.

Fit a linear regression

In the "classical machine learning" part of this course, we'll use `scikit-learn` implementations of most ML models. These all follow the same standard format, so once you learn how to use one, you know the basic usage of all of them.

The basic format is:

```
m = Model()         # create an instance of the model - whatever type it is
m.fit(x_tr, y_tr)   # fit the model using the training data. Note: x_tr must be 2D
                    # if x_tr is 1D, make it 2D by passing x_tr.reshape((-1,1)) or
                        x_tr[:,None]

y_tr_hat = m.predict(x_tr)  # now get model prediction on training data (note: x_tr still
    must be 2D!)
y_ts_hat = m.predict(x_ts)  # also get model prediction on test data (note: x_ts must be 2D!)
```

```
metrics.mean_squared_error(y_ts, y_ts_hat)  # for regression: get error on the test data
metrics.r2_score(y_ts, y_ts_hat)            # or R2 on the test data
m.score(x_ts, y_ts)                         # another way to get test data performance (R2
    for regression)
```

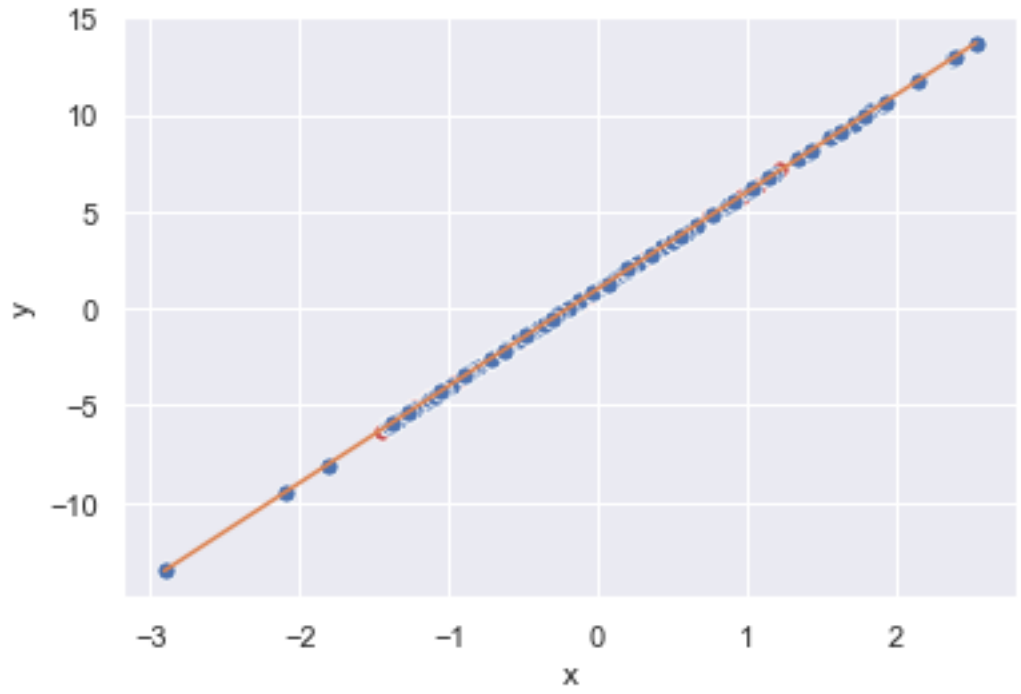
Many models have some additional arguments you can set, or parameters you can check after fitting - check the documentation for details. For example, you can review the LinearRegression model documentation.

Here's how this would apply for a `LinearRegression()` model.

```
reg_simple = LinearRegression().fit(x_train, y_train)
print("Intercept: " , reg_simple.intercept_)
print("Coefficient list: ", reg_simple.coef_)
```

```
Intercept:  0.9999999999999998
Coefficient list:  [5.]
```

```
x_line = [np.min(x_train), np.max(x_train)]
y_line = x_line*reg_simple.coef_ + reg_simple.intercept_
sns.scatterplot(x=x_test.squeeze(), y=y_test, s=50, color=sns.color_palette()[3]);
sns.scatterplot(x=x_train.squeeze(), y=y_train, s=50);
sns.lineplot(x=x_line, y=y_line, color=sns.color_palette()[1]);
plt.xlabel('x');
plt.ylabel('y');
```




```
# Note: other ways to do the same thing...
# first, add a ones column to design matrix
x_tilde = np.hstack((np.ones((n_samples, 1)), x_train))
```

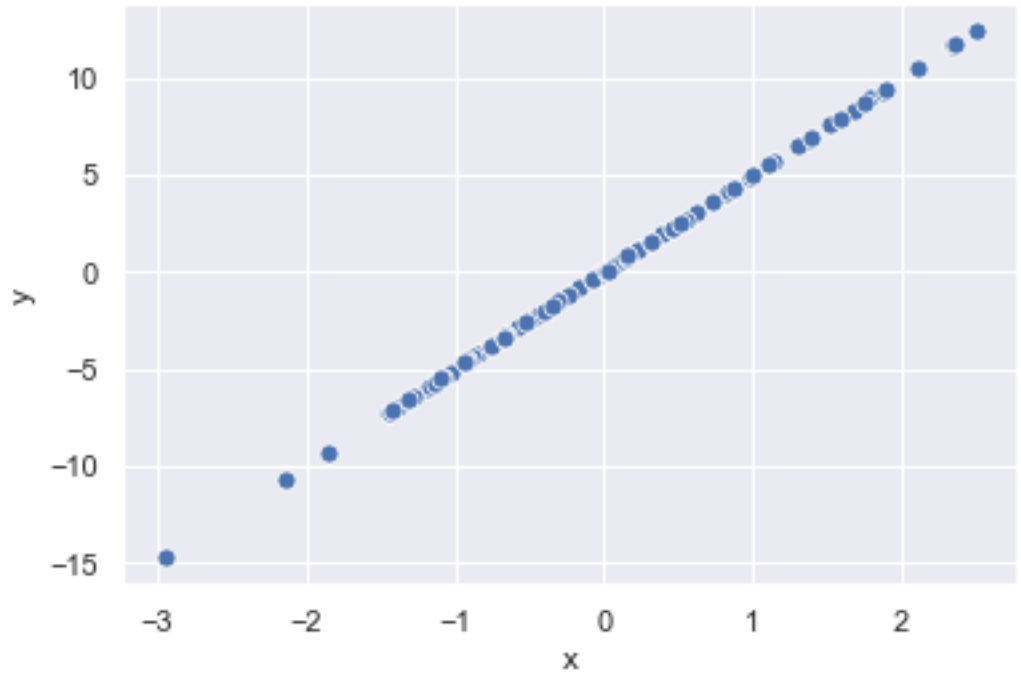
```
# using matrix operations to find w = (X^T X)^{-1} X^T y
print( (np.linalg.inv((x_tilde.T @ x_tilde)) @ x_tilde.T @ y_train) )

# using solve on normal equations: X^T X w = X^T y
# solve only works on matrix that is square and of full-rank
# see https://numpy.org/doc/stable/reference/generated/numpy.linalg.solve.html
print( np.linalg.solve(x_tilde.T @ x_tilde, x_tilde.T @ y_train) )

# using the lstsq solver
# problem may be under-, well-, or over-determined
# see https://numpy.org/doc/stable/reference/generated/numpy.linalg.lstsq.html
print( np.linalg.lstsq(x_tilde,y_train,rcond=0)[0] )
```

```
[1. 5.]
[1. 5.]
[1. 5.]
```

The mean-removed equivalent

Quick digression - what if we don't want to bother with intercept?

```
x_train_mr = x_train - np.mean(x_train)
y_train_mr = y_train - np.mean(y_train)
sns.scatterplot(x=x_train_mr.squeeze(), y=y_train_mr, s=50);
plt.xlabel('x');
plt.ylabel('y');
```



Note that now the data is mean removed - zero mean in every dimension. (Removing the mean is also called *centering* the data.)

This time, the fitted linear regression has 0 intercept:

(We could have specified `fit_intercept=False` as an argument to the model, but we didn't so that we could see for ourselves that the intercept is zero!)

```
reg_mr = LinearRegression().fit(x_train_mr, y_train_mr)
print("Intercept: " , reg_mr.intercept_)
print("Coefficient list: ", reg_mr.coef_)
```

```
Intercept:  -2.442490654175354e-17
Coefficient list:  [5.]
```

Important: when pre-processing data (for example, scaling, or removing the mean), we will always use the training data *only* to get the pre-processing parameters. For example, to get the mean-removed test data we would use

```
x_test_mr = x_test - np.mean(x_train)
y_test_mr = y_test - np.mean(y_train)
```

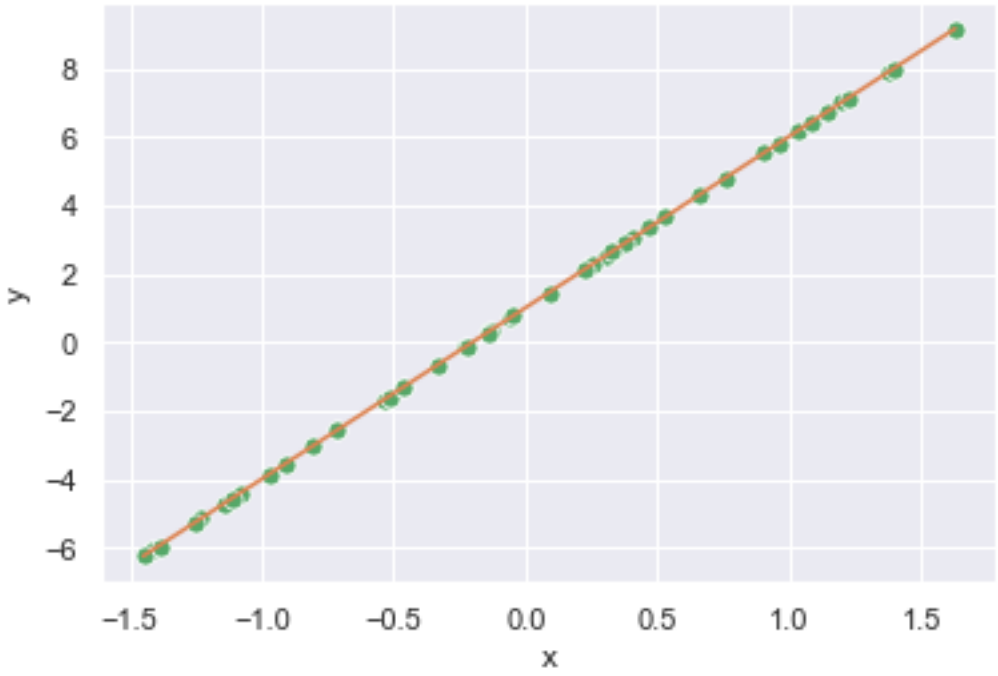
Predict some new points

OK, now we can predict some new points:

```
y_test_hat = reg_simple.predict(x_test)
```

```
x_line = [np.min(x_test), np.max(x_test)]
y_line = x_line*reg_simple.coef_ + reg_simple.intercept_
```

```
sns.lineplot(x=x_line, y=y_line, color=sns.color_palette()[1]);
sns.scatterplot(x=x_test.squeeze(), y=y_test_hat, s=50, color=sns.color_palette()[2]);
plt.xlabel('x');
plt.ylabel('y');
```



Compute MSE

To evaluate the model, we will compute the MSE on the test data (*not* the data used to find the parameters).

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - (w_0 + w_1 x_i))^2$$

Use $\hat{y}_i = w_0 + w_1 x_i$, then

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Here's the `numpy` way:

```
y_test_hat = reg_simple.intercept_ + np.dot(x_test,reg_simple.coef_)
mse_simple = 1.0/(len(y_test)) * np.sum((y_test - y_test_hat)**2)
mse_simple
```

```
1.4546348573242577e-29
```

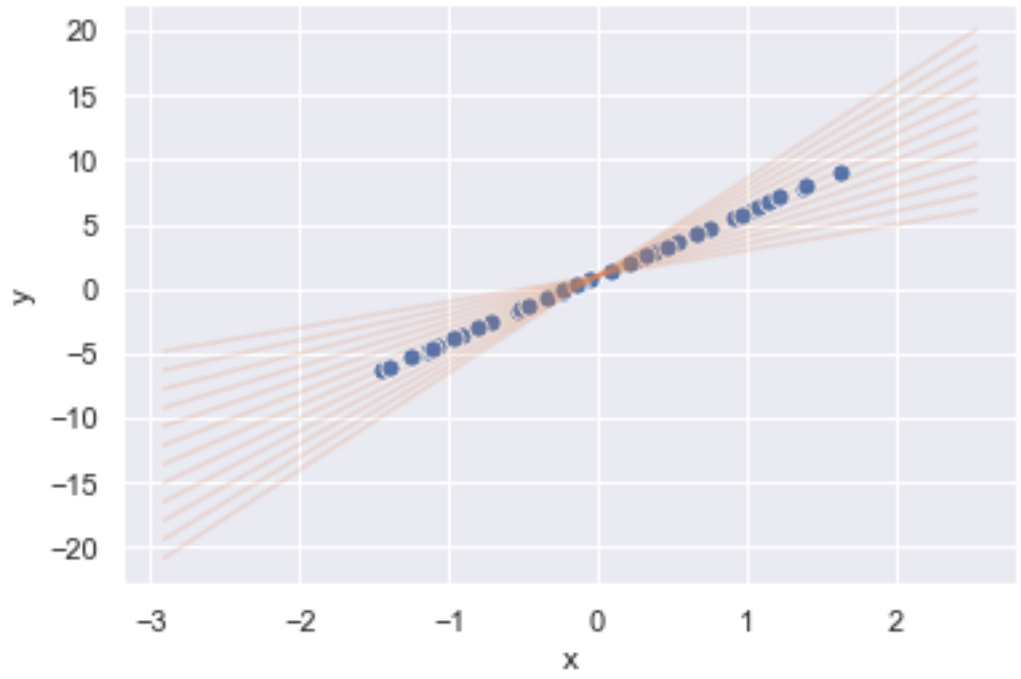
Here's the `scikit-learn` way:

```
# another way to do the same thing using sklearn
y_test_hat = reg_simple.predict(x_test)
metrics.mean_squared_error(y_test, y_test_hat)
```

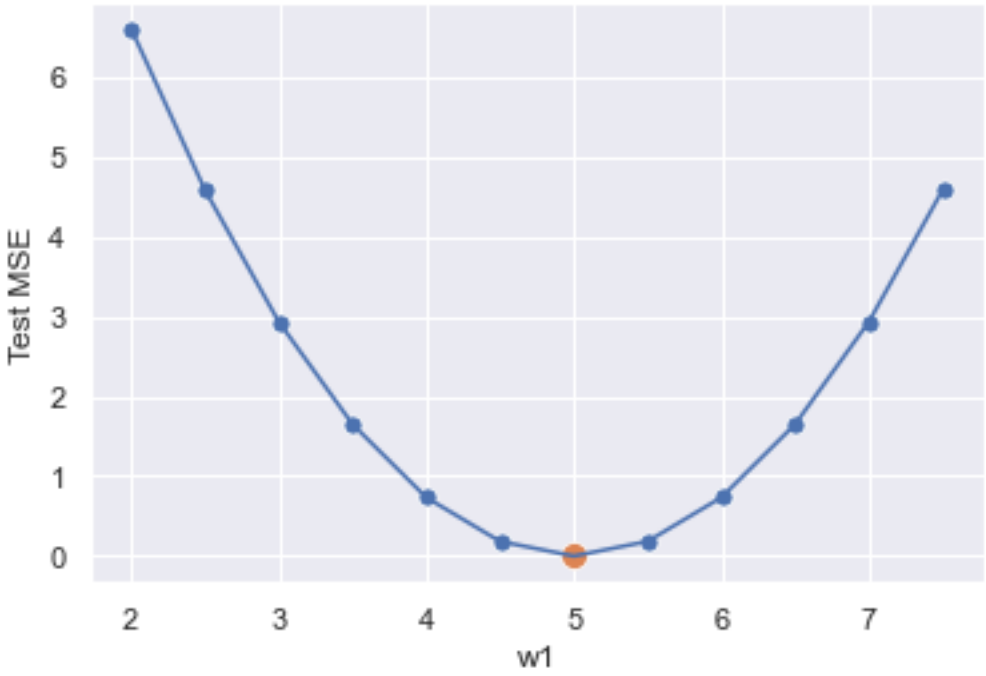
```
1.4546348573242577e-29
```

Visualize MSE for different coefficients

```
coefs = np.arange(2, 8, 0.5)

x_line_c = np.array([np.min(x_train), np.max(x_train)])
y_line_c = coefs.reshape(-1,1)*x_line_c.reshape(1,-1) + reg_simple.intercept_

p = sns.scatterplot(x=x_test.squeeze(), y=y_test_hat, s=50);
p = plt.xlabel('x')
p = plt.ylabel('y')
for idx, c in enumerate(coefs):
  p = sns.lineplot(x=x_line_c, y=y_line_c[idx], color=sns.color_palette()[1], alpha=0.2);
```




```
y_test_hat_c = coefs.reshape(-1,1)*x_test.reshape(1,-1) + reg_simple.intercept_
mses_c = 1.0/(len(y_test)) * np.sum((y_test - y_test_hat_c)**2, axis=1)

sns.lineplot(x=coefs, y=mses_c);
sns.scatterplot(x=coefs, y=mses_c, s=50);
sns.scatterplot(x=reg_simple.coef_, y=mse_simple, color=sns.color_palette()[1], s=100);
p = plt.xlabel('w1');
p = plt.ylabel('Test MSE');
```



Variance, explained variance, R2

Quick reminder:

Mean of x and y:

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i, \quad \bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$$

Sample variance of x and y:

$$\sigma_x^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2, \quad \sigma_y^2 = \frac{1}{n}\sum_{i=1}^{n}(y_i - \bar{y})^2$$

Sample covariance of x and y:

$$\sigma_{xy} = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})$$

```
var_y = 1.0/len(y_test) * np.sum((y_test - np.mean(y_test))**2) # or use np.var()
var_y
```

```
18.31279988426537
```

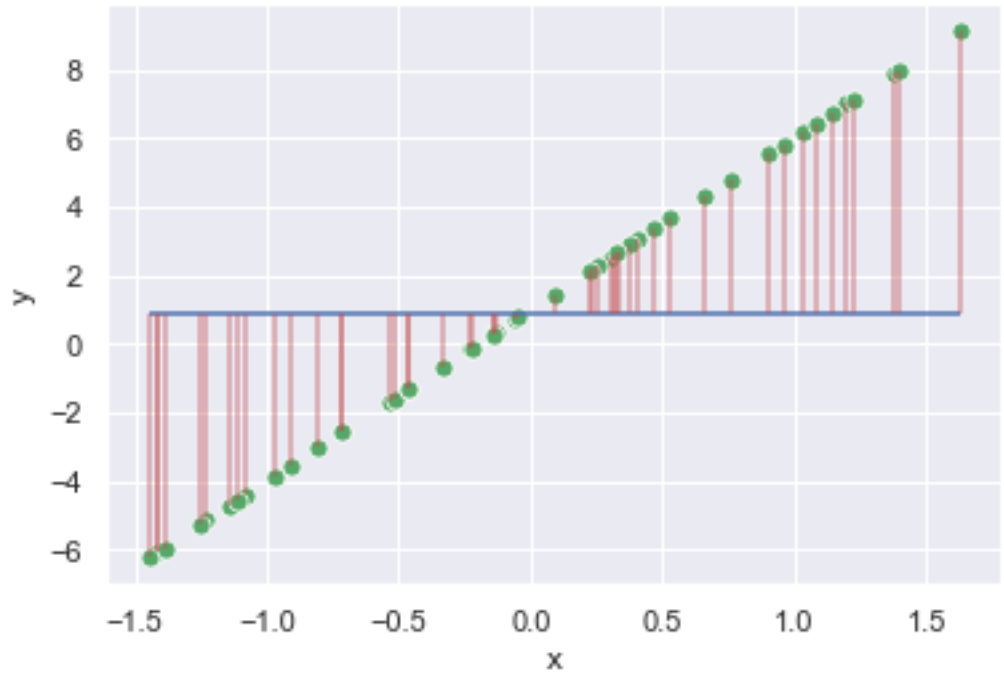
```
mean_y = np.mean(y_test)
mean_y
```

```
0.8372053064991384
```

