linear_regression

January 11, 2022

0.1 Linear regression workbook: Sunay Bhat - W2022

This workbook will walk you through a linear regression example. It will provide familiarity with Jupyter Notebook and Python. Please print (to pdf) a completed version of this workbook for submission with HW #1.

ECE C147/C247 Winter Quarter 2022, Prof. J.C. Kao, TAs Y. Li, P. Lu, T. Monsoor, T. wang

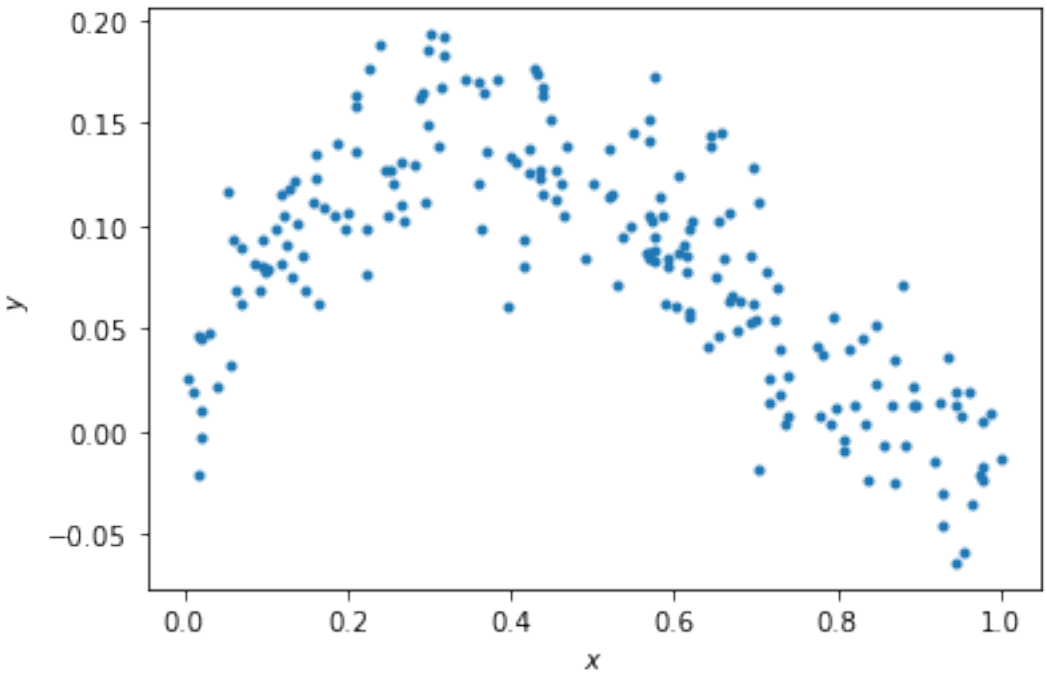
[1]:
```
import numpy as np
import matplotlib.pyplot as plt

#allows matlab plots to be generated in line
%matplotlib inline
```

0.1.1 Data generation

For any example, we first have to generate some appropriate data to use. The following cell generates data according to the model: $y = x - 2x^2 + x^3 + \epsilon$

[2]:
```
np.random.seed(0)  # Sets the random seed.
num_train = 200      # Number of training data points

# Generate the training data
x = np.random.uniform(low=0, high=1, size=(num_train,))
y = x - 2*x**2 + x**3 + np.random.normal(loc=0, scale=0.03, size=(num_train,))
f = plt.figure()
ax = f.gca()
ax.plot(x, y, '.')
ax.set_xlabel('$x$')
ax.set_ylabel('$y$')
```

[2]:
```
Text(0, 0.5, '$y$')
```



0.1.2 QUESTIONS:

Write your answers in the markdown cell below this one:

(1) What is the generating distribution of x?

(2) What is the distribution of the additive noise ϵ?

0.1.3 ANSWERS:

(1) $U(0, 1)$ - Uniform distribution with parameters 0 and 1.

(2) $N(0, 0.3)$ - Normal distirbution with parameters mean $\mu = 0$ and standard deviation $\sigma = 0.3$.