The variance of y is the mean sum of the squares of the distances from each y_i to $\bar{y}$. These distances are illustrated here:

- the horizontal line shows $\bar{y}$
- each vertical line is a distance from a y_i to $\bar{y}$

```
plt.hlines(y=mean_y, xmin=np.min(x_test), xmax=np.max(x_test));
plt.vlines(x_test, ymin=mean_y, ymax=y_test, alpha=0.5, color=sns.color_palette()[3]);
sns.scatterplot(x=x_test.squeeze(), y=y_test, color=sns.color_palette()[2], s=50);
plt.xlabel('x');
plt.ylabel('y');
```

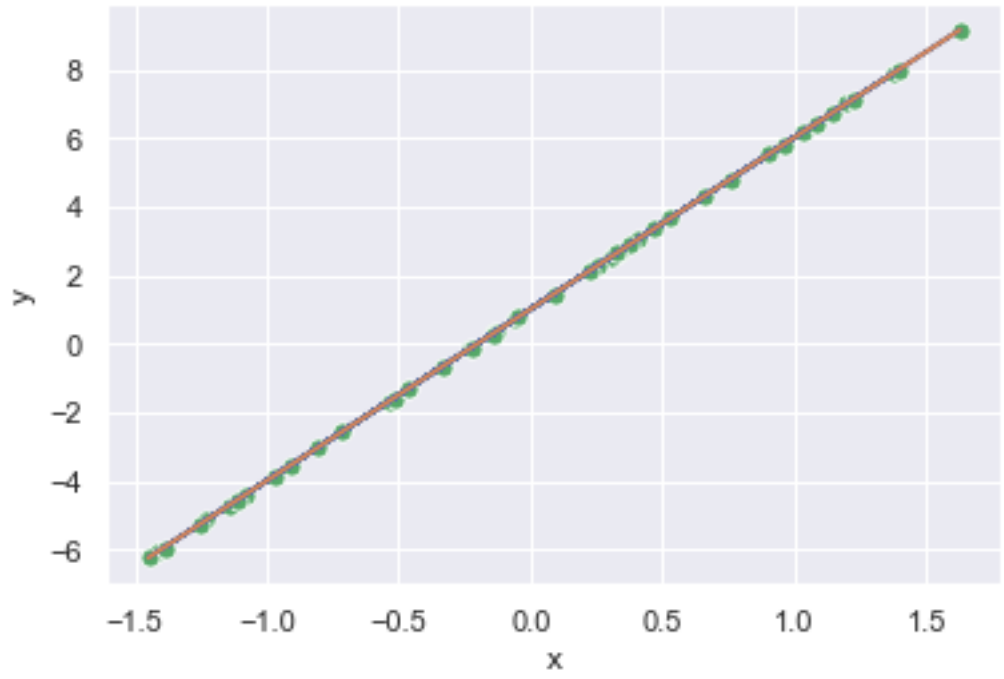


Now let's look at a similar kind of plot, but with distances to the regression line instead of the to mean line:

- In the previous plot, each vertical line was a $y_i - \bar{y}$
- In the following plot, each vertical line is a $y_i - \hat{y}_i$

(where $\hat{y}_i$ is the prediction of the linear regression for a given sample i)

```
plt.plot(x_test, y_test_hat);
plt.vlines(x_test, ymin=y_test, ymax=y_test_hat, color=sns.color_palette()[3], alpha=0.5);
sns.scatterplot(x=x_test.squeeze(), y=y_test, color=sns.color_palette()[2], s=50);
x_line = [np.min(x_test), np.max(x_test)]
y_line = x_line*reg_simple.coef_ + reg_simple.intercept_
sns.lineplot(x=x_line, y=y_line, color=sns.color_palette()[1]);
plt.xlabel('x');
plt.ylabel('y');
```



These two plots together show how well the variance of y is "explained" by the linear regression model:

- The total variance of y is shown in the first plot, where each vertical line is

$$y_i - \bar{y}$$

- The *unexplained* variance of y is shown in the second plot, where each vertical line is the error of the model,

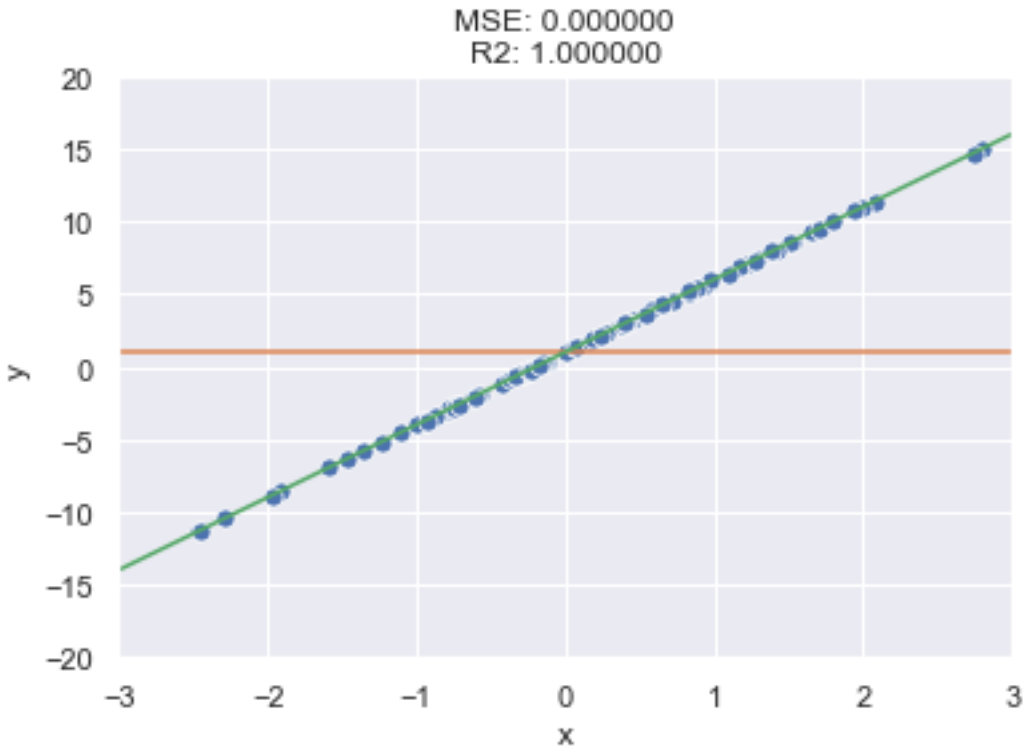
$$y_i - \hat{y}_i$$

In this example, *all* of the variance of y is "explained" by the linear regression.

MSE for this example is 0, R2 is 1.

```
@interact(intercept_fit = widgets.FloatSlider(min=-8, max=8, step=0.5, value=1),
   coef_fit = widgets.FloatSlider(min=-8, max=8, step=0.1, value=5),
   show_residual=True)
def plot_reg(intercept_fit, coef_fit, show_residual):
    x_train, y_train = generate_linear_regression_data(n=20000, d=1, coef=5, intercept=1,
        sigma=0)
    x_test,  y_test =  generate_linear_regression_data(n=10000, d=1, coef=5, intercept=1,
        sigma=0)
    y_test_hat = intercept_fit + coef_fit*x_test
    r2_test = metrics.r2_score(y_test, y_test_hat)
    mse_test = metrics.mean_squared_error(y_test, y_test_hat)
    x_line = np.array([-3, 3])
    y_line = intercept_fit + coef_fit*x_line
    plt.axhline(y=np.mean(y_train), color=sns.color_palette()[1]);
    if show_residual:
      plt.vlines(x_test[:100,], ymin=y_test[:100,], ymax=y_test_hat[:100,], alpha=0.5,
          color=sns.color_palette()[3]);
```

```
    sns.lineplot(x=x_line, y=y_line, color=sns.color_palette()[2]);
    sns.scatterplot(x=x_test[:100,].squeeze(), y=y_test[:100], s=50);
    plt.xlabel('x');
    plt.ylabel('y');
    plt.ylim(-20,20)
    plt.xlim(-3,3)
    plt.title("MSE: %f\nR2: %f" % (mse_test, r2_test) )
```

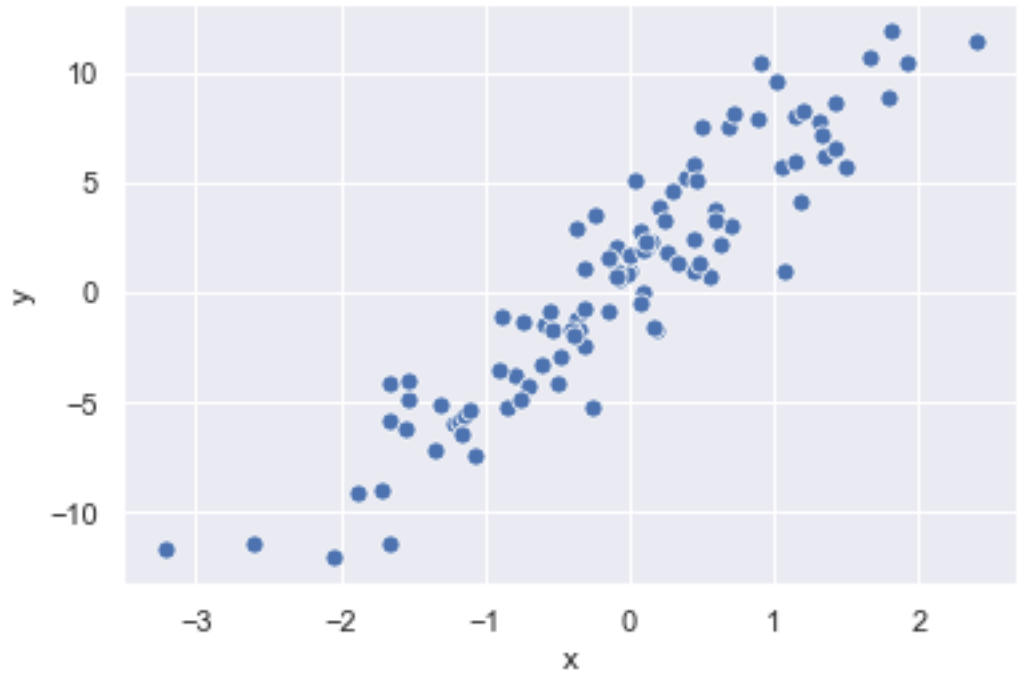


Simple linear regression with noise

Generate some data

```
x_train, y_train = generate_linear_regression_data(n=n_samples, d=1, coef=coef,
    intercept=intercept, sigma=2)
x_test,  y_test =  generate_linear_regression_data(n=50, d=1, coef=coef,
    intercept=intercept, sigma=2)
```

```
sns.scatterplot(x=x_train.squeeze(), y=y_train, s=50);
plt.xlabel('x');
plt.ylabel('y');
```



Fit a linear regression

```
reg_noisy = LinearRegression().fit(x_train, y_train)
print("Coefficient list: ", reg_noisy.coef_)
print("Intercept: " , reg_noisy.intercept_)
```

```
Coefficient list:  [5.00008325]
Intercept:  0.9655615667684232
```

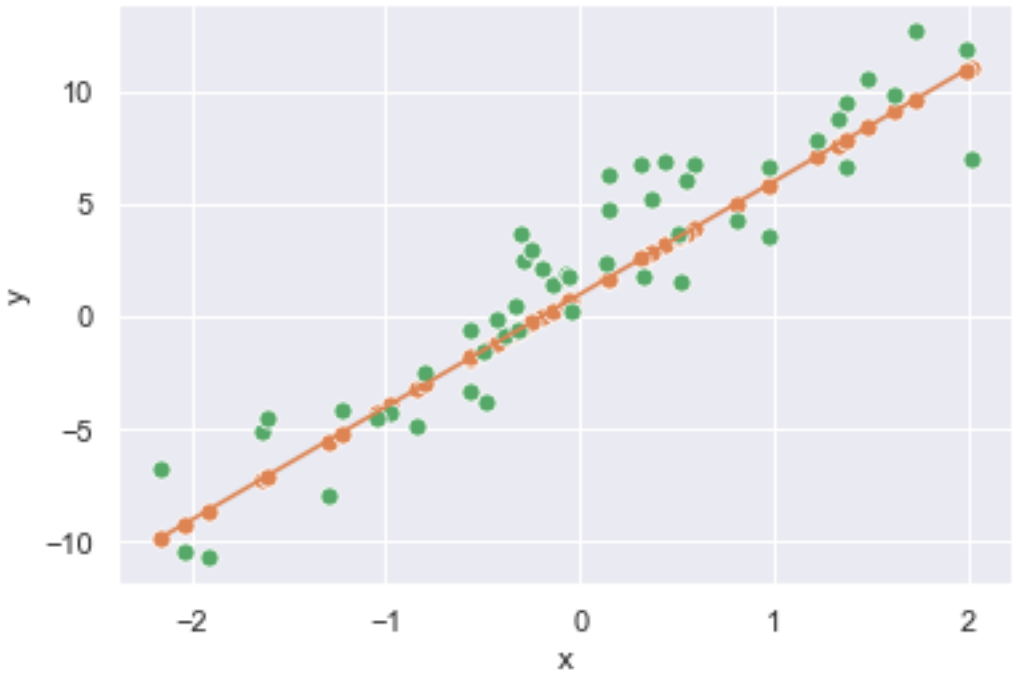
```
x_line = [np.min(x_train), np.max(x_train)]
y_line = x_line*reg_noisy.coef_ + reg_noisy.intercept_

sns.scatterplot(x=x_train.squeeze(), y=y_train, s=50);
sns.lineplot(x=x_line, y=y_line, color=sns.color_palette()[1]);
plt.xlabel('x');
plt.ylabel('y');
```

Predict some new points

```
y_test_hat = reg_noisy.intercept_ + np.dot(x_test,reg_noisy.coef_)
```

```
x_line = [np.min(x_test), np.max(x_test)]
y_line = x_line*reg_noisy.coef_ + reg_noisy.intercept_
```

```
sns.lineplot(x=x_line, y=y_line, color=sns.color_palette()[1]);
sns.scatterplot(x=x_test.squeeze(), y=y_test_hat, color=sns.color_palette()[1], s=50);
sns.scatterplot(x=x_test.squeeze(), y=y_test, color=sns.color_palette()[2], s=50);
plt.xlabel('x');
plt.ylabel('y');
```



Compute MSE

```
y_test_hat = reg_noisy.intercept_ + np.dot(x_test,reg_noisy.coef_)
mse_noisy = 1.0/(len(y_test)) * np.sum((y_test - y_test_hat)**2)
mse_noisy
```

```
4.502030445724956
```

The MSE is higher than before! (When it was essentially zero.)

Does this mean our estimate of w_0 and w_1 is not optimal?

Since we generated the data, we know the "true" coefficient value and we can see how much the MSE would be with the true coefficient values.

```
y_test_perfect_coef = intercept + np.dot(x_test,coef)

mse_perfect_coef = 1.0/(len(y_test_perfect_coef)) * np.sum((y_test_perfect_coef - y_test)**2)
mse_perfect_coef
```

```
4.446303652118996
```

Sometimes our linear regression doesn't select the "true" coefficients?

```
y_train_hat = reg_noisy.intercept_ + np.dot(x_train,reg_noisy.coef_)
mse_train_est = 1.0/(len(y_train)) * np.sum((y_train - y_train_hat)**2)
mse_train_est
```

```
4.207977617008657
```

```
y_train_perfect_coef = intercept + np.dot(x_train,coef)
mse_train_perfect = 1.0/(len(y_train_perfect_coef)) * np.sum((y_train_perfect_coef -
    y_train)**2)
mse_train_perfect
```

```
4.209163964529751
```

The "correct" coefficients had slightly higher MSE on the training set than the fitted coefficients. We fit parameters so that they are optimal on the *training* set, then we use the test set to understand how the model will generalize to new, unseen data.

We saw that part of the MSE is due to noise in the data, and part is due to error in the parameter estimates.

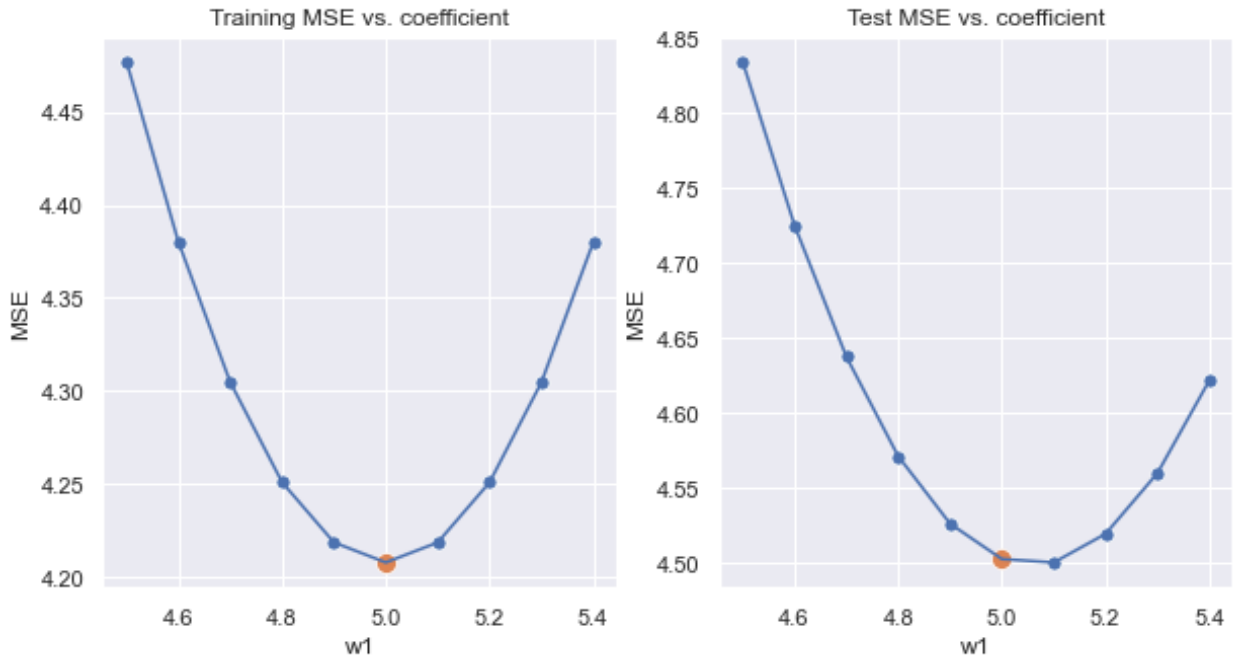
Soon - we will formalize this discussion of different sources of error:

- Error in parameter estimates
- "Noise" - any variation in data that is not a function of the X that we use as input to the model
- Other error - for example, model (hypothesis class) not a good choice for the data

Visualize MSE for different coefficients

```
coefs = np.arange(4.5, 5.5, 0.1)

y_test_hat_c = reg_noisy.intercept_ + np.dot(x_test,coefs.reshape(1,-1))
mses_test =  np.mean((y_test.reshape(-1,1) - y_test_hat_c)**2, axis=0)
y_train_hat_c = reg_noisy.intercept_ + np.dot(x_train,coefs.reshape(1,-1))
mses_train =  np.mean((y_train.reshape(-1,1) - y_train_hat_c)**2, axis=0)
```

```
plt.figure(figsize=(10,5))

plt.subplot(1,2,1)
sns.lineplot(x=coefs, y=mses_train)
sns.scatterplot(x=coefs, y=mses_train, s=50);
sns.scatterplot(x=reg_noisy.coef_, y=mse_train_est, color=sns.color_palette()[1], s=100);
plt.title("Training MSE vs. coefficient");
plt.xlabel('w1');
plt.ylabel('MSE');

plt.subplot(1,2,2)
sns.lineplot(x=coefs, y=mses_test)
sns.scatterplot(x=coefs, y=mses_test, s=50);
sns.scatterplot(x=reg_noisy.coef_, y=mse_noisy, color=sns.color_palette()[1], s=100);
plt.title("Test MSE vs. coefficient");
plt.xlabel('w1');
plt.ylabel('MSE');
```



In the plot on the left (for training MSE), the orange dot (our coefficient estimate) should always have minimum MSE, because we select parameters to minimize MSE on the training set.