0.1.4 Fitting data to the model (5 points)

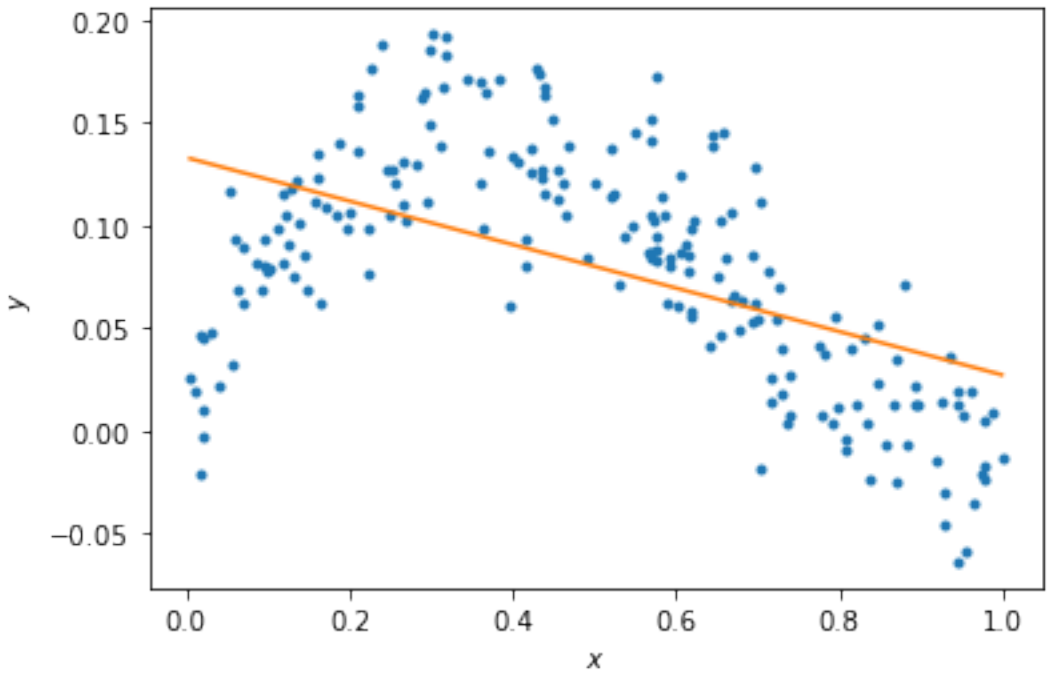
Here, we'll do linear regression to fit the parameters of a model $y = ax + b$.

[3]:
```
# xhat = (x, 1)
xhat = np.vstack((x, np.ones_like(x)))

# ==================== #
# START YOUR CODE HERE #
# ==================== #
# GOAL: create a variable theta; theta is a numpy array whose elements are [a,␣
 ↪b]
```

```
# Normal Equation: (X'*X)^-1 * X'*y
theta = np.linalg.inv(xhat.dot(xhat.T)).dot(xhat.dot(y))

# ================== #
# END YOUR CODE HERE #
# ================== #
```

[4]:
```
# Plot the data and your model fit.
f = plt.figure()
ax = f.gca()
ax.plot(x, y, '.')
ax.set_xlabel('$x$')
ax.set_ylabel('$y$')

# Plot the regression line
xs = np.linspace(min(x), max(x),50)
xs = np.vstack((xs, np.ones_like(xs)))
plt.plot(xs[0,:], theta.dot(xs))
```

[4]: [<matplotlib.lines.Line2D at 0x7fb9242b5eb0>]



0.1.5 QUESTIONS

(1) Does the linear model under- or overfit the data?

(2) How to change the model to improve the fitting?

0.1.6 ANSWERS

(1) The linear model **underfits** the data.

(2) We can use a **higher order polynomial** like a 2nd or 3rd order polynomial or many other models with **higher complexity**.

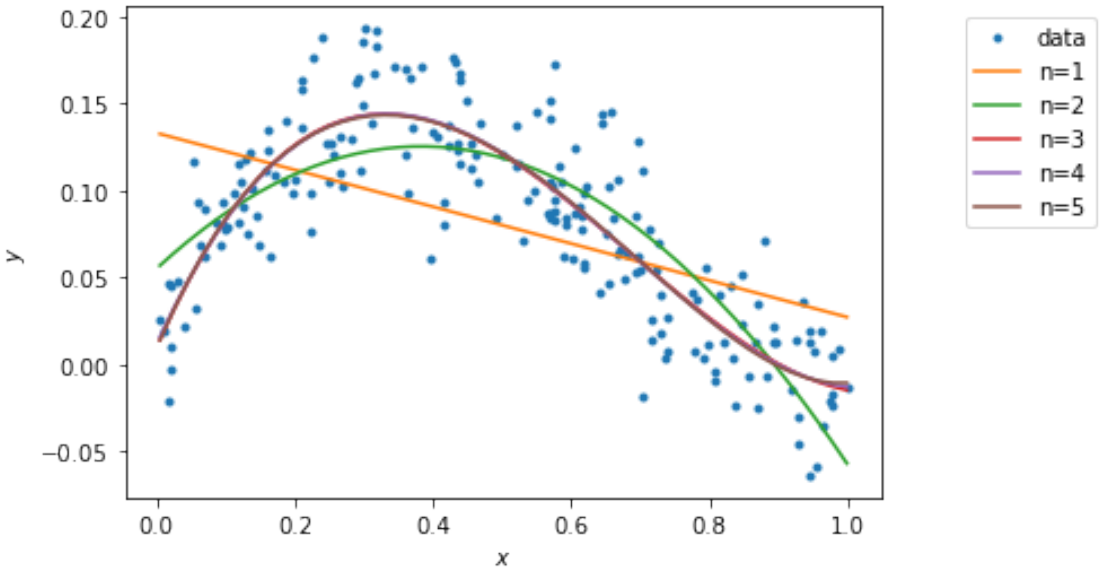
0.1.7 Fitting data to the model (10 points)

Here, we'll now do regression to polynomial models of orders 1 to 5. Note, the order 1 model is the linear model you prior fit.

[5]:
```
N = 5
xhats = []
thetas = []

# ==================== #
# START YOUR CODE HERE #
# ==================== #

# GOAL: create a variable thetas.
# thetas is a list, where theta[i] are the model parameters for the polynomial␣
 ↪fit of order i+1.
#   i.e., thetas[0] is equivalent to theta above.
#   i.e., thetas[1] should be a length 3 np.array with the coefficients of the␣
 ↪x^2, x, and 1 respectively.
#   ... etc.

xhats.append(np.vstack((x, np.ones_like(x)))) # Inital xhat = = (x, 1)
for order in np.arange(N):
    if order > 0:
        xhats.append(np.vstack((x**(order+1), xhats[order-1]))) #add next order␣
 ↪features to last xhat and append

    # Normal Equation: (X'*X)^-1 * X'*y
    thetas.append(np.linalg.inv(xhats[order].dot(xhats[order].T)).
 ↪dot(xhats[order].dot(y)))


# ================== #
# END YOUR CODE HERE #
# ================== #
```

[6]:
```
# Plot the data
f = plt.figure()
ax = f.gca()
ax.plot(x, y, '.')
ax.set_xlabel('$x$')
ax.set_ylabel('$y$')
```

```
# Plot the regression lines
plot_xs = []
for i in np.arange(N):
    if i == 0:
        plot_x = np.vstack((np.linspace(min(x), max(x),50), np.ones(50)))
    else:
        plot_x = np.vstack((plot_x[-2]**(i+1), plot_x))
    plot_xs.append(plot_x)

for i in np.arange(N):
    ax.plot(plot_xs[i][-2,:], thetas[i].dot(plot_xs[i]))

labels = ['data']
[labels.append('n={}'.format(i+1)) for i in np.arange(N)]
bbox_to_anchor=(1.3, 1)
lgd = ax.legend(labels, bbox_to_anchor=bbox_to_anchor)
```



0.1.8 Calculating the training error (10 points)

Here, we'll now calculate the training error of polynomial models of orders 1 to 5:

$L(\theta) = \frac{1}{2}\sum_j(\hat{y}_j - y_j)^2$

[7]:
```
training_errors = []

# ==================== #
# START YOUR CODE HERE #
```

```
# ==================== #

# GOAL: create a variable training_errors, a list of 5 elements,
# where training_errors[i] are the training loss for the polynomial fit of␣
 ↪order i+1.

for order in np.arange(N):
    # 1/2 * (theta*xhat - y)' * (theta*xhat - y)'
    errors = thetas[order] @ xhats[order] - y
    training_errors.append(1/2 * (errors.T @ errors))

# ================== #
# END YOUR CODE HERE #
# ================== #

print ('Training errors are: \n', training_errors)
```

```
Training errors are:
 [0.2379961088362701, 0.10924922209268528, 0.08169603801105374,
0.08165353735296982, 0.08161479195525295]
```

0.1.9 QUESTIONS

(1) Which polynomial model has the best training error?