In the plot on the right (for test MSE), the orange dot might not have the minimum MSE, because the best coefficient on the training set might not be the best coefficient on the test set. This gives us some idea of how our model will generalize to new, unseen data. We may suspect that if the coefficient estimate is not perfect for *this* test data, it might have some error on other new, unseen data, too.

If you re-run this notebook many times, you'll get a new random sample of training and test data each time. Sometimes, the "true" coefficients may have smaller MSE on the test set than the estimated coefficients. On other runs, the estimated coefficients might have smaller MSE on the test set.

Variance, explained variance, R2

```
var_y = 1.0/len(y_test) * np.sum((y_test - np.mean(y_test))**2)
var_y
```

```
31.10681211791439
```

```
mean_y = np.mean(y_test)
mean_y
```

```
1.8354938932097873
```

```
x_line = [np.min(x_test), np.max(x_test)]
y_line = x_line*reg_noisy.coef_ + reg_noisy.intercept_
plt.hlines(mean_y, xmin=np.min(x_test), xmax=np.max(x_test));
plt.vlines(x_test, ymin=mean_y, ymax=y_test, color=sns.color_palette()[4], alpha=0.5);
sns.scatterplot(x=x_test.squeeze(), y=y_test, color=sns.color_palette()[2], s=50);
plt.xlabel('x');
plt.ylabel('y');
```


```
plt.vlines(x_test, ymin=y_test, ymax=y_test_hat, color=sns.color_palette()[1], alpha=0.5);
sns.scatterplot(x=x_test.squeeze(), y=y_test, color=sns.color_palette()[2], s=50);
x_line = [np.min(x_test), np.max(x_test)]
y_line = x_line*reg_noisy.coef_ + reg_noisy.intercept_
sns.lineplot(x=x_line, y=y_line, color=sns.color_palette()[1]);
plt.xlabel('x');
plt.ylabel('y');
```

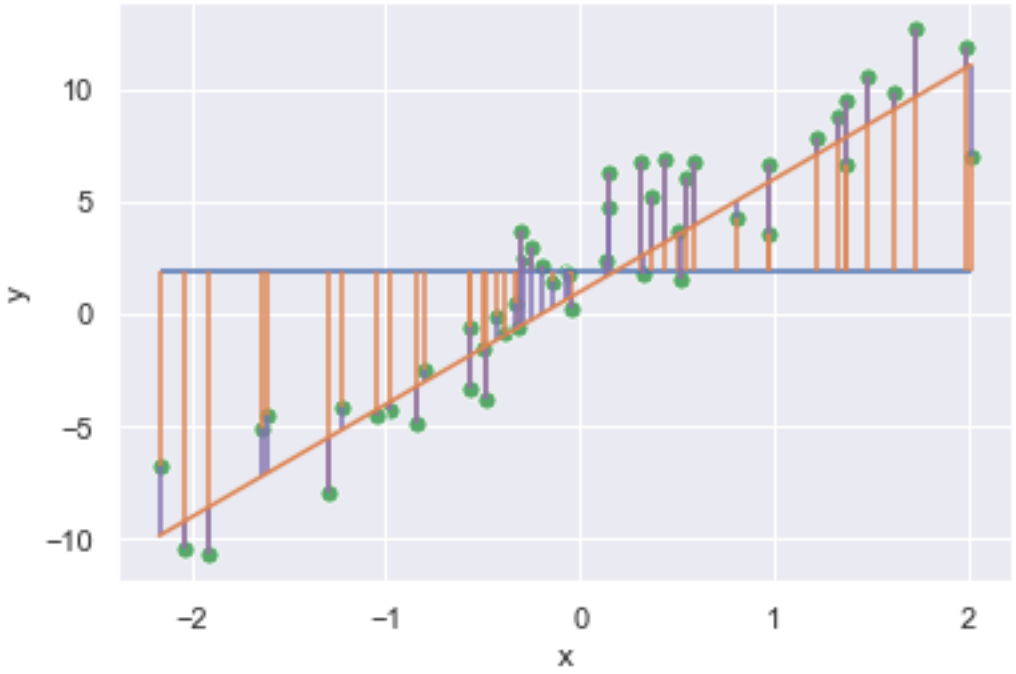
Remember:

The total variance of y is shown in the first plot, where each vertical line is $y_i - \bar{y}$

The *unexplained* variance of y is shown in the second plot, where each vertical line is the error of the model, $y_i - \hat{y}_i$

In the next plot, we'll combine them to get some intuition regarding the *fraction of unexplained variance*. The purple part of each vertical bar is the *unexplained* part, while the orange part is *explained* by the linear regression.

```
x_line = [np.min(x_test), np.max(x_test)]
y_line = x_line*reg_noisy.coef_ + reg_noisy.intercept_

plt.hlines(mean_y, xmin=np.min(x_test), xmax=np.max(x_test));
plt.vlines(x_test, ymin=mean_y, ymax=y_test, color=sns.color_palette()[1]);
plt.vlines(x_test, ymin=y_test, ymax=y_test_hat, color=sns.color_palette()[4]);
sns.scatterplot(x=x_test.squeeze(), y=y_test, color=sns.color_palette()[2], s=50);
sns.lineplot(x=x_line, y=y_line, color=sns.color_palette()[1]);
plt.xlabel('x');
plt.ylabel('y');
```



Fraction of variance unexplained is the ratio of the sum of squared distances from data to the regression line (sum of squared vertical distances in second plot), to the sum of squared distanced from data to the mean (sum of squared vertical distances in first plot):

$$\frac{MSE}{Var(y)} = \frac{Var(y - \hat{y})}{Var(y)} = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$$

Alternative interpretation: imagine we would develop a very simple ML model, in which we always predict $\hat{y}_i = \bar{y}_i$. Then, we use this model as a basis for comparison for other, more sophisticated models. The ratio above is the ratio of error of the regression model, to the error of a "prediction by mean" model.

- If this quantity is less than 1, our model is better than “prediction by mean”
- If this quantity is greater than 1, our model is worse than “prediction by mean”

```
fvu = mse_noisy/var_y
fvu
```

```
0.144728120279874
```

```
r2 = 1 - fvu
r2
```

```
0.855271879720126
```

```
# another way to do the same thing...
metrics.r2_score(y_test, y_test_hat)
```

```
0.855271879720126
```

What does a negative R2 mean, in terms of a comparison to “prediction by mean”?

More on coefficient value, metrics

```
@interact(sigma = widgets.IntSlider(min=0, max=5, step=1, value=2),
   coef = widgets.IntSlider(min=-5, max=5, step=1, value=2))
def plot_reg(sigma, coef):
    x_train, y_train = generate_linear_regression_data(n=20000, d=1, coef=coef,
        intercept=intercept, sigma=sigma)
    x_test,  y_test =  generate_linear_regression_data(n=10000, d=1, coef=coef,
        intercept=intercept, sigma=sigma)
    r_mod = LinearRegression().fit(x_train, y_train)
    r2_test = r_mod.score(x_test, y_test)
    mse_test = metrics.mean_squared_error(y_test, r_mod.predict(x_test))
    x_line = np.array([-3, 3])
    y_line = r_mod.predict(x_line.reshape(-1,1))
    plt.axhline(y=np.mean(y_train), color=sns.color_palette()[1]);
    sns.lineplot(x=x_line, y=y_line, color=sns.color_palette()[2]);
    sns.scatterplot(x=x_test[:100,].squeeze(), y=y_test[:100], s=50);
    plt.xlabel('x');
    plt.ylabel('y');
    plt.ylim(-20,20)
    plt.xlim(-3,3)
    plt.title("MSE: %f\nR2: %f" % (mse_test, r2_test) )
```

Remember that in this data, the *only* source of error is the ϵ in

$$y_i = w_0 + w_1 x_{i,1} + ... + w_d x_{i,d} + \epsilon_i$$

where $\epsilon_i \sim N(0, \sigma^2)$. If not for this, our regression would fit the data perfectly.

Interpreting the coefficient:

> An increase in x of 1 is, on average, associated with an increase in y of about w_1.

Note that it does not imply any causal relationship!

Interpreting MSE and R2:

- MSE shows us the variance of the data around the regression line (for data with this specific type of "noise").
- MSE is a measure of the model error, not relative to any baseline. We can use it to compare different models on the same dataset (but not on different datasets).
- R2 tells us what fraction of the variance in the data is "explained" by the regression line.
- R2 is a measure relative to the "prediction by mean" baseline. (Note that if "prediction by mean" is already good, even a well fitting regression line will not have a high R2.)
- Prediction by mean is the same thing as prediction by a line with

$$w_0 = \overline{y}, w_1 = 0$$

- The greater the true w_1, the more "wrong" the $w_1 = 0$ "prediction" is.

Residual analysis

```
df = sns.load_dataset("anscombe")
df.groupby('dataset').agg({'x': ['count','mean', 'std'], 'y': ['count','mean', 'std']})
```

```
           x                   y
       count mean       std count      mean       std
dataset
I         11  9.0  3.316625    11  7.500909  2.031568
II        11  9.0  3.316625    11  7.500909  2.031657
III       11  9.0  3.316625    11  7.500000  2.030424
IV        11  9.0  3.316625    11  7.500909  2.030579
```

```
data_i   = df[df['dataset'].eq('I')]
data_ii  = df[df['dataset'].eq('II')]
data_iii = df[df['dataset'].eq('III')]
data_iv  = df[df['dataset'].eq('IV')]
```

```
reg_i   = LinearRegression().fit(data_i[['x']],   data_i['y'])
reg_ii  = LinearRegression().fit(data_ii[['x']],  data_ii['y'])
reg_iii = LinearRegression().fit(data_iii[['x']], data_iii['y'])
reg_iv  = LinearRegression().fit(data_iv[['x']],  data_iv['y'])
```

```
print("Dataset I:   ",   reg_i.coef_,   reg_i.intercept_)
print("Dataset II:  ",  reg_ii.coef_,  reg_ii.intercept_)
print("Dataset III: ", reg_iii.coef_, reg_iii.intercept_)
print("Dataset IV:  ",  reg_iv.coef_,  reg_iv.intercept_)
```

```
Dataset I:    [0.50009091] 3.0000909090909094
Dataset II:   [0.5] 3.000909090909089
Dataset III:  [0.49972727] 3.002454545454544
Dataset IV:   [0.49990909] 3.0017272727272726
```

```
print("Dataset I:   ", metrics.r2_score(data_i['y'],  reg_i.predict(data_i[['x']])))
print("Dataset II:  ", metrics.r2_score(data_ii['y'], reg_ii.predict(data_ii[['x']])))
print("Dataset III: ", metrics.r2_score(data_iii['y'],reg_iii.predict(data_iii[['x']])))
print("Dataset IV:  ", metrics.r2_score(data_iv['y'], reg_iv.predict(data_iv[['x']])))
```

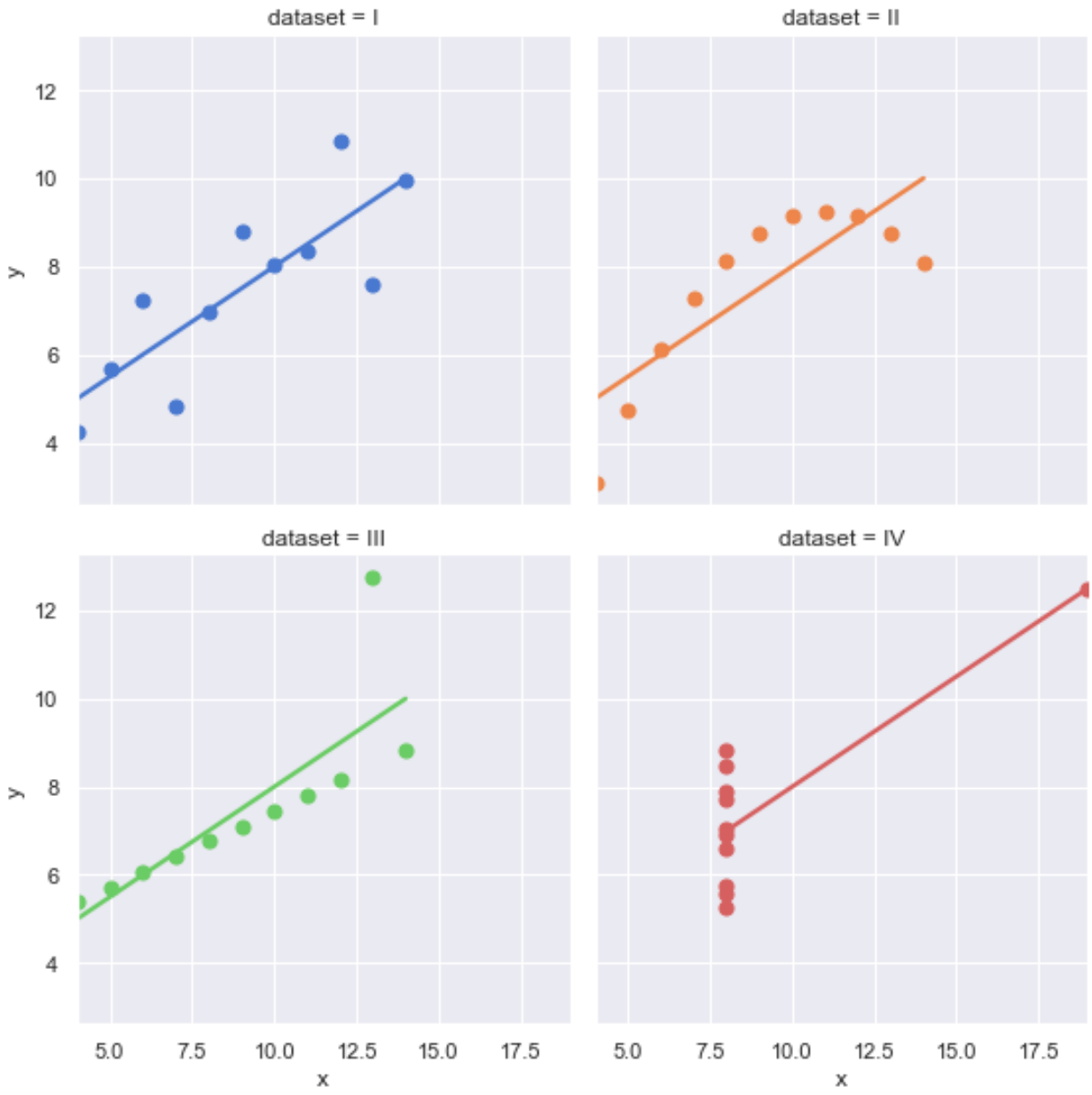
```
Dataset I:    0.6665424595087748
Dataset II:   0.6662420337274844
Dataset III:  0.6663240410665591
Dataset IV:   0.6667072568984653
```

```
print("Dataset I:   ", metrics.mean_squared_error(data_i['y'],
    reg_i.predict(data_i[['x']])))
print("Dataset II:  ", metrics.mean_squared_error(data_ii['y'],
    reg_ii.predict(data_ii[['x']])))
print("Dataset III: ",
    metrics.mean_squared_error(data_iii['y'],reg_iii.predict(data_iii[['x']])))
print("Dataset IV:  ", metrics.mean_squared_error(data_iv['y'],
    reg_iv.predict(data_iv[['x']])))
```

```
Dataset I:    1.2511536363636366
Dataset II:   1.2523900826446281
Dataset III:  1.250562892561984
Dataset IV:   1.249317272727273
```

All of these models are equally “good” according to our scoring metrics... BUT

```
sns.lmplot(x="x", y="y", col="dataset", hue="dataset",
           data=df, col_wrap=2, ci=None, palette="muted", height=4,
           scatter_kws={"s": 50, "alpha": 1});
```



Does the linear model fit well?

- the linear model is a good fit for Dataset I
- Dataset II is clearly non-linear
- Dataset III has an outlier
- Dataset IV has a high leverage point

Easy to identify problems in 1D - what about in higher D?

- Plot $\hat{y}$ against y

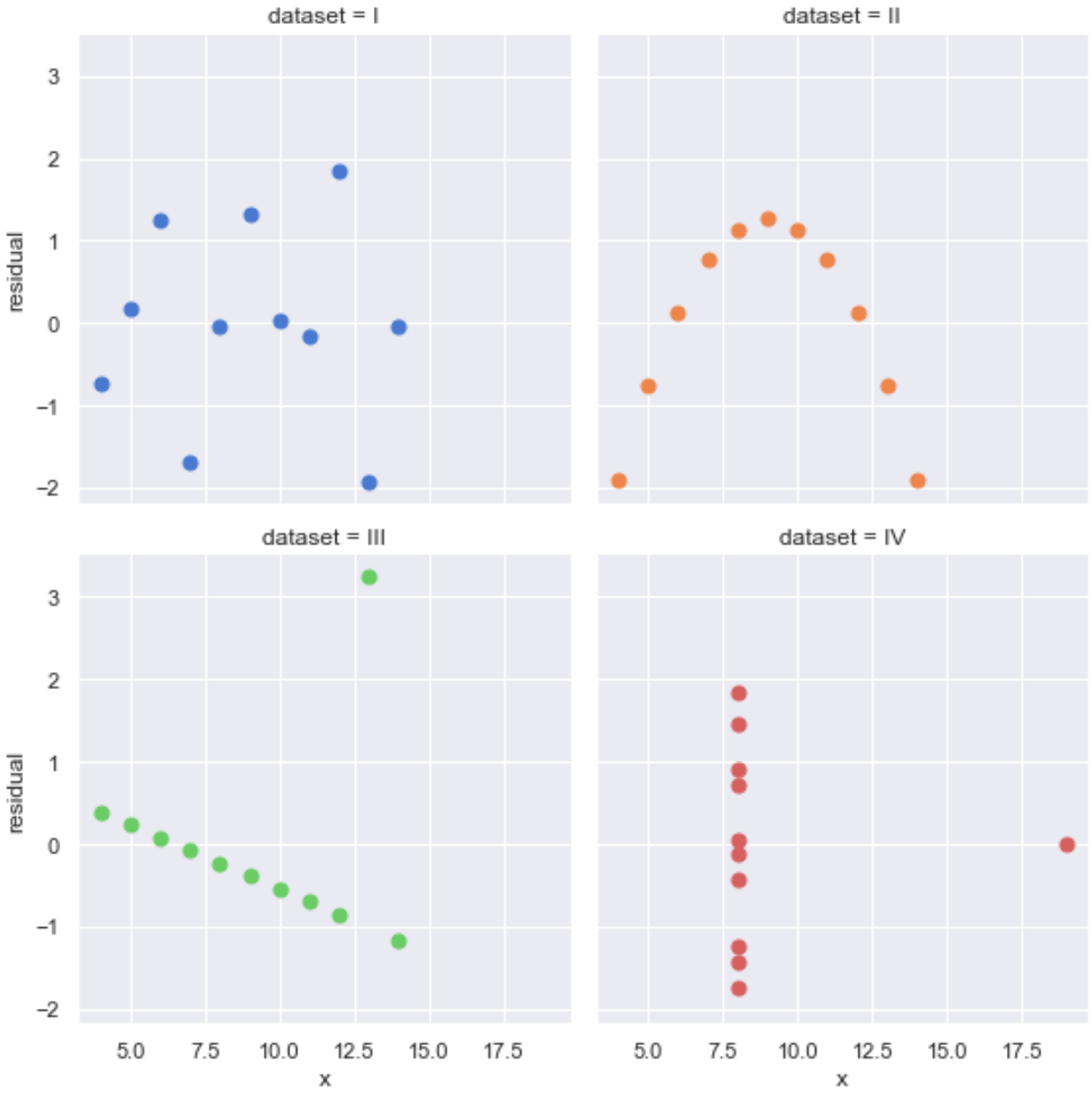
- Plot residuals against $\widehat{y}$
- Plot residuals against each x (including any x not in the model)
- Plot residuals against time (for time series data)

What should each of these plots look like if the regression is "good"?

```
data_i   = data_i.assign(   yhat = reg_i.predict(  data_i[['x']]) )
data_ii  = data_ii.assign(  yhat = reg_ii.predict( data_ii[['x']]) )
data_iii = data_iii.assign( yhat = reg_iii.predict( data_iii[['x']]) )
data_iv  = data_iv.assign(  yhat = reg_iv.predict(  data_iv[['x']]) )

data_i   = data_i.assign(   residual = data_i['y'] - data_i['yhat'] )
data_ii  = data_ii.assign(  residual = data_ii['y'] - data_ii['yhat'] )
data_iii = data_iii.assign( residual = data_iii['y'] - data_iii['yhat'] )
data_iv  = data_iv.assign(  residual = data_iv['y'] - data_iv['yhat'] )

data_all = pd.concat([data_i, data_ii, data_iii, data_iv])
data_all.head()
```

```
  dataset     x     y      yhat  residual
0       I  10.0  8.04  8.001000  0.039000
1       I   8.0  6.95  7.000818 -0.050818
2       I  13.0  7.58  9.501273 -1.921273
3       I   9.0  8.81  7.500909  1.309091
4       I  11.0  8.33  8.501091 -0.171091
```

```
sns.lmplot(x="x", y="residual", col="dataset", hue="dataset",
           data=data_all, col_wrap=2, ci=None, palette="muted", height=4,
           scatter_kws={"s": 50, "alpha": 1}, fit_reg=False);
```



Multiple linear regression

Generate some data

```
x_train, y_train = generate_linear_regression_data(n=n_samples, d=2, coef=[5,5],
    intercept=intercept)
x_test,  y_test  = generate_linear_regression_data(n=50, d=2, coef=[5,5],
    intercept=intercept)
```

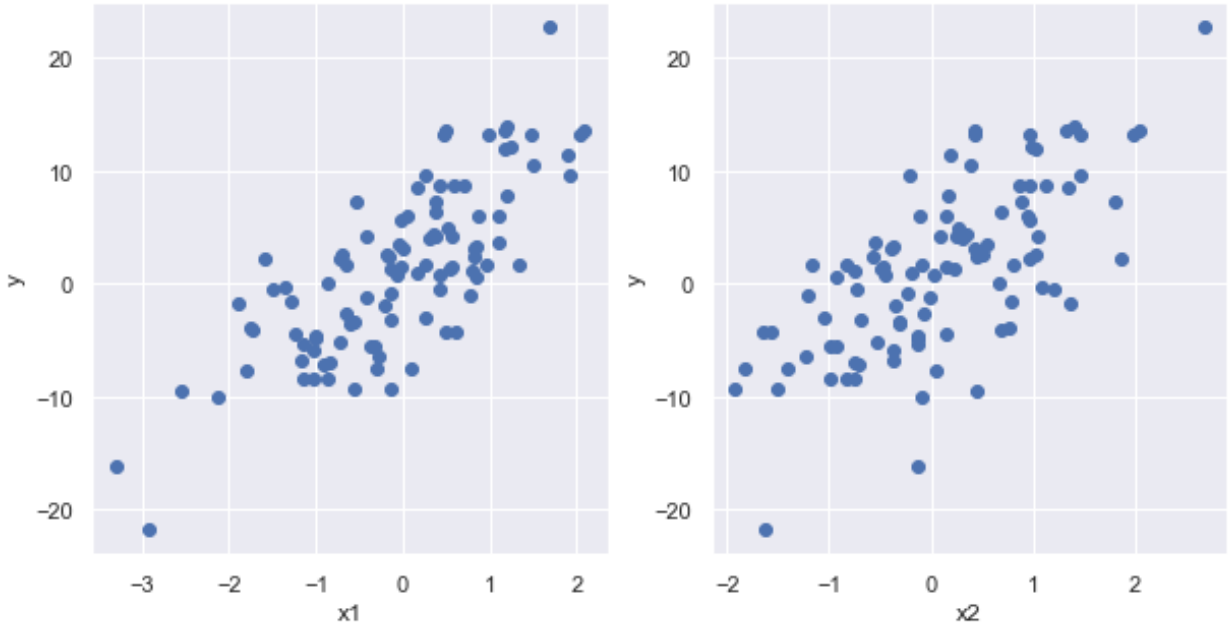
```
x_train.shape
```

```
(100, 2)
```

```
y_train.shape
```

```
(100,)
```

```
plt.figure(figsize=(10,5));
plt.subplot(1,2,1);
plt.scatter(x_train[:,0],  y_train);
plt.xlabel("x1");
plt.ylabel("y");
plt.subplot(1,2,2);
plt.scatter(x_train[:,1],  y_train);
plt.xlabel("x2");
plt.ylabel("y");
```



Recall that there is no stochastic noise in this data - so it fits a linear model perfectly. But it's more difficult to see that linear relationship in higher dimensions.

Fit a linear regression

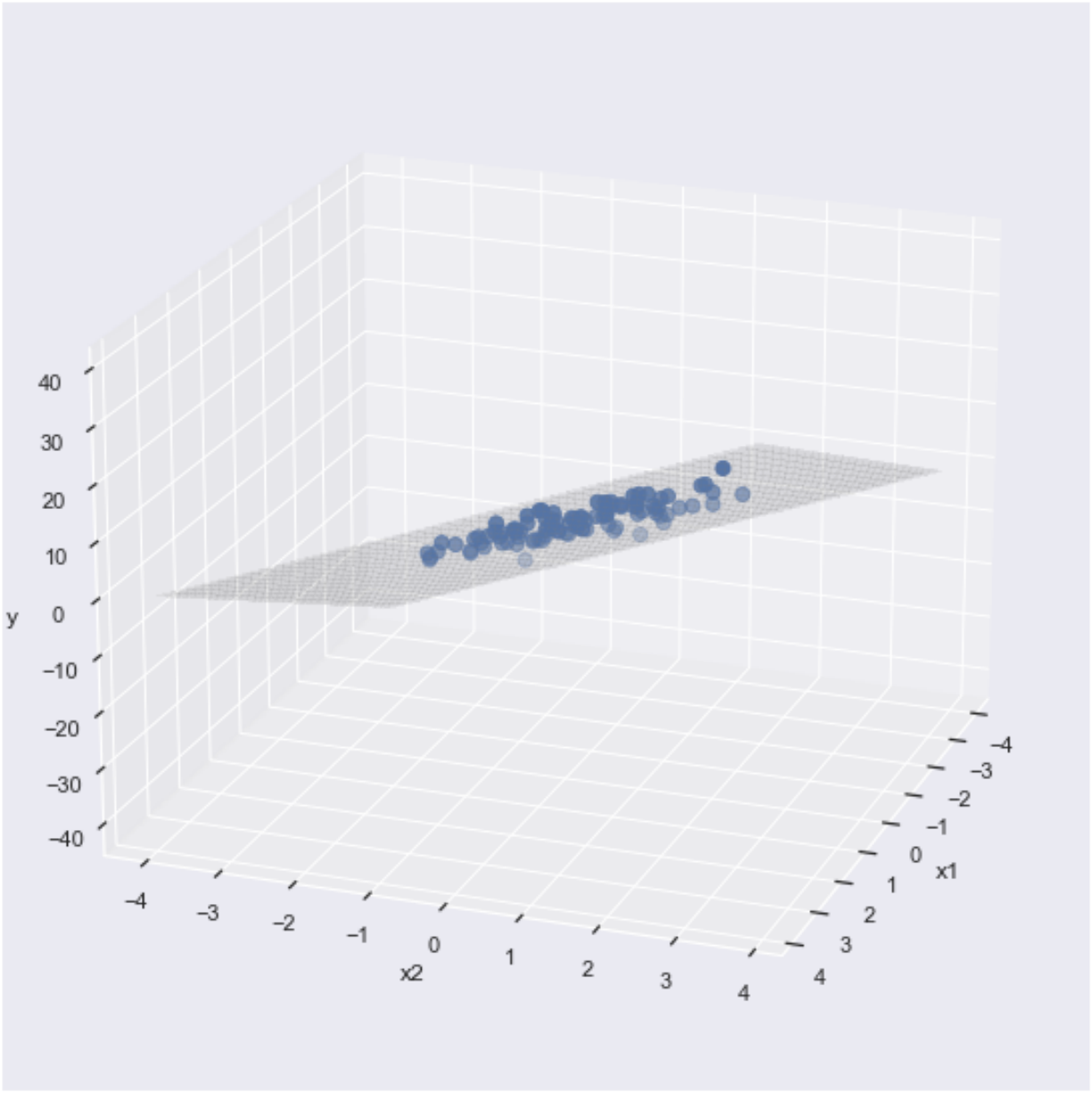
```
reg_multi = LinearRegression().fit(x_train, y_train)
print("Coefficient list: ", reg_multi.coef_)
print("Intercept: " , reg_multi.intercept_)
```

```
Coefficient list:  [5. 5.]
Intercept:  1.0
```

Plot hyperplane

```
def plot_3D(elev=20, azim=-20, X=x_train, y=y_train):
    plt.figure(figsize=(10,10))
```

```
    ax = plt.subplot(projection='3d')


    X1 = np.arange(-4, 4, 0.2)
    X2 = np.arange(-4, 4, 0.2)
    X1, X2 = np.meshgrid(X1, X2)
    Z = X1*reg_multi.coef_[0] + X2*reg_multi.coef_[1]

    # Plot the surface.
    ax.plot_surface(X1, X2, Z, alpha=0.1, color='gray',
                          linewidth=0, antialiased=False)
    ax.scatter3D(X[:, 0], X[:, 1], y, s=50)

    ax.view_init(elev=elev, azim=azim)
    ax.set_xlabel('x1')
    ax.set_ylabel('x2')
    ax.set_zlabel('y')

interact(plot_3D, elev=widgets.IntSlider(min=-90, max=90, step=10, value=20),
          azim=widgets.IntSlider(min=-90, max=90, step=10, value=20),
         X=fixed(x_train), y=fixed(y_train));
```



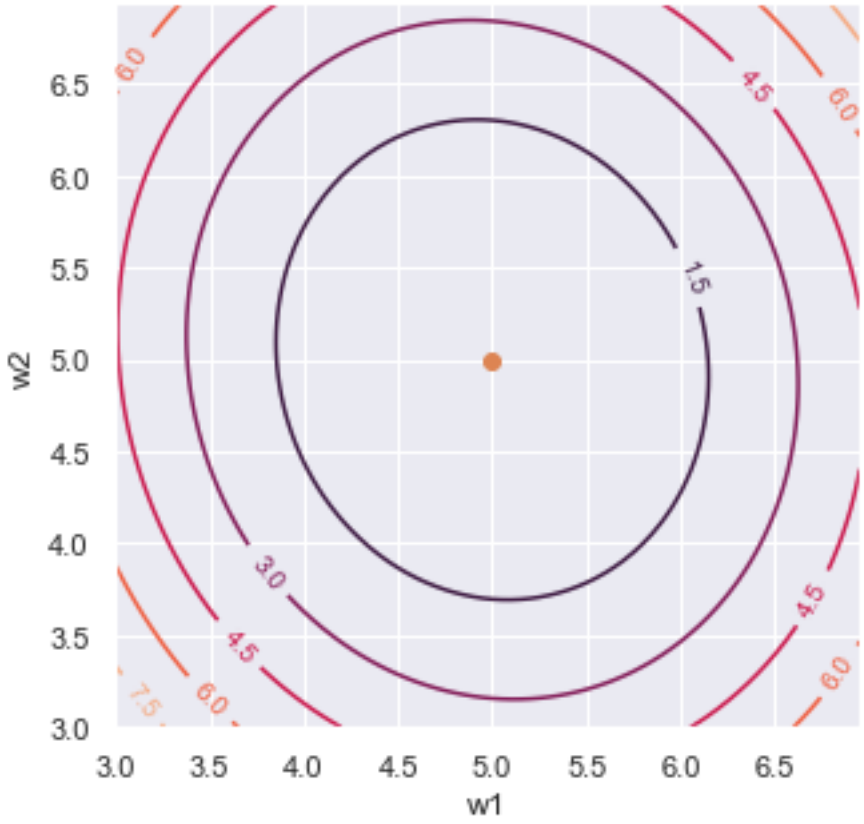
MSE contour

```
coefs = np.arange(3.0, 7.0, 0.05)

coef_grid = np.array(np.meshgrid(coefs, coefs)).reshape(1, 2, coefs.shape[0], coefs.shape[0])
y_train_hat_c = (reg_multi.intercept_ + np.sum(coef_grid * x_train.reshape(x_train.shape[0],
    2, 1, 1), axis=1) )
mses_train = np.mean((y_train_hat_c- y_train.reshape(-1, 1, 1))**2, axis=0)
```

```
plt.figure(figsize=(5,5));
p = plt.scatter(x=reg_multi.coef_[0], y=reg_multi.coef_[1], c=sns.color_palette()[1])
p = plt.contour(coef_grid[0, 0, :, :], coef_grid[0, 1, :, :], mses_train, levels=5);
plt.clabel(p, inline=1, fontsize=10);
```

```
plt.xlabel('w1');
plt.ylabel('w2');
```

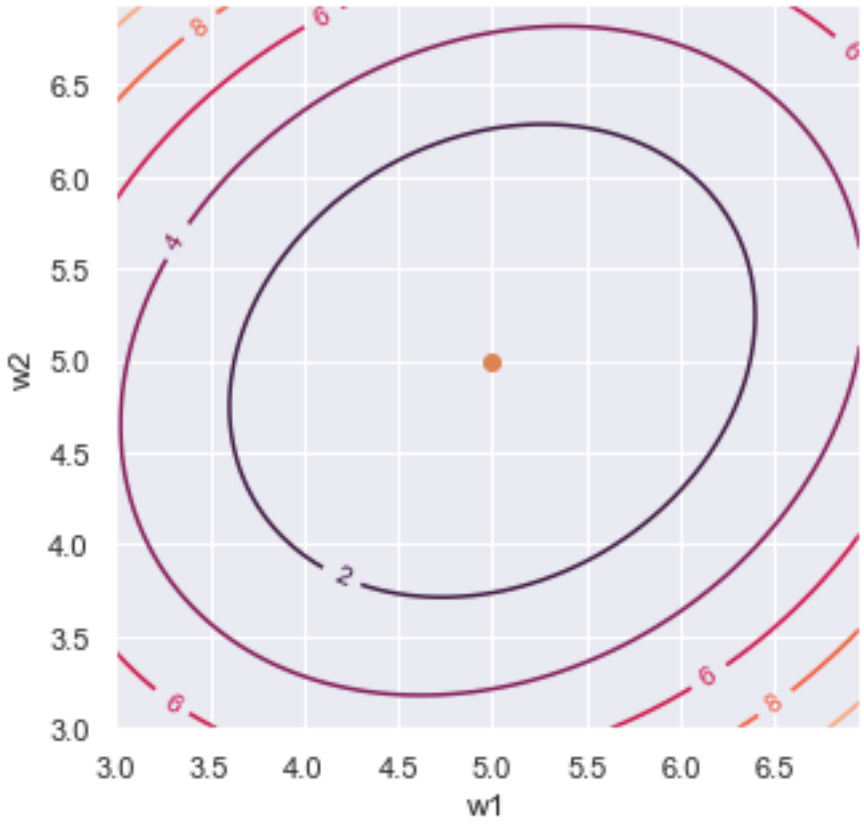
```
*c* argument looks like a single numeric RGB or RGBA sequence, which should be avoided as
    value-mapping will have precedence in case its length matches with *x* & *y*.  Please
    use the *color* keyword-argument or provide a 2D array with a single row if you intend
    to specify the same RGB or RGBA value for all points.
```




```
coefs = np.arange(3.0, 7.0, 0.05)

coef_grid = np.array(np.meshgrid(coefs, coefs)).reshape(1, 2, coefs.shape[0], coefs.shape[0])
y_test_hat_c = (reg_multi.intercept_ + np.sum(coef_grid * x_test.reshape(x_test.shape[0], 2,
    1, 1), axis=1) )
mses_test = np.mean((y_test_hat_c- y_test.reshape(-1, 1, 1))**2, axis=0)
```

```
plt.figure(figsize=(5,5));
p = plt.scatter(x=reg_multi.coef_[0], y=reg_multi.coef_[1], c=sns.color_palette()[1])
p = plt.contour(coef_grid[0, 0, :, :], coef_grid[0, 1, :, :], mses_test, levels=5);
plt.clabel(p, inline=1, fontsize=10);
plt.xlabel('w1');
plt.ylabel('w2');
```

```
*c* argument looks like a single numeric RGB or RGBA sequence, which should be avoided as
    value-mapping will have precedence in case its length matches with *x* & *y*.  Please
    use the *color* keyword-argument or provide a 2D array with a single row if you intend
    to specify the same RGB or RGBA value for all points.
```

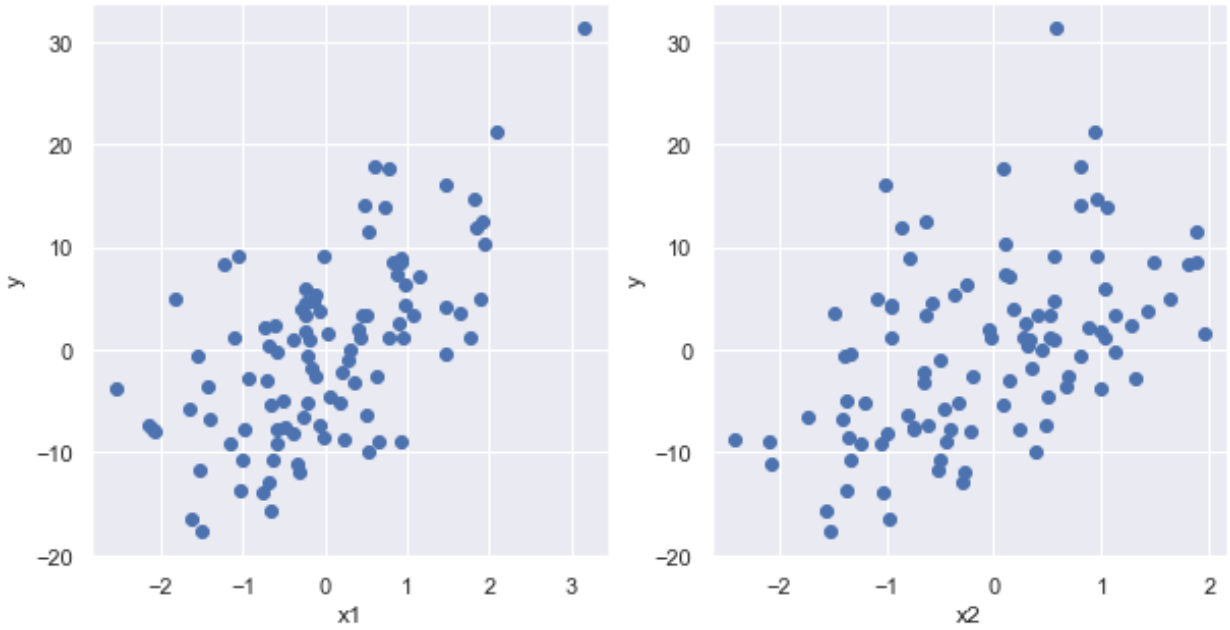


Multiple linear regression with noise

Generate some data

```
x_train, y_train = generate_linear_regression_data(n=n_samples, d=2, coef=[5,5],
    intercept=intercept, sigma=5)
x_test,  y_test  = generate_linear_regression_data(n=50, d=2, coef=[5,5],
    intercept=intercept, sigma=5)
```

```
plt.figure(figsize=(10,5));
plt.subplot(1,2,1);
plt.scatter(x_train[:,0],  y_train);
plt.xlabel("x1");
plt.ylabel("y");
plt.subplot(1,2,2);
plt.scatter(x_train[:,1],  y_train);
plt.xlabel("x2");
plt.ylabel("y");
```



Fit a linear regression

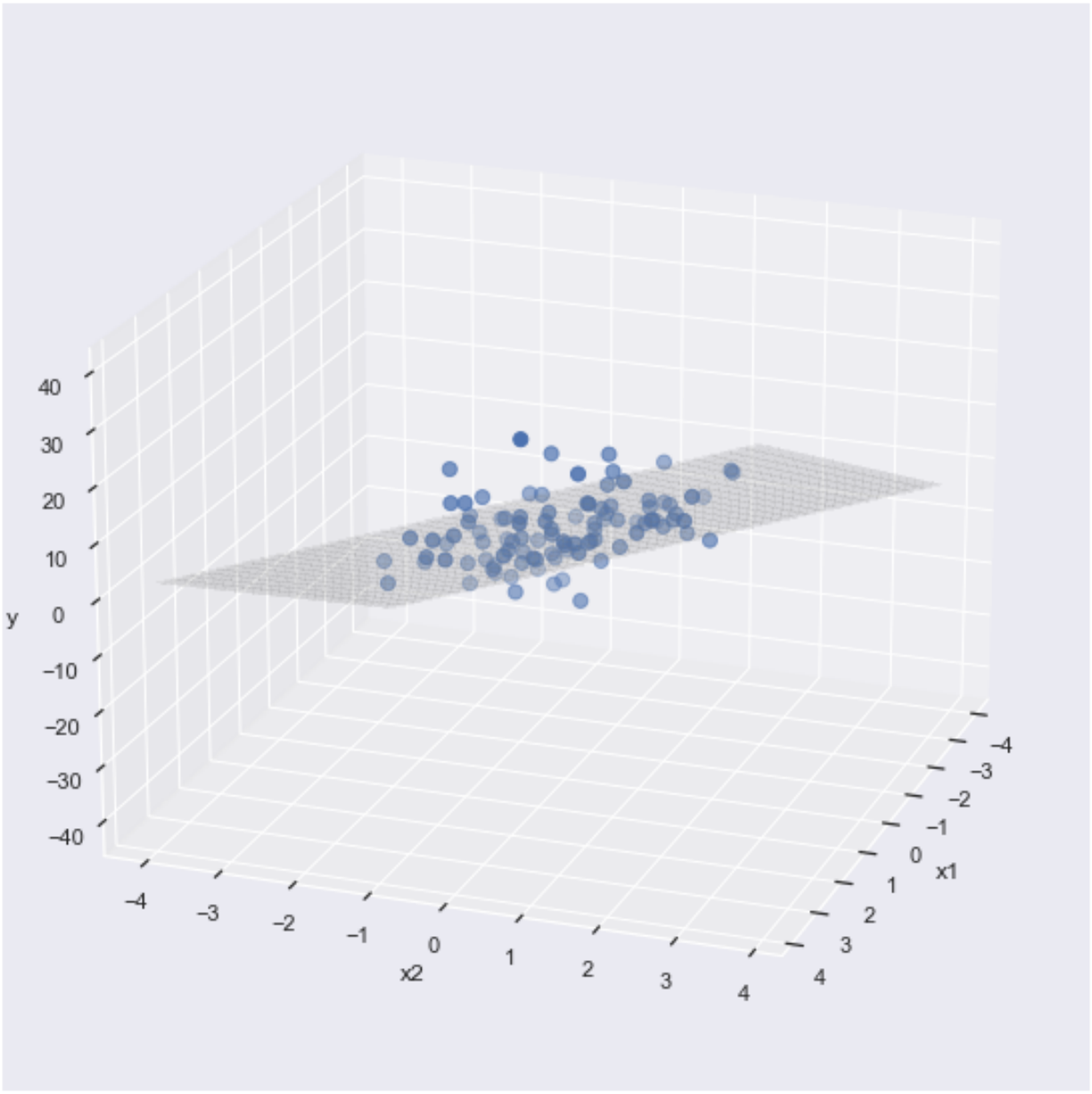
```
reg_multi_noisy = LinearRegression().fit(x_train, y_train)
print("Coefficient list: ", reg_multi_noisy.coef_)
print("Intercept: " , reg_multi_noisy.intercept_)
```

```
Coefficient list:  [5.38523768 4.74850936]
Intercept:  0.4822959554317295
```

Plot hyperplane

```
def plot_3D(elev=20, azim=-20, X=x_train, y=y_train):
    plt.figure(figsize=(10,10))
    ax = plt.subplot(projection='3d')


    X1 = np.arange(-4, 4, 0.2)
    X2 = np.arange(-4, 4, 0.2)
    X1, X2 = np.meshgrid(X1, X2)
    Z = X1*reg_multi_noisy.coef_[0] + X2*reg_multi_noisy.coef_[1]

    # Plot the surface.
    ax.plot_surface(X1, X2, Z, alpha=0.1, color='gray',
                          linewidth=0, antialiased=False)
    ax.scatter3D(X[:, 0], X[:, 1], y, s=50)

    ax.view_init(elev=elev, azim=azim)
    ax.set_xlabel('x1')
    ax.set_ylabel('x2')
    ax.set_zlabel('y')
```

```
interact(plot_3D, elev=widgets.IntSlider(min=-90, max=90, step=10, value=20),
          azim=widgets.IntSlider(min=-90, max=90, step=10, value=20),
         X=fixed(x_train), y=fixed(y_train));
```

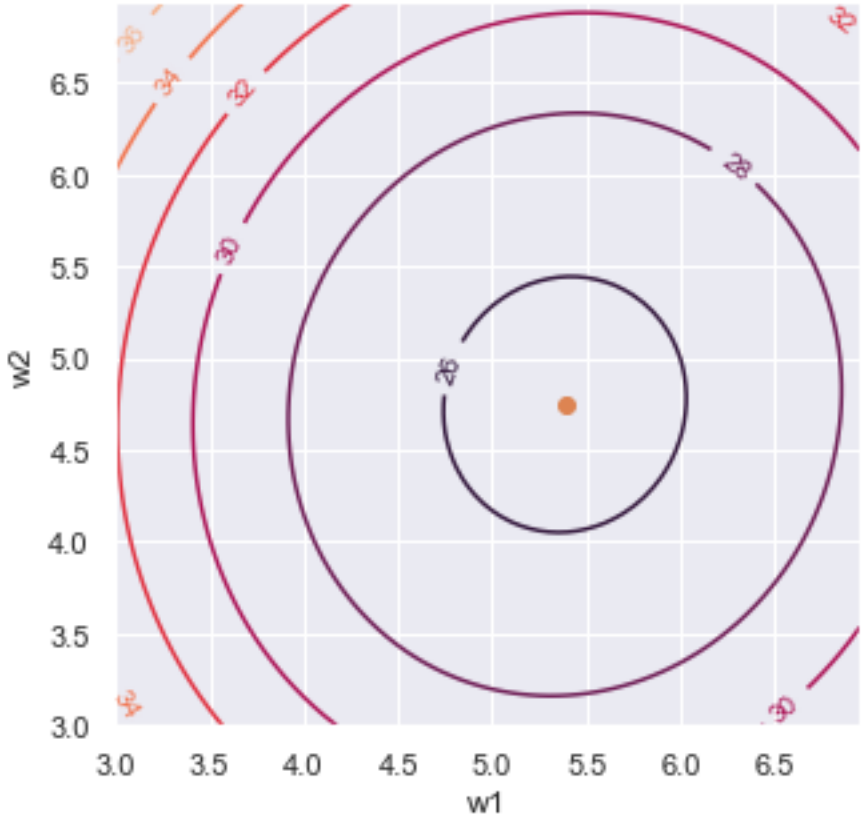


MSE contour

```
coefs = np.arange(3.0, 7.0, 0.05)

coef_grid = np.array(np.meshgrid(coefs, coefs)).reshape(1, 2, coefs.shape[0], coefs.shape[0])
y_train_hat_c = (reg_multi_noisy.intercept_ + np.sum(coef_grid *
    x_train.reshape(x_train.shape[0], 2, 1, 1), axis=1) )
mses_train = np.mean((y_train_hat_c- y_train.reshape(-1, 1, 1))**2, axis=0)
```

```
plt.figure(figsize=(5,5));
p = plt.scatter(x=reg_multi_noisy.coef_[0], y=reg_multi_noisy.coef_[1],
    c=sns.color_palette()[1])
p = plt.contour(coef_grid[0, 0, :, :], coef_grid[0, 1, :, :], mses_train, levels=5);
plt.clabel(p, inline=1, fontsize=10);
plt.xlabel('w1');
plt.ylabel('w2');
```

```
*c* argument looks like a single numeric RGB or RGBA sequence, which should be avoided as
    value-mapping will have precedence in case its length matches with *x* & *y*.  Please
    use the *color* keyword-argument or provide a 2D array with a single row if you intend
    to specify the same RGB or RGBA value for all points.
```

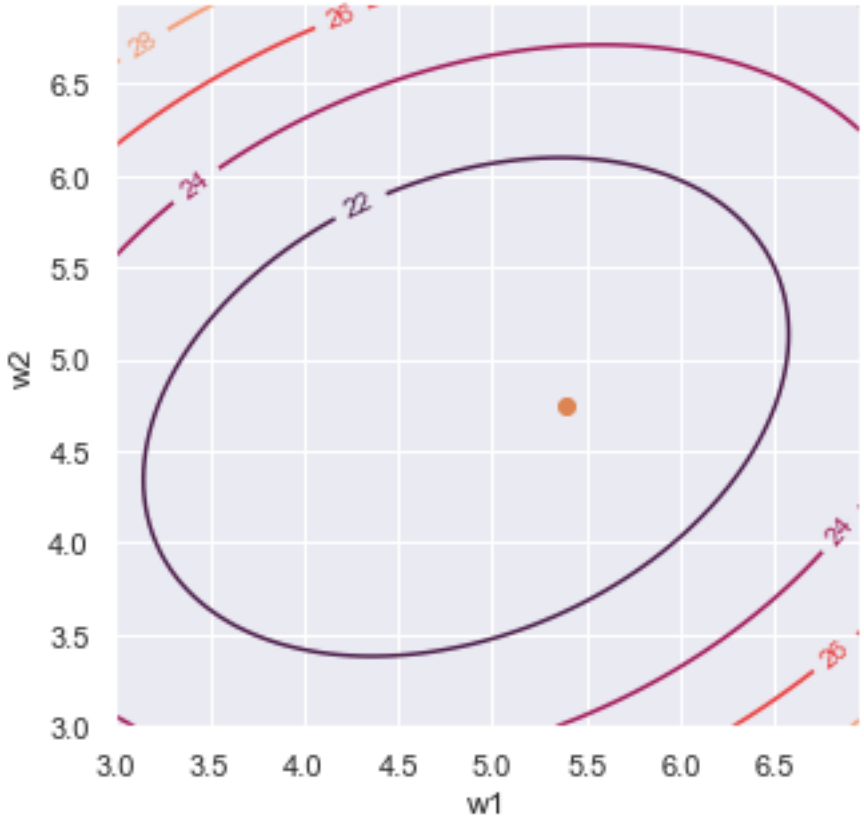



```
coefs = np.arange(3.0, 7.0, 0.05)

coef_grid = np.array(np.meshgrid(coefs, coefs)).reshape(1, 2, coefs.shape[0], coefs.shape[0])
y_test_hat_c = (reg_multi_noisy.intercept_ + np.sum(coef_grid *
    x_test.reshape(x_test.shape[0], 2, 1, 1), axis=1) )
mses_test = np.mean((y_test_hat_c- y_test.reshape(-1, 1, 1))**2, axis=0)
```

```
plt.figure(figsize=(5,5));
p = plt.scatter(x=reg_multi_noisy.coef_[0], y=reg_multi_noisy.coef_[1],
    c=sns.color_palette()[1])
p = plt.contour(coef_grid[0, 0, :, :], coef_grid[0, 1, :, :], mses_test, levels=5);
plt.clabel(p, inline=1, fontsize=10);
```

```
plt.xlabel('w1');
plt.ylabel('w2');
```

```
*c* argument looks like a single numeric RGB or RGBA sequence, which should be avoided as
    value-mapping will have precedence in case its length matches with *x* & *y*.  Please
    use the *color* keyword-argument or provide a 2D array with a single row if you intend
    to specify the same RGB or RGBA value for all points.
```



Linear basis function regression

The assumptions of the linear model (that the target variable can be predicted as a linear combination of the features) can be restrictive. We can capture more complicated relationships using linear basis function regression.

Fundamental idea: with a set of "basis" functions, we represent the "shape" of our data as a weighted sum of basis functions:

$$\hat{y_i} = \sum_{j=0}^{p} w_p \phi_p(\mathrm{x_i})$$

(We'll revisit this idea again later in the semester, when we talk about kernels; and again, when we talk about activation functions in neural networks.)

We're going to look at some examples of basis functions (but not an exhaustive list...)

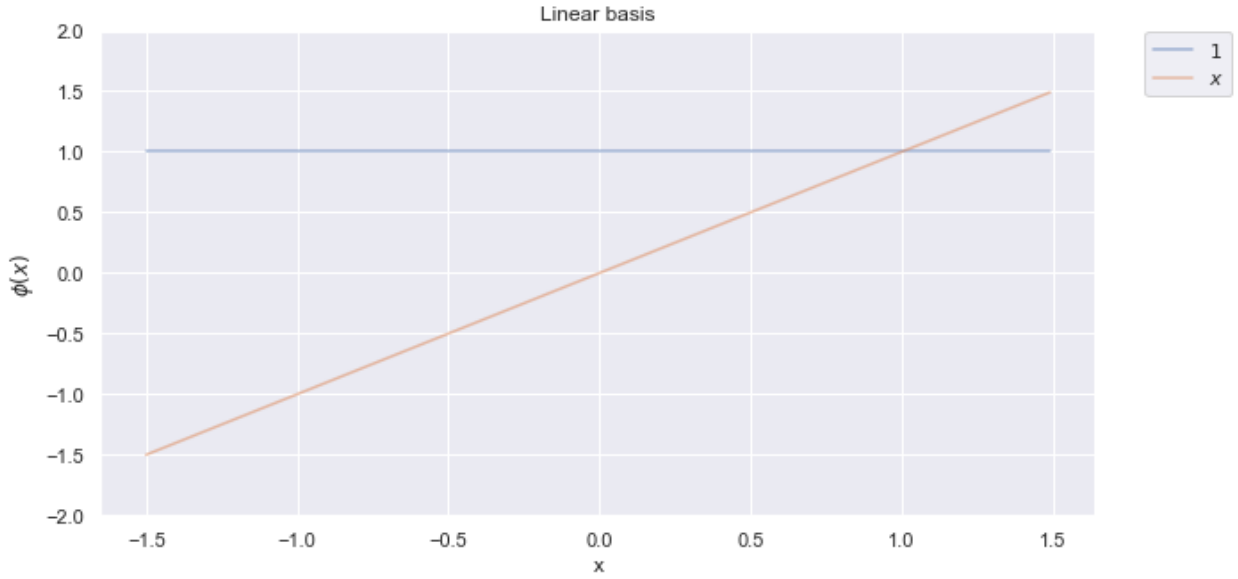
(Note: it's also possible to mix-and-match basis functions from different "families" in the same model! And, you can apply basis functions to multiple features, too.)

Linear basis

Transform a feature x using $\phi_0(x) = 1, \phi_1(x) = x$, i.e.

$$y \approx w_0 + w_1 x$$

```
def linear_basis(x):
  return np.hstack([np.ones(x.shape),x])

x = np.arange(-1.5,1.5,step=0.01).reshape(-1,1)
x_trans = linear_basis(x)

@interact(w0 = widgets.FloatSlider(min=-2, max=2, step=0.1, value=1),
          w1 = widgets.FloatSlider(min=-2, max=2, step=0.1, value=1),
          show_sum = False)
def plot_linear(w0, w1, show_sum):
  plt.figure(figsize=(10,5));
  w = np.array([w0, w1])
  l = ['1', 'x']
  y = np.sum(w*x_trans, axis=1)
  if show_sum:
    sns.lineplot(x=x.squeeze(), y=y, label='sum', alpha=1, lw=2);
  for i in range(2):
    sns.lineplot(x=x.squeeze(), y=w[i]*x_trans[:,i], label='$' + l[i] + '$', alpha=0.5);
  plt.ylim(-2, 2);
  plt.title("Linear basis");
  plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0);
  plt.xlabel('x')
  plt.ylabel('$\phi(x)$')
```



Polynomial basis

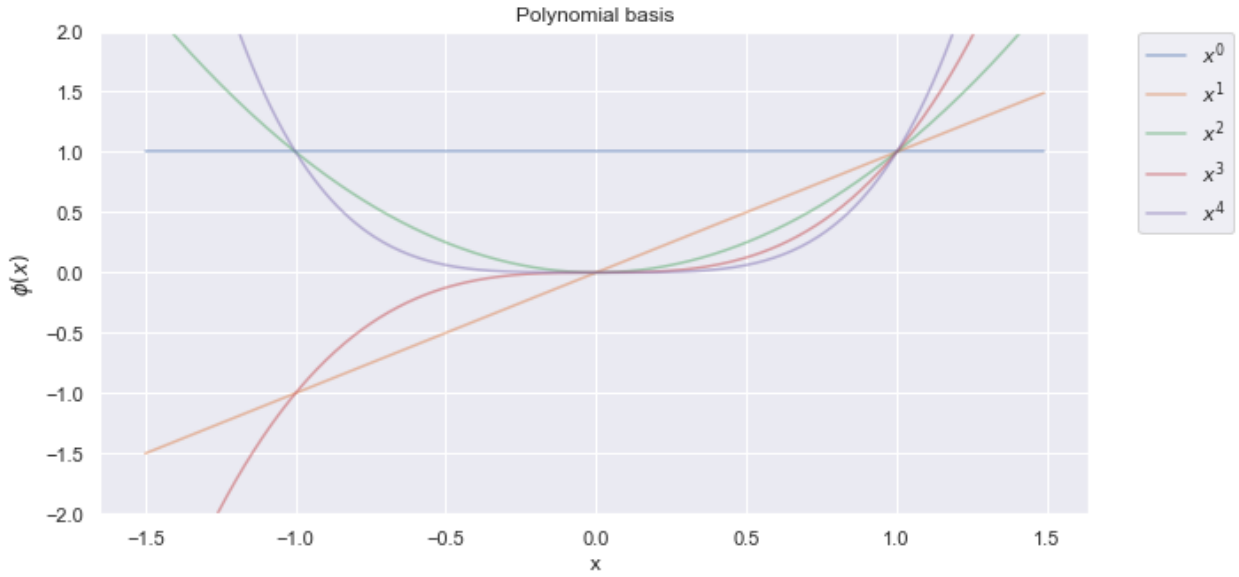
Transform a feature x using $\phi_j(x) = x^j$, i.e.

$$y \approx w_0 x^0 + w_1 x^1 + ... + w_p x^p$$

Note that the model is linear in the parameters w, which is what makes it a linear model even though it is not linear in x.