(2) Why is this expected?

0.1.10 ANSWERS

(1) The **5th order** polynomial has the lowest training error.

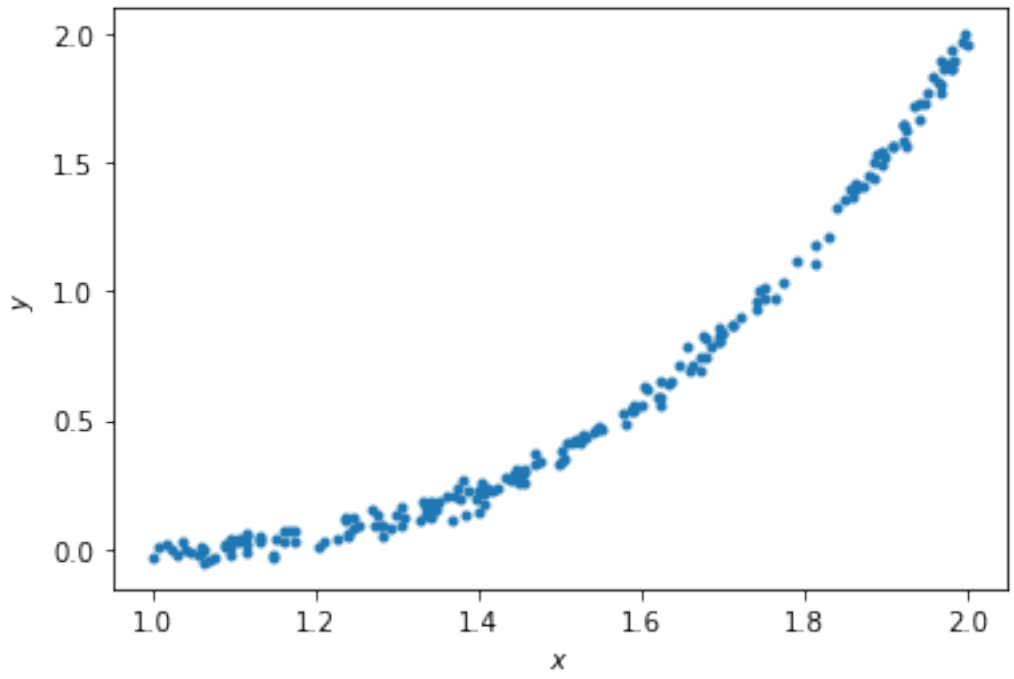
(2) Yes, higher order polynomials will have <= training error than lower order. You can do as well as lower order polynomials by setting higher coefficients to zero (and others equal to each other), but you can generally utilize the added degrees of freedom from higher order terms to better fit the training data.

0.1.11 Generating new samples and validation error (5 points)

Here, we'll now generate new samples and calculate the validation error of polynomial models of orders 1 to 5.

[8]:
```
x = np.random.uniform(low=1, high=2, size=(num_train,))
y = x - 2*x**2 + x**3 + np.random.normal(loc=0, scale=0.03, size=(num_train,))
f = plt.figure()
ax = f.gca()
ax.plot(x, y, '.')
ax.set_xlabel('$x$')
ax.set_ylabel('$y$')
```