Issue: polynomials are "global" functions - affect the entire range from $-\infty$ to ∞, and changes very quickly outside the range $[-1, 1]$.

```
def polynomial_basis(x, d):
  return np.hstack([x**i for i in range(d)])

x = np.arange(-1.5,1.5,step=0.01).reshape(-1,1)
x_trans = polynomial_basis(x,5)

@interact(w0 = widgets.FloatSlider(min=-1, max=2, step=0.1, value=1),
          w1 = widgets.FloatSlider(min=-1, max=2, step=0.1, value=1),
          w2 = widgets.FloatSlider(min=-1, max=2, step=0.1, value=1),
          w3 = widgets.FloatSlider(min=-1, max=2, step=0.1, value=1),
          w4 = widgets.FloatSlider(min=-1, max=2, step=0.1, value=1),
          show_sum = False)
def plot_poly(w0, w1, w2, w3, w4, show_sum):
  plt.figure(figsize=(10,5));
  w = np.array([w0, w1, w2, w3, w4])
  y = np.sum(w*x_trans, axis=1)
  if show_sum:
    sns.lineplot(x=x.squeeze(), y=y, label='sum', alpha=1, lw=2);
  for i in range(5):
    sns.lineplot(x=x.squeeze(), y=w[i]*x_trans[:,i], label='$x^' + str(i) + '$', alpha=0.5);
  plt.ylim(-2, 2);
  plt.title("Polynomial basis");
  plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0);
  plt.xlabel('x')
  plt.ylabel('$\phi(x)$')
```



Radial basis

Transform a feature x using $\phi_j(x) = \exp\left(-\frac{(x-\mu_j)^2}{s^2}\right)$, i.e.

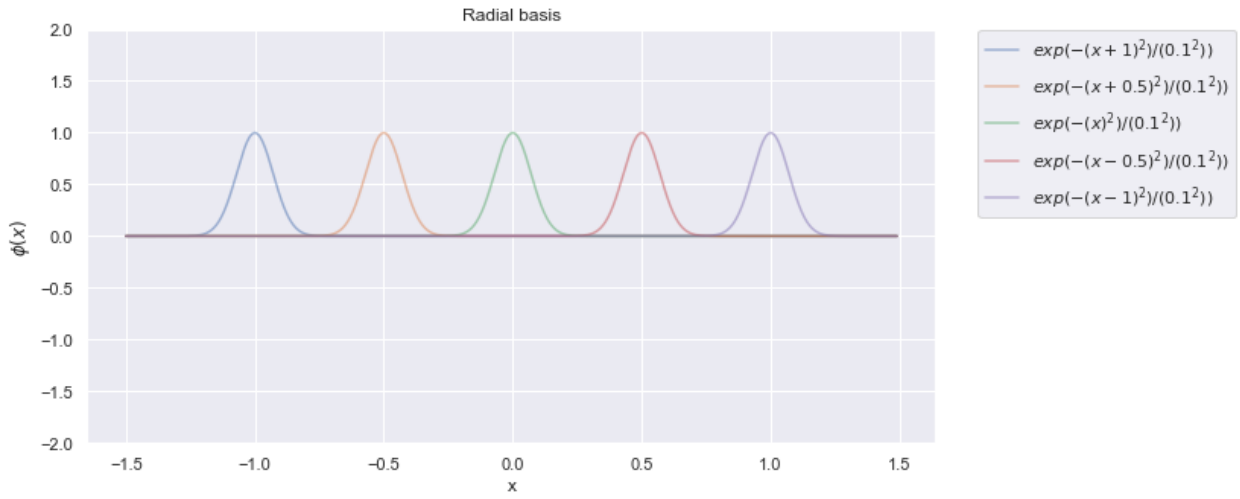
$$y \approx w_0 \exp\left(-\frac{(x-\mu_0)^2}{s^2}\right) + w_1 \exp\left(-\frac{(x-\mu_1)^2}{s^2}\right) + \ldots + w_p \exp\left(-\frac{(x-\mu_p)^2}{s^2}\right)$$

The model is linear in the parameters w, which is what makes it a linear model even though it is not linear in x. However, it is not linear in the basis function parameters μ_j or s! (Those basis function parameters will not be "learned" - they are fixed by you.)

Note that in contrast to the polynomials which had "global" effect, each radial basis function has "local" effect. (Think of it as a weighted sum of little "bumps".)

```
s = 0.1
def radial_basis(x, mu_list):
  return np.hstack([ np.exp(-1*(x-mu)**2/s**2) for mu in mu_list])

x = np.arange(-1.5,1.5,step=0.01).reshape(-1,1)
x_trans = radial_basis(x, [-1, -0.5, 0, 0.5, 1])

@interact(w0 = widgets.FloatSlider(min=-2, max=2, step=0.1, value=1),
          w1 = widgets.FloatSlider(min=-2, max=2, step=0.1, value=1),
          w2 = widgets.FloatSlider(min=-2, max=2, step=0.1, value=1),
          w3 = widgets.FloatSlider(min=-2, max=2, step=0.1, value=1),
          w4 = widgets.FloatSlider(min=-2, max=2, step=0.1, value=1),
          show_sum = False)
def plot_radial(w0, w1, w2, w3, w4, show_sum):
  plt.figure(figsize=(10,5));
  w = np.array([w0, w1, w2, w3, w4])
  labels = ['$exp(-(x+1)^2)/(' + str(s) + '^2))$', '$exp(-(x+0.5)^2)/(' + str(s) + '^2))$',
            '$exp(-(x)^2)/(' + str(s) + '^2))$', '$exp(-(x-0.5)^2)/(' + str(s) + '^2))$',
            '$exp(-(x-1)^2)/(' + str(s) + '^2))$']
```

```
  y = np.sum(w*x_trans, axis=1)
  if show_sum:
    sns.lineplot(x=x.squeeze(), y=y, label='sum', alpha=1, lw=2);
  for i in range(5):
    sns.lineplot(x=x.squeeze(), y=w[i]*x_trans[:,i], label=labels[i], alpha=0.5);
  plt.ylim(-2, 2)
  plt.title("Radial basis");
  plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0);
  plt.xlabel('x')
  plt.ylabel('$\phi(x)$')
```



Sigmoidal basis

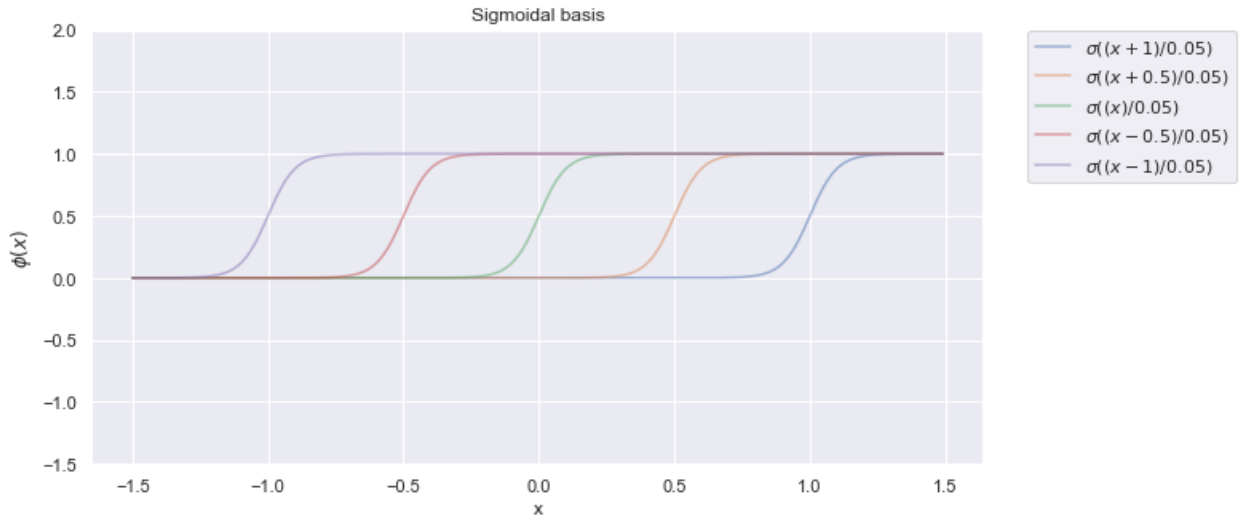
Transform a feature x using $_j(x) = ( ) $ where $\sigma(a) = \frac{1}{1+\exp(-a)}$, i.e.

$$y \approx w_0\sigma\left(\frac{(x-\mu_0)}{s}\right) + w_1\sigma\left(\frac{(x-\mu_1)}{s}\right) + \ldots + w_p\sigma\left(\frac{(x-\mu_p)}{s}\right)$$

(Similar to the RBF but with “steps” instead of “bumps”...)

```
s = 0.05
def sigmoid_basis(x, mu_list):
  return np.hstack([((1+np.exp((-x-mu)/s))**-1) for mu in mu_list])

x = np.arange(-1.5,1.5,step=0.01).reshape(-1,1)
x_trans = sigmoid_basis(x, [-1, -0.5, 0, 0.5, 1])

@interact(w0 = widgets.FloatSlider(min=-2, max=2, step=0.1, value=1),
          w1 = widgets.FloatSlider(min=-2, max=2, step=0.1, value=1),
          w2 = widgets.FloatSlider(min=-2, max=2, step=0.1, value=1),
          w3 = widgets.FloatSlider(min=-2, max=2, step=0.1, value=1),
          w4 = widgets.FloatSlider(min=-2, max=2, step=0.1, value=1),
          show_sum = False)
def plot_sigmoid(w0, w1, w2, w3, w4, show_sum):
  plt.figure(figsize=(10,5));
  w = np.array([w0, w1, w2, w3, w4])
```

```
  labels = ['$\sigma((x+1)/' + str(s) + ')$', '$\sigma((x+0.5)/' + str(s) + ')$',
            '$\sigma((x)/' + str(s) + ')$', '$\sigma((x-0.5)/' + str(s) + ')$',
            '$\sigma((x-1)/' + str(s) + ')$']
  y = np.sum(w*x_trans, axis=1)
  if show_sum:
    sns.lineplot(x=x.squeeze(), y=y, label='sum', alpha=1, lw=2);
  for i in range(5):
    sns.lineplot(x=x.squeeze(), y=w[i]*x_trans[:,i], label=labels[i], alpha=0.5);
  plt.ylim(-1.5, 2)
  plt.title("Sigmoidal basis");
  plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0);
  plt.xlabel('x')
  plt.ylabel('$\phi(x)$')
```



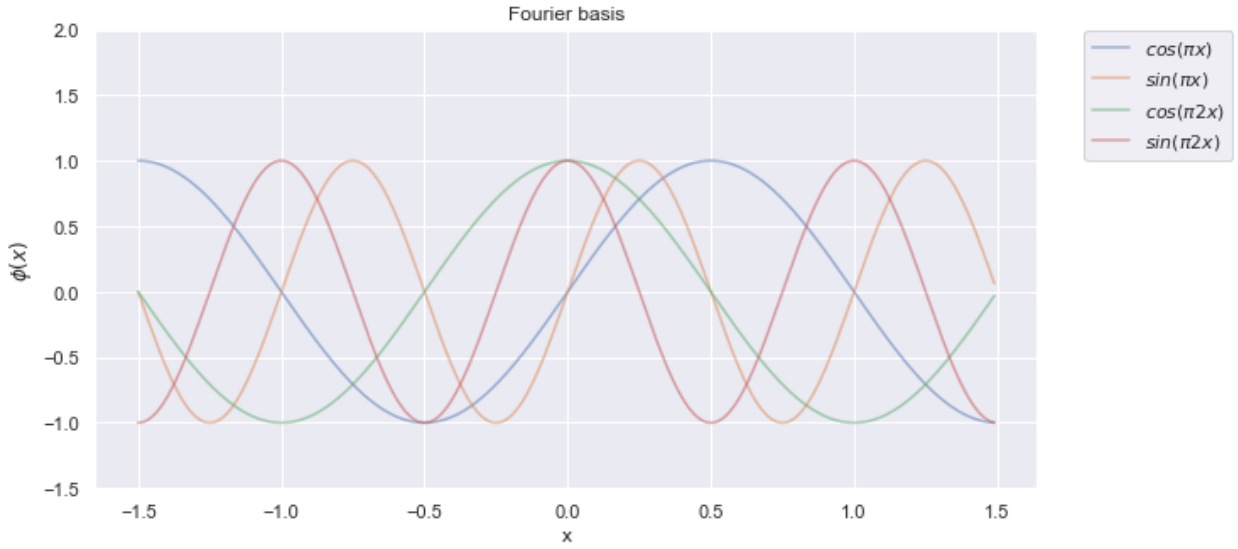
Fourier basis

Transform a feature x using $\phi_j(x) = cos(\pi jx) + sin(\pi jx)$, i.e.

$$y \approx w_0 + w_1 \sin(\pi x) + w_2 \cos(\pi x) + w_3 \sin(\pi 2x) + w_4 \cos(\pi 2x) + ...$$

```
def fourier_basis(x, d):
  sins = np.hstack([np.sin(np.pi*i*x) for i in range(1,d+1)])
  coss = np.hstack([np.cos(np.pi*i*x) for i in range(1,d+1)])
  return np.hstack([sins, coss])

x = np.arange(-1.5,1.5,step=0.01).reshape(-1,1)
x_trans = fourier_basis(x, 2)

@interact(w1 = widgets.FloatSlider(min=-2, max=2, step=0.1, value=1),
          w2 = widgets.FloatSlider(min=-2, max=2, step=0.1, value=1),
          w3 = widgets.FloatSlider(min=-2, max=2, step=0.1, value=1),
          w4 = widgets.FloatSlider(min=-2, max=2, step=0.1, value=1),
          show_sum = False)
def plot_fourier(w1, w2, w3, w4, show_sum):
  plt.figure(figsize=(10,5));
```

```
  w = np.array([w1, w2, w3, w4])
  labels = ["cos(\pi x)", "sin(\pi x)", "cos(\pi 2 x)", "sin(\pi 2 x)"]
  y = np.sum(w*x_trans, axis=1)
  if show_sum:
    sns.lineplot(x=x.squeeze(), y=y, label='sum', alpha=1, lw=2);
  for i in range(4):
    sns.lineplot(x=x.squeeze(), y=w[i]*x_trans[:,i], label='$' + labels[i] + '$', alpha=0.5);
  plt.ylim(-1.5, 2)
  plt.title("Fourier basis");
  plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0);
  plt.xlabel('x')
  plt.ylabel('$\phi(x)$')
```



Generate some data

Let's consider an example! Now suppose we have a process that generates data as

$$y_i = w_0 + w_1 x_{i,1} + w_2 x_{i,2} + w_3 x_{i,1}^2 + w_4 x_{i,2}^2 + w_5 x_{i,1} x_{i,2} + \epsilon_i$$

where $\epsilon_i \sim N(0, \sigma^2)$. In other words:

$$? = [1, x_1, x_2, x_1^2, x_2^2, x_1 x_2]$$

Note that the model is *linear* in **w**.

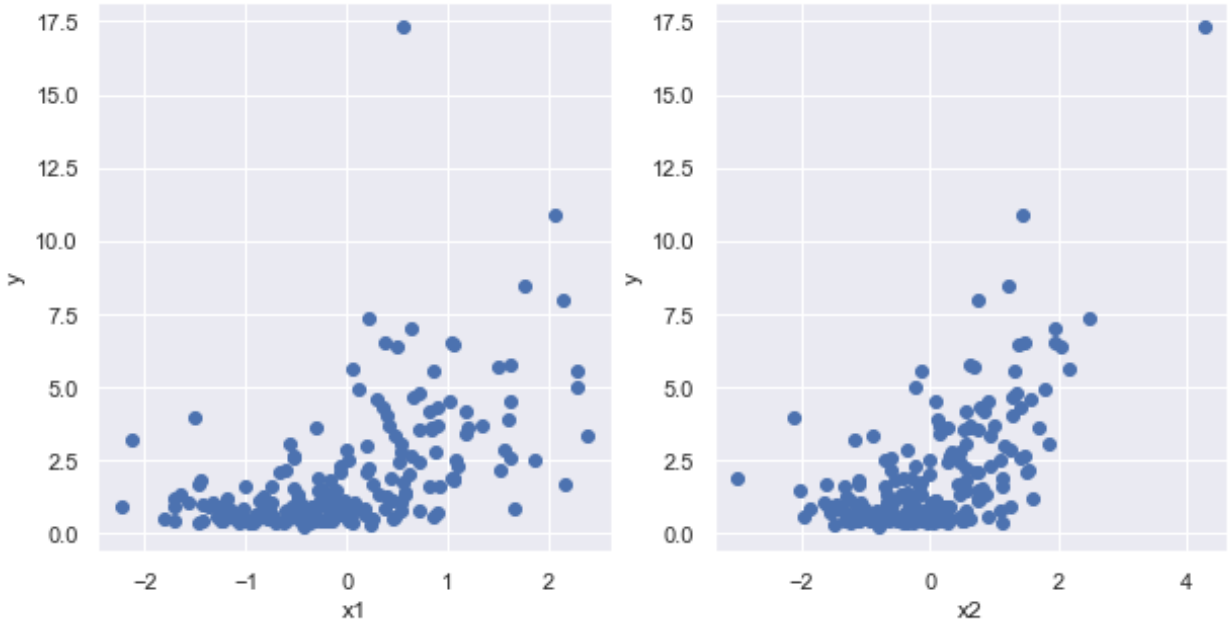
```
import itertools

def generate_linear_basis_data(n=200, d=2, coef=[1,1,0.5,0.5,1], intercept=1, sigma=0):
  x = np.random.randn(n,d)
  x = np.column_stack((x, x**2 ))
  for pair in list(itertools.combinations(range(d), 2)):
    x = np.column_stack((x, x[:,pair[0]]*x[:,pair[1]]))
  y = (np.dot(x, coef) + intercept).squeeze() + sigma * np.random.randn(n)
  return x[:,:d], y
```

```
x_train, y_train = generate_linear_basis_data(sigma=0.2)
x_test,  y_test  = generate_linear_basis_data(n=50, sigma=0.2)
```

```
print(x_train.shape)
print(y_train.shape)
```

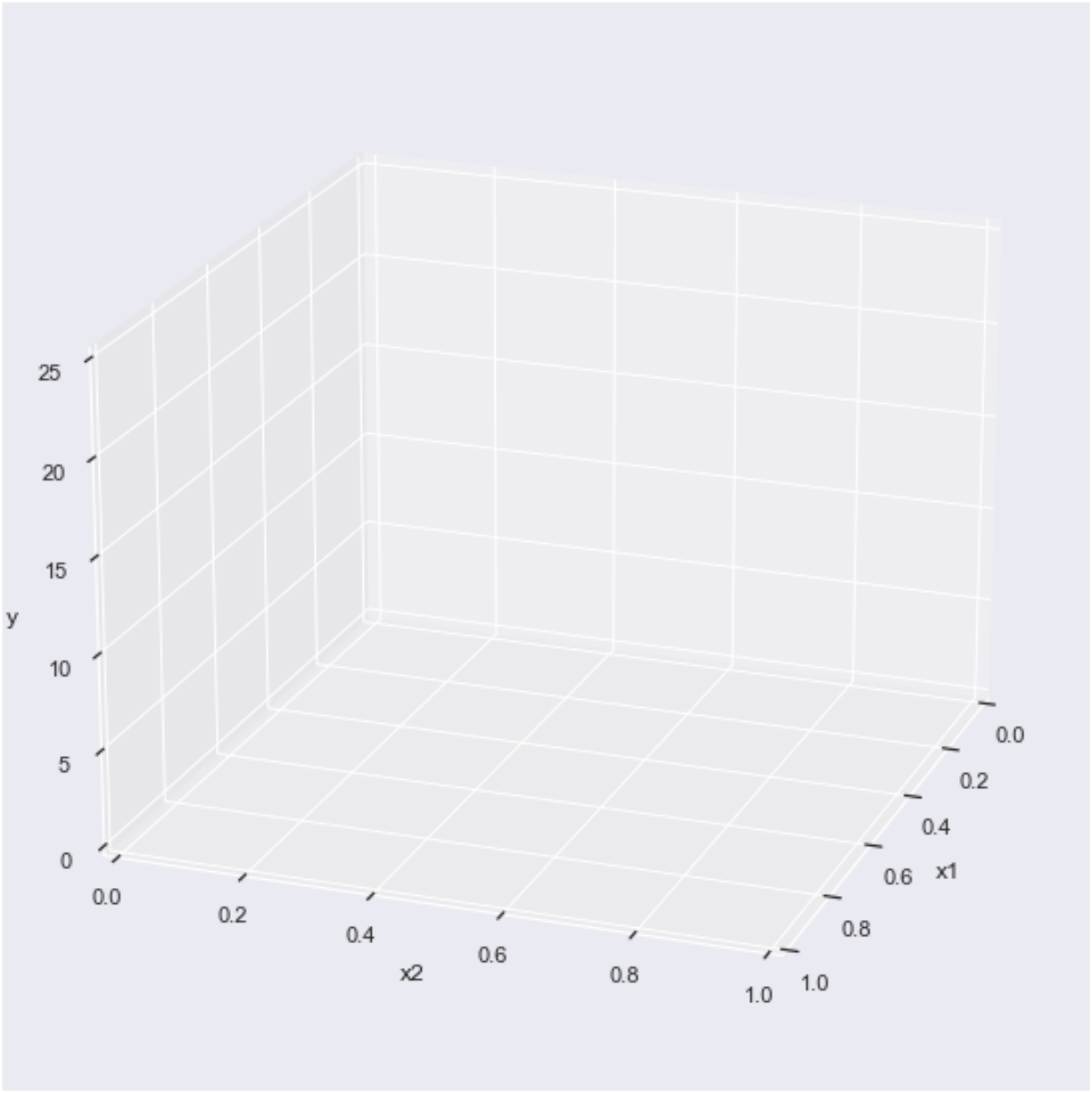
```
(200, 2)
(200,)
```

```
plt.figure(figsize=(10,5));
plt.subplot(1,2,1);
plt.scatter(x_train[:,0],  y_train);
plt.xlabel("x1");
plt.ylabel("y");
plt.subplot(1,2,2);
plt.scatter(x_train[:,1],  y_train);
plt.xlabel("x2");
plt.ylabel("y");
```




```
def plot_3D(elev, azim, w0, w1, w2, w3, w4, w5, show_sum, show_basis, show_data, X, y):
    plt.figure(figsize=(10,10))
    ax = plt.subplot(projection='3d')

    X1 = np.arange(-4, 4, 0.2)
    X2 = np.arange(-4, 4, 0.2)
    X1, X2 = np.meshgrid(X1, X2)
    Z0 = w0*np.ones(shape=(X1.shape[0], X2.shape[0]))
    Z1 = w1*X1
    Z2 = w2*X2
    Z3 = w3*X1**2
    Z4 = w4*X2**2
```

```
    Z5 = w5*X1*X2

    # Plot the surfaces.
    if show_basis:
      ax.plot_surface(X1, X2, Z0, alpha=0.1, color=sns.color_palette()[1], linewidth=0,
          antialiased=False)
      ax.plot_surface(X1, X2, Z1, alpha=0.1, color=sns.color_palette()[2], linewidth=0,
          antialiased=False)
      ax.plot_surface(X1, X2, Z2, alpha=0.1, color=sns.color_palette()[3], linewidth=0,
          antialiased=False)
      ax.plot_surface(X1, X2, Z3, alpha=0.1, color=sns.color_palette()[4], linewidth=0,
          antialiased=False)
      ax.plot_surface(X1, X2, Z4, alpha=0.1, color=sns.color_palette()[5], linewidth=0,
          antialiased=False)
      ax.plot_surface(X1, X2, Z5, alpha=0.1, color=sns.color_palette()[6], linewidth=0,
          antialiased=False)
    if show_sum:
      ax.plot_surface(X1, X2, (Z0+Z1+Z2+Z3+Z4+Z5), alpha=0.5, color='white', linewidth=0,
          antialiased=False)
    if show_data:
      ax.scatter3D(X[:, 0], X[:, 1], y, s=50)

    ax.view_init(elev=elev, azim=azim)
    ax.set_xlabel('x1')
    ax.set_ylabel('x2')
    ax.set_zlabel('y')
    ax.set_zlim(0, 25)

interact(plot_3D, elev=widgets.IntSlider(min=-90, max=90, step=10, value=20),
          azim=widgets.IntSlider(min=-90, max=90, step=10, value=20),
          w0 = widgets.FloatSlider(min=-1, max=2, step=0.1, value=1),
          w1 = widgets.FloatSlider(min=-1, max=2, step=0.1, value=1),
          w2 = widgets.FloatSlider(min=-1, max=2, step=0.1, value=1),
          w3 = widgets.FloatSlider(min=-1, max=2, step=0.1, value=0.5),
          w4 = widgets.FloatSlider(min=-1, max=2, step=0.1, value=0.5),
          w5 = widgets.FloatSlider(min=-1, max=2, step=0.1, value=1),
         show_sum = False, show_basis = False, show_data = False,
         X=fixed(x_train), y=fixed(y_train));
```



Fit a linear regression

```
reg_lbf = LinearRegression().fit(x_train, y_train)
print("Intercept: " , reg_lbf.intercept_)
print("Coefficient list: ", reg_lbf.coef_)
```

```
Intercept:  2.0131530109480855
Coefficient list:  [1.08829676 1.18752456]
```

Compute MSE and R2

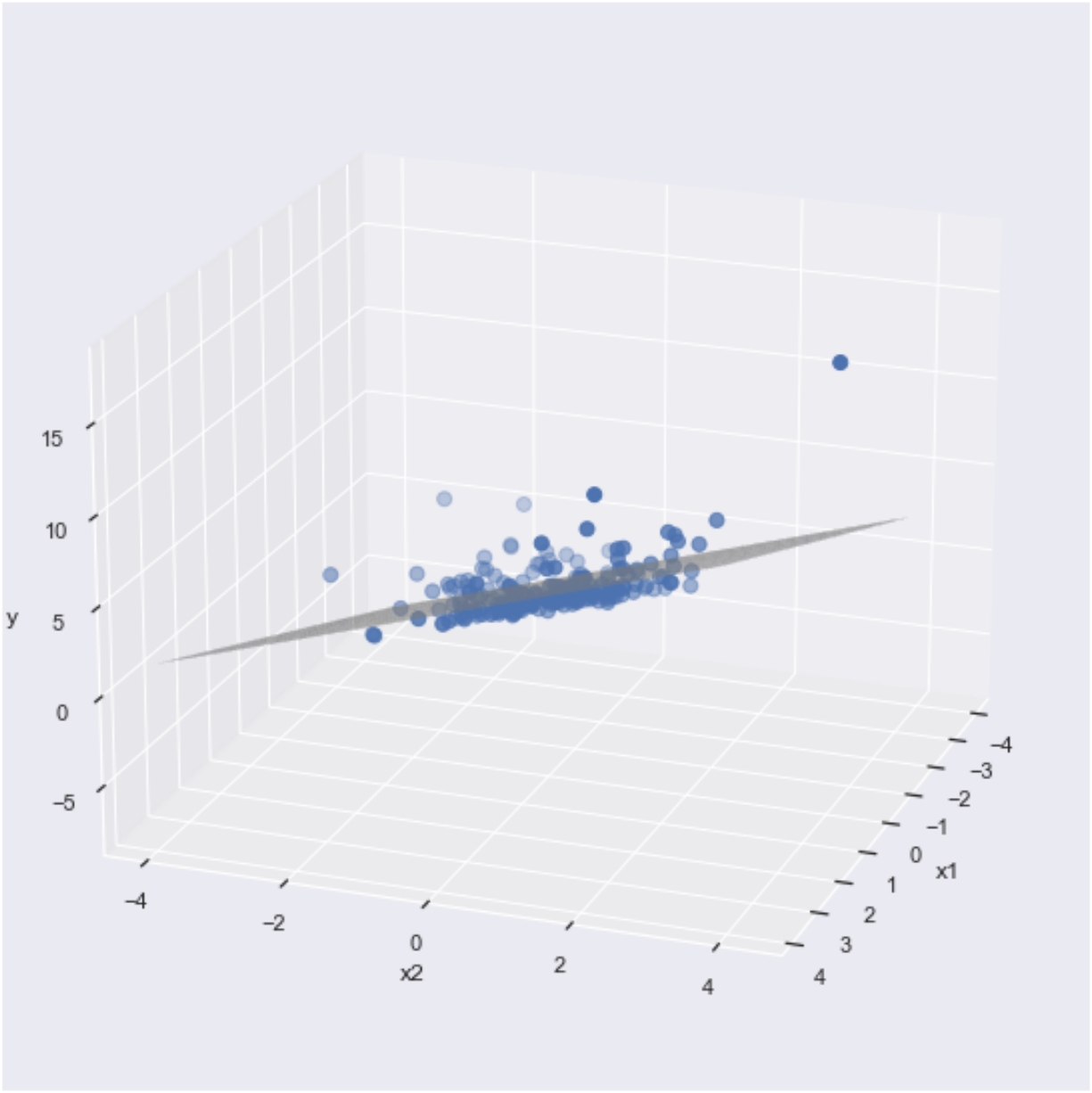
```
y_train_hat = reg_lbf.predict(x_train)
```

```
print("Training MSE: ", metrics.mean_squared_error(y_train, y_train_hat))
print("Training R2:  ", metrics.r2_score(y_train, y_train_hat))
```

```
Training MSE:  1.833682781181276
Training R2:   0.5920807396333971
```

Plot hyperplane

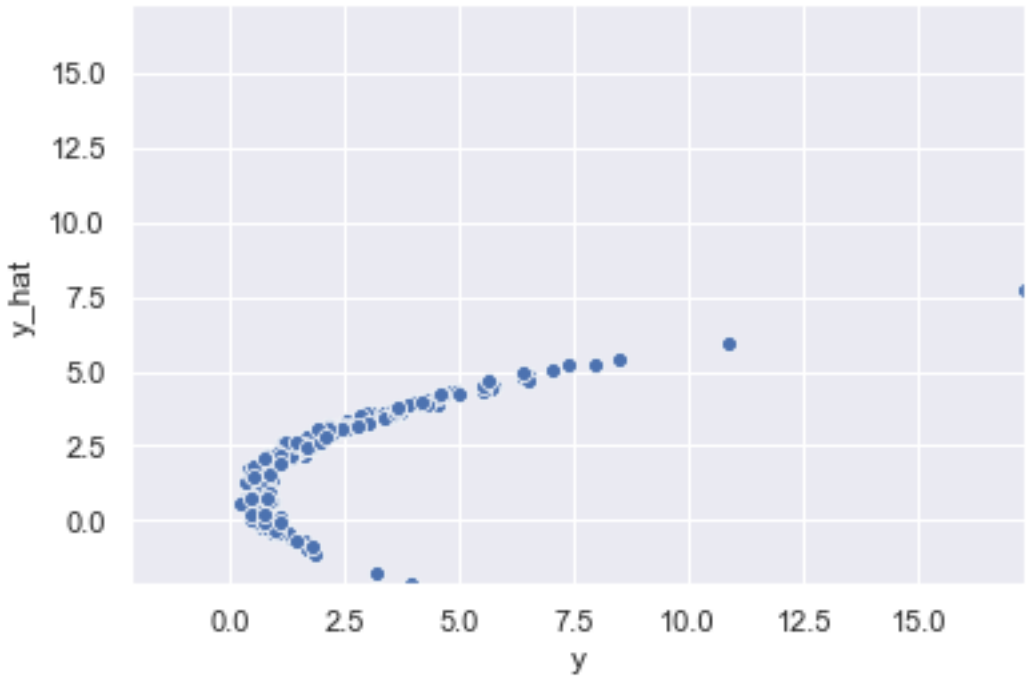
```
def plot_3D(elev=20, azim=-20, X=x_train, y=y_train):
    plt.figure(figsize=(10,10))
    ax = plt.subplot(projection='3d')


    X1 = np.arange(-4, 4, 0.2)
    X2 = np.arange(-4, 4, 0.2)
    X1, X2 = np.meshgrid(X1, X2)
    Z = X1*reg_lbf.coef_[0] + X2*reg_lbf.coef_[1] + reg_lbf.intercept_

    # Plot the surface.
    ax.plot_surface(X1, X2, Z, alpha=0.1, color='gray',
                          linewidth=0, antialiased=False)
    ax.scatter3D(X[:, 0], X[:, 1], y, s=50)

    ax.view_init(elev=elev, azim=azim)
    ax.set_xlabel('x1')
    ax.set_ylabel('x2')
    ax.set_zlabel('y')

interact(plot_3D, elev=widgets.IntSlider(min=-90, max=90, step=10, value=20),
          azim=widgets.IntSlider(min=-90, max=90, step=10, value=20),
         X=fixed(x_train), y=fixed(y_train));
```



Residual analysis

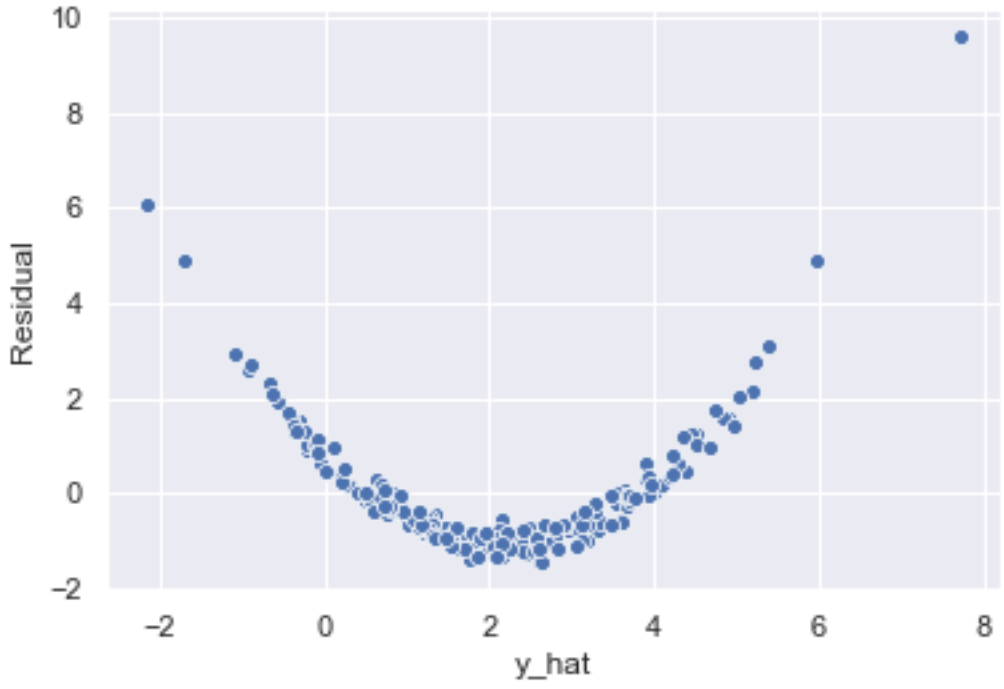
```
residual_train = y_train - y_train_hat
```

```
lim_hi = np.max(np.concatenate([y_train, y_train_hat]))
lim_lo = np.min(np.concatenate([y_train, y_train_hat]))
sns.scatterplot(x=y_train, y=y_train_hat);
plt.xlabel('y');
plt.ylabel('y_hat');
plt.xlim(lim_lo, lim_hi);
plt.ylim(lim_lo, lim_hi);
```



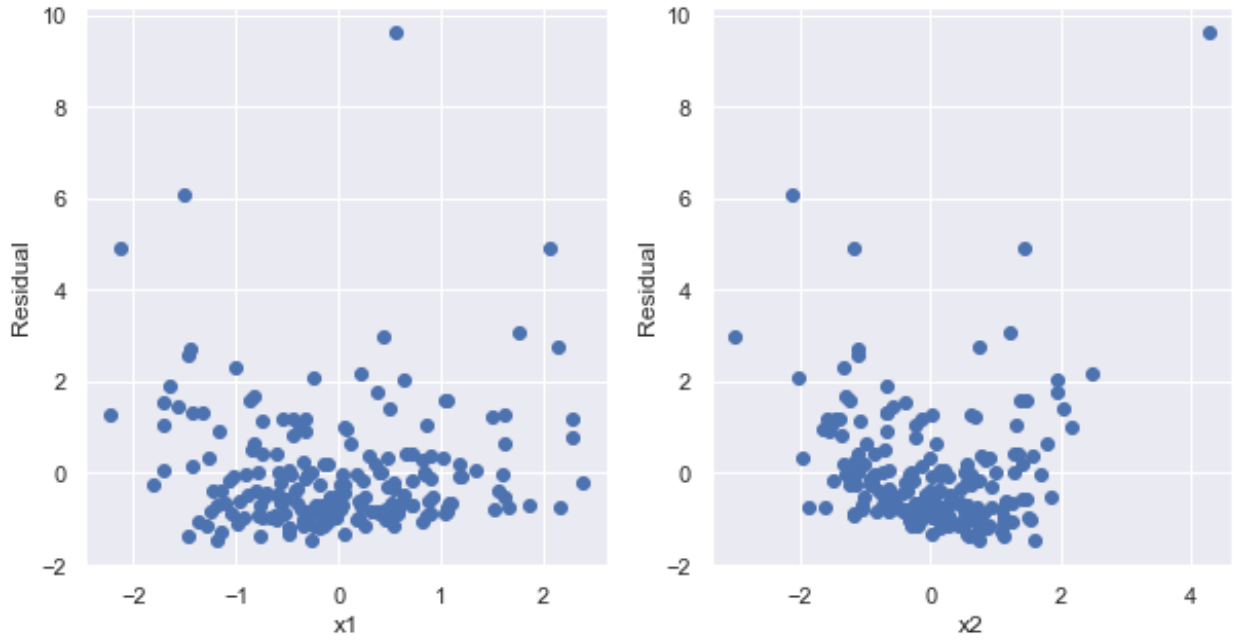
Is the error random? Or does it look systematic?

```
sns.scatterplot(x=y_train_hat, y=residual_train);
plt.xlabel('y_hat');
plt.ylabel('Residual');
```




```
plt.figure(figsize=(10,5));
plt.subplot(1,2,1);
```

```
plt.scatter(x_train[:,0],  residual_train);
plt.xlabel("x1");
plt.ylabel("Residual");
plt.subplot(1,2,2);
plt.scatter(x_train[:,1],  residual_train);
plt.xlabel("x2");
plt.ylabel("Residual");
```



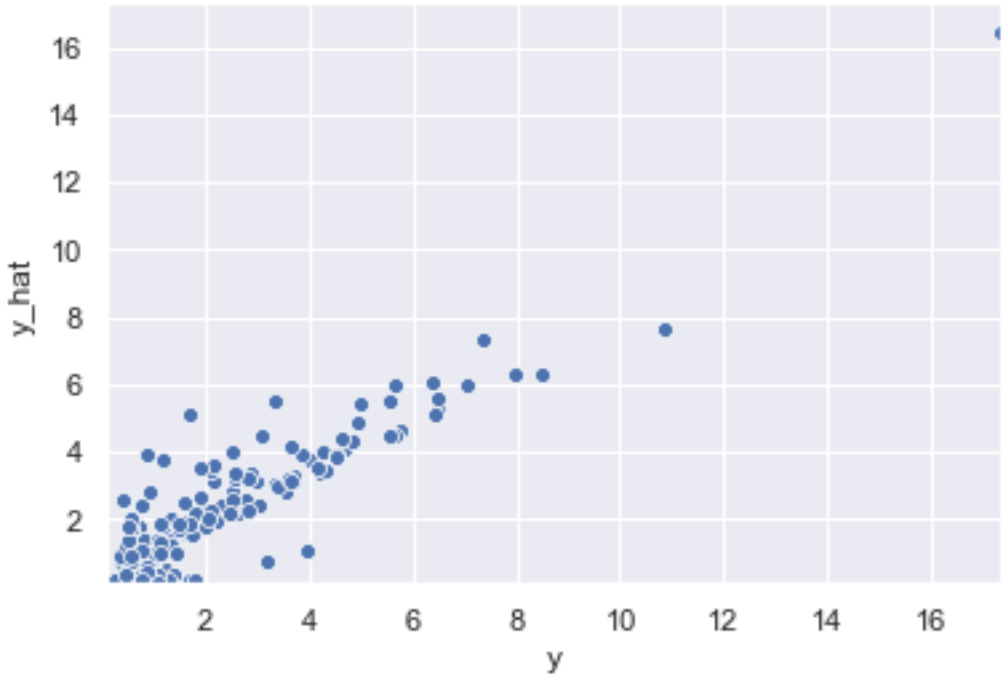
Since there is clearly some non-linearity, we can try to fit a model to a non-linear transformation of the features.

```
x_train_trans = np.column_stack((x_train, x_train**2))

reg_lbf_trans = LinearRegression().fit(x_train_trans, y_train)
print("Intercept: " , reg_lbf_trans.intercept_)
print("Coefficient list: ", reg_lbf_trans.coef_)

y_train_trans_hat = reg_lbf_trans.predict(x_train_trans)
print("Training MSE: ", metrics.mean_squared_error(y_train, y_train_trans_hat))
print("Training R2:  ", metrics.r2_score(y_train, y_train_trans_hat))

residual_train_trans = y_train - y_train_trans_hat
```

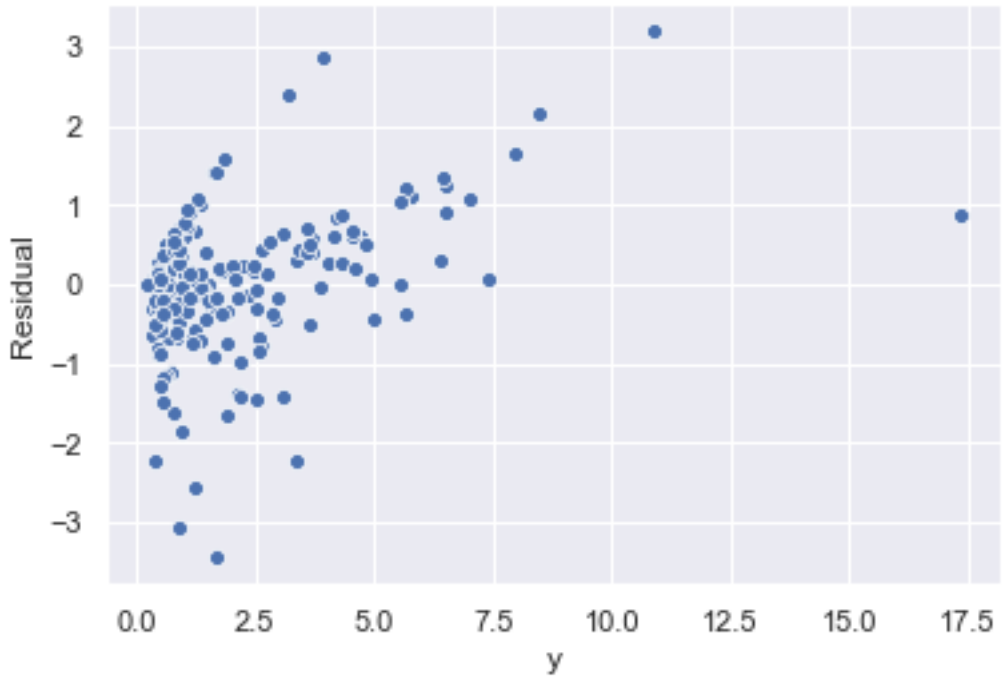
```
Intercept:  1.0158405134412498
Coefficient list:  [0.93368064 1.03797313 0.48762336 0.56514263]
Training MSE:  0.7152082980625866
Training R2:   0.8408954684267685
```

```
lim_hi = np.max(np.concatenate([y_train, y_train_trans_hat]))
lim_lo = np.min(np.concatenate([y_train, y_train_trans_hat]))
sns.scatterplot(x=y_train, y=y_train_trans_hat);
plt.xlabel('y');
```

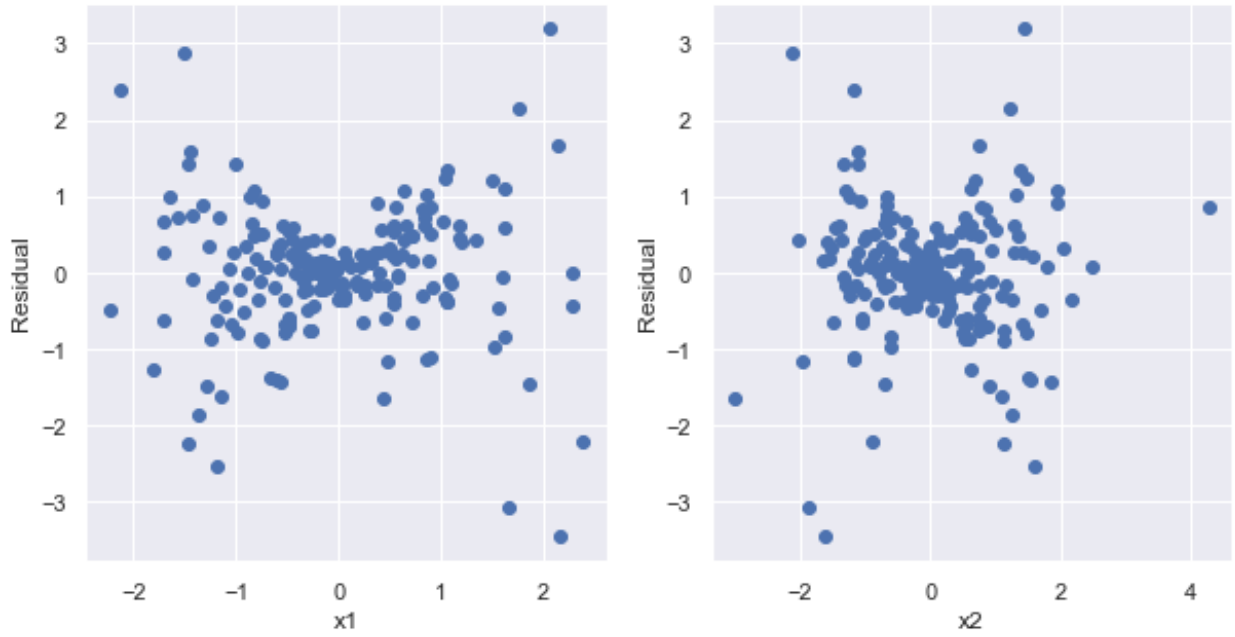
```
plt.ylabel('y_hat');
plt.xlim(lim_lo, lim_hi);
plt.ylim(lim_lo, lim_hi);
```




```
sns.scatterplot(x=y_train, y=residual_train_trans);
plt.xlabel('y');
plt.ylabel('Residual');
```



```
plt.figure(figsize=(10,5));
plt.subplot(1,2,1);
plt.scatter(x_train[:,0],  residual_train_trans);
plt.xlabel("x1");
plt.ylabel("Residual");
plt.subplot(1,2,2);
plt.scatter(x_train[:,1],  residual_train_trans);
plt.xlabel("x2");
plt.ylabel("Residual");
```

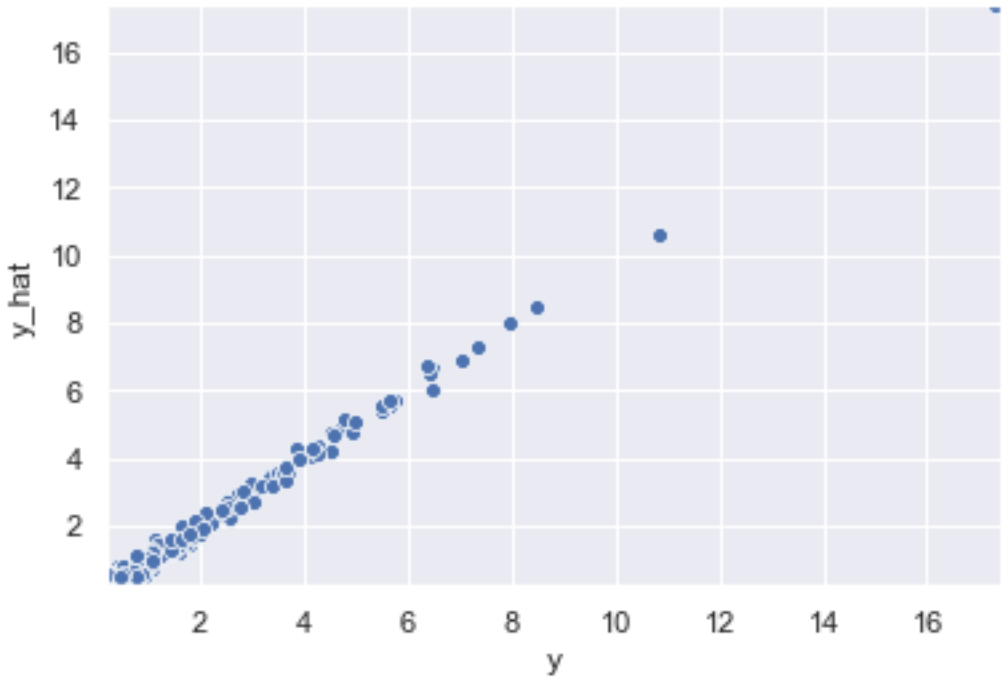



```
x_train_inter = np.column_stack((x_train_trans, x_train[:,0]*x_train[:,1]))

reg_lbf_inter = LinearRegression().fit(x_train_inter, y_train)
print("Intercept: " , reg_lbf_inter.intercept_)
print("Coefficient list: ", reg_lbf_inter.coef_)

y_train_inter_hat = reg_lbf_inter.predict(x_train_inter)
print("Training MSE: ", metrics.mean_squared_error(y_train, y_train_inter_hat))
print("Training R2:  ", metrics.r2_score(y_train, y_train_inter_hat))

residual_train_inter = y_train - y_train_inter_hat
```

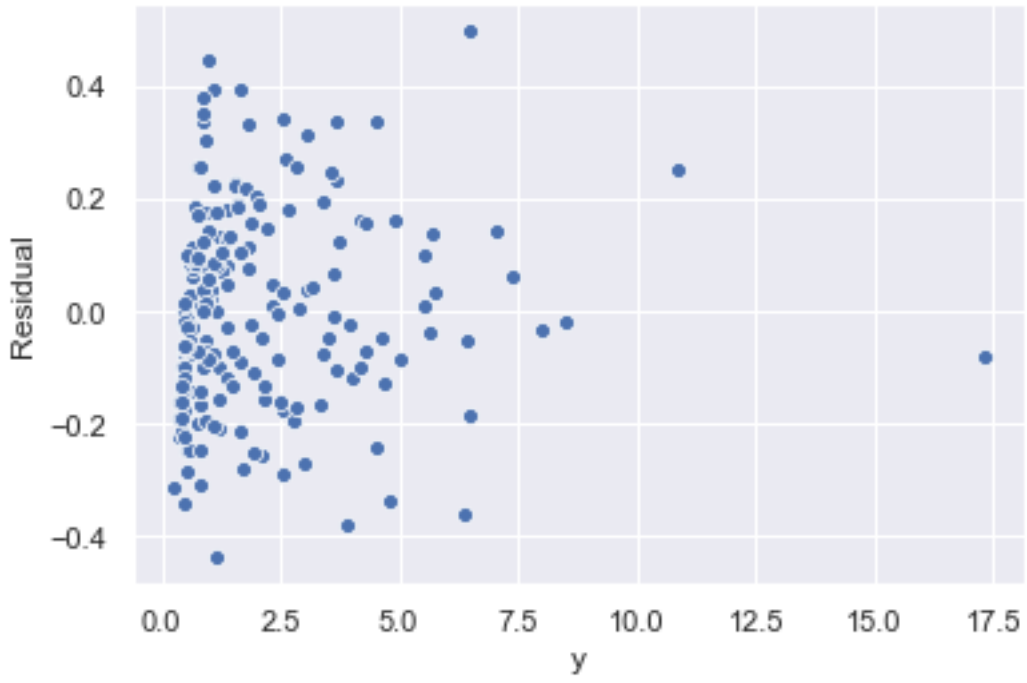
```
Intercept:  1.0156225345987617
Coefficient list:  [1.01990112 0.98477914 0.48269725 0.49270588 1.0221473 ]
Training MSE:  0.03134938550935531
Training R2:  0.993026046663488
```

```
lim_hi = np.max(np.concatenate([y_train, y_train_inter_hat]))
lim_lo = np.min(np.concatenate([y_train, y_train_inter_hat]))
sns.scatterplot(x=y_train, y=y_train_inter_hat);
plt.xlabel('y');
```

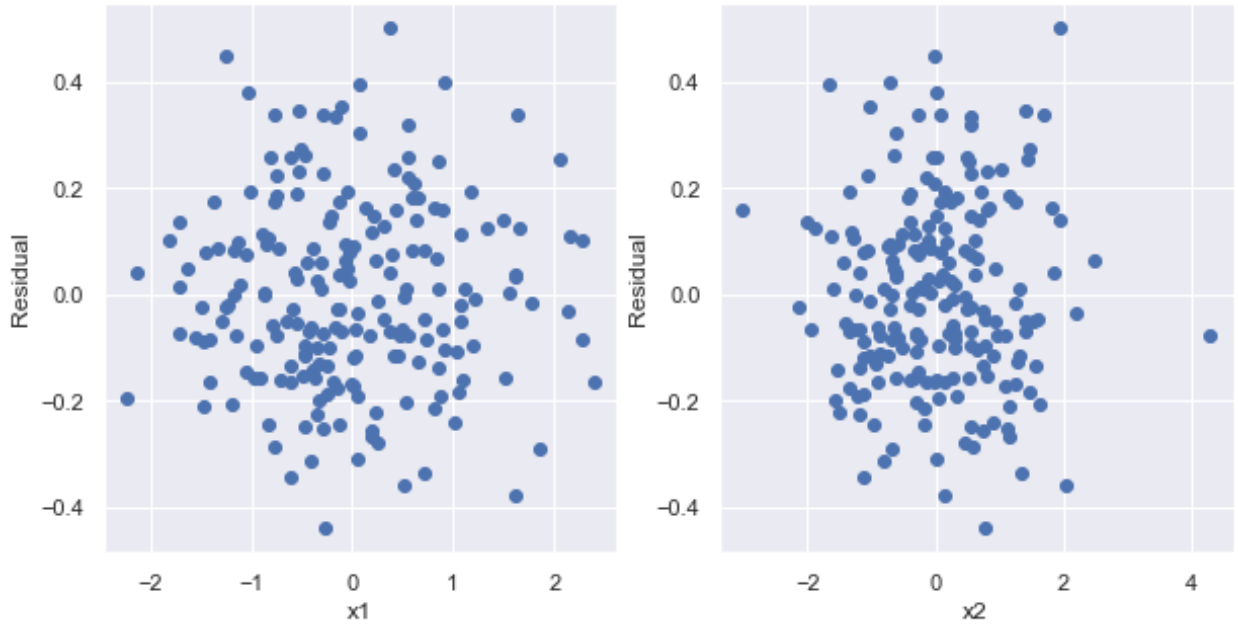
```
plt.ylabel('y_hat');
plt.xlim(lim_lo, lim_hi);
plt.ylim(lim_lo, lim_hi);
```




```
sns.scatterplot(x=y_train, y=residual_train_inter);
plt.xlabel('y');
plt.ylabel('Residual');
```



```
plt.figure(figsize=(10,5));
plt.subplot(1,2,1);
plt.scatter(x_train[:,0],  residual_train_inter);
plt.xlabel("x1");
plt.ylabel("Residual");
plt.subplot(1,2,2);
plt.scatter(x_train[:,1],  residual_train_inter);
plt.xlabel("x2");
plt.ylabel("Residual");
```



Evaluate on test set

```
y_train_hat = reg_lbf_inter.predict(x_train_inter)
print("Training MSE: ", metrics.mean_squared_error(y_train, y_train_hat))
print("Training R2:  ", metrics.r2_score(y_train, y_train_hat))
```

```
Training MSE:  0.03134938550935531
Training R2:   0.993026046663488
```

```
x_test_inter = np.column_stack((x_test, x_test**2))
x_test_inter = np.column_stack((x_test_inter, x_test[:,0]*x_test[:,1]))

y_test_hat = reg_lbf_inter.predict(x_test_inter)
print("Test MSE: ", metrics.mean_squared_error(y_test, y_test_hat))
print("Test R2:  ", metrics.r2_score(y_test, y_test_hat))
```

```
Test MSE:  0.048169360850192966
Test R2:   0.9812295205840841
```