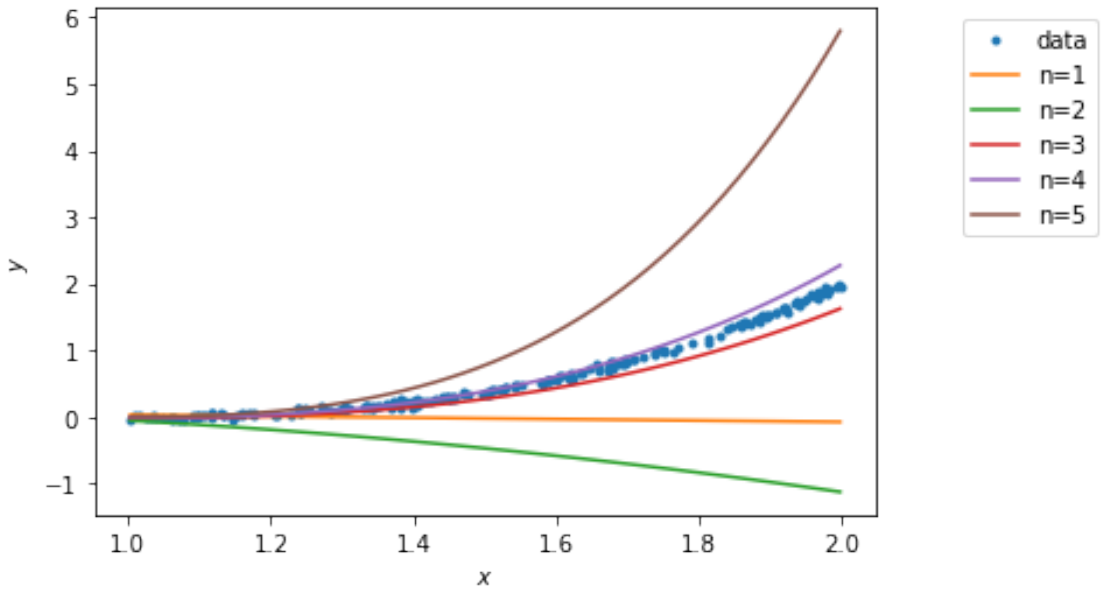
[8]:
```
Text(0, 0.5, '$y$')
```



[9]:
```
xhats = []
for i in np.arange(N):
    if i == 0:
        xhat = np.vstack((x, np.ones_like(x)))
        plot_x = np.vstack((np.linspace(min(x), max(x),50), np.ones(50)))
    else:
        xhat = np.vstack((x**(i+1), xhat))
        plot_x = np.vstack((plot_x[-2]**(i+1), plot_x))

    xhats.append(xhat)
```

[10]:
```
# Plot the data
f = plt.figure()
ax = f.gca()
ax.plot(x, y, '.')
ax.set_xlabel('$x$')
ax.set_ylabel('$y$')

# Plot the regression lines
plot_xs = []
for i in np.arange(N):
    if i == 0:
        plot_x = np.vstack((np.linspace(min(x), max(x),50), np.ones(50)))
    else:
```

```
        plot_x = np.vstack((plot_x[-2]**(i+1), plot_x))
    plot_xs.append(plot_x)

for i in np.arange(N):
    ax.plot(plot_xs[i][-2,:], thetas[i].dot(plot_xs[i]))

labels = ['data']
[labels.append('n={}'.format(i+1)) for i in np.arange(N)]
bbox_to_anchor=(1.3, 1)
lgd = ax.legend(labels, bbox_to_anchor=bbox_to_anchor)
```



[11]:
```
validation_errors = []

# ==================== #
# START YOUR CODE HERE #
# ==================== #

# GOAL: create a variable validation_errors, a list of 5 elements,
# where validation_errors[i] are the validation loss for the polynomial fit of␣
 ↪order i+1.

for order in np.arange(N):
    # 1/2 * (theta*xhat - y)' * (theta*xhat - y)'
    errors = thetas[order] @ xhats[order] - y
    validation_errors.append(1/2 * (errors.T @ errors))

# ================== #
```

```
# END YOUR CODE HERE #
# ================== #

print ('Validation errors are: \n', validation_errors)
```

```
Validation errors are:
 [80.86165184550586, 213.19192445057894, 3.125697108276393, 1.1870765189474703,
214.91021817652626]
```

0.1.12 QUESTIONS

(1) Which polynomial model has the best validation error?

(2) Why does the order-5 polynomial model not generalize well?

0.1.13 ANSWERS

(1) The **4th order** polynomial has the best, or lowest, validation error, with the 3rd order being close.

(2) The 5th order model **overfits the data and does not generalize well when we validate our dataset in the previously unseen region of** $x = [1 : 2]$. In this case, the problem is made acute by validating in an extension of the domain ($x = [1 : 2]$) instead of new points from within the training data domain ($x = [0, 1]$). Specifically, the 4th order coefficient for the 5th order model is too large and rapidly diverges from the 3rd order generative model in the new domain space. The 4th order model's 4th order coefficient is much smaller though, allowing it to perform well even though the generative model is only 3rd order.

[ ]: