Linear regression workbook

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 2024, Prof. J.C. Kao, TAs: T.Monsoor, Y. Liu, S. Rajesh, L. Julakanti, K. Pang

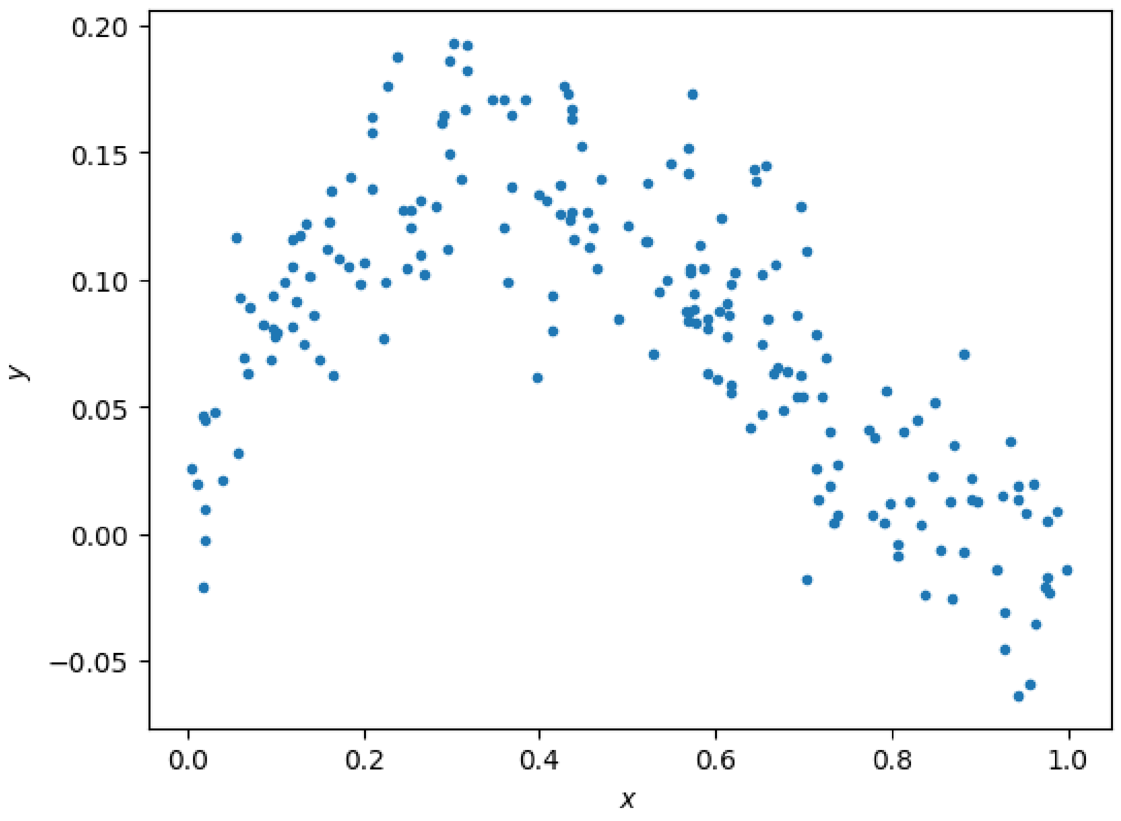
```
import numpy as np
import matplotlib.pyplot as plt

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

Data generation

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

```
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('$y$')
```

```
Text(0, 0.5, '$y$')
```



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?

ANSWERS:

(1) The code above generates a uniform distribution, as seen in the above graph and from the average number of datapoints. According to the code, we used np.random.uniform indicating the uniform distribution between the values 0 and 1.

(2) The additive noise generated has a normal distribution, with a mean of 0 and a standard deviation of 0.3. The np.random library adds this noise to the y output, but the general distribution of x on y is x - 2*x^2 + x^3

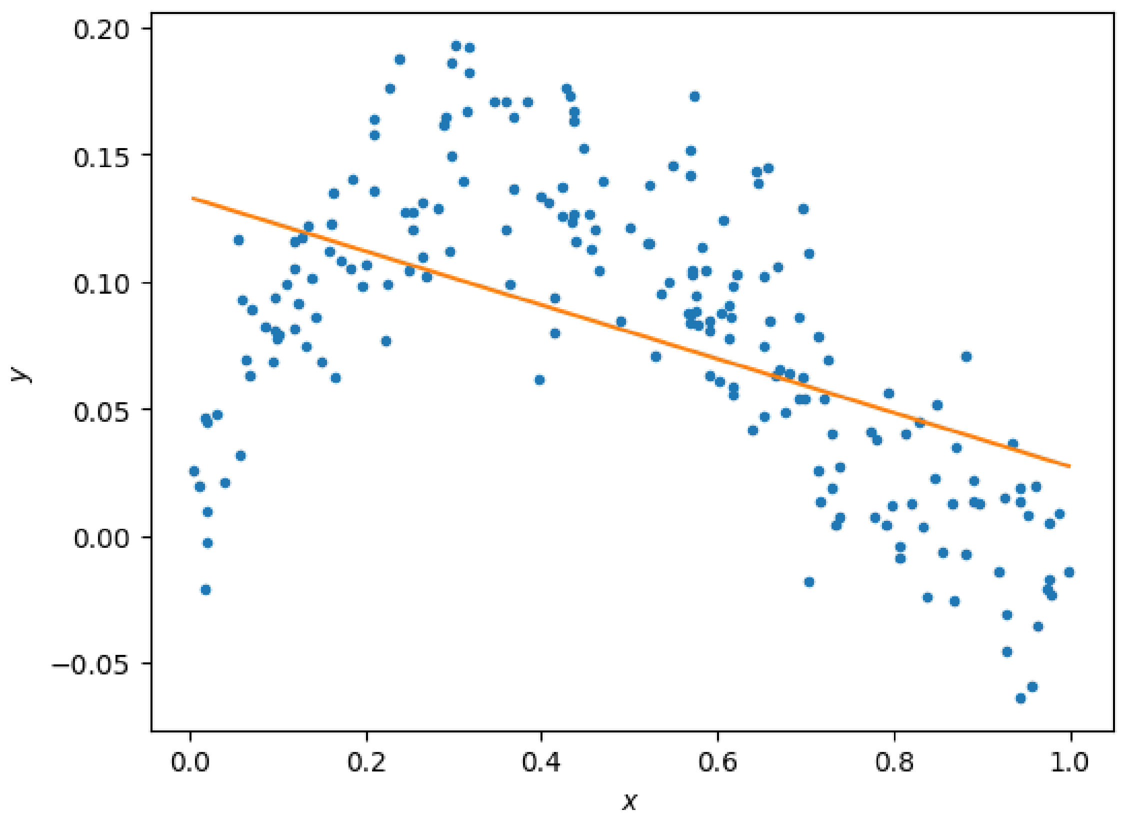
Fitting data to the model (5 points)

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

```
# 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]

#theta = np.linalg.inv(xhat.T.dot(xhat)).dot(xhat.T.dot(y))# please
modify this line
theta = np.linalg.inv((xhat.T).T.dot(xhat.T)).dot((xhat.T).T.dot(y))
# =================== #
# END YOUR CODE HERE #
# =================== #

# 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))
```

```
[<matplotlib.lines.Line2D at 0x117fb53d0>]
```



QUESTIONS

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

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

ANSWERS

(1) The model underfits the data, which is evident from the general distribution of the data points. The data looks like a polynomial, however by predicted it with a linear regression model, we get a line that doesn't capture the shape of the data points.

(2) We could increase the complexity of the model by including more polynomial terms in order to minimize the loss and better fit the data. This can be done by increasing the number of terms in theta to fit the number of the polynomial terms we want, and add more terms to X indicating the power of these polynomials.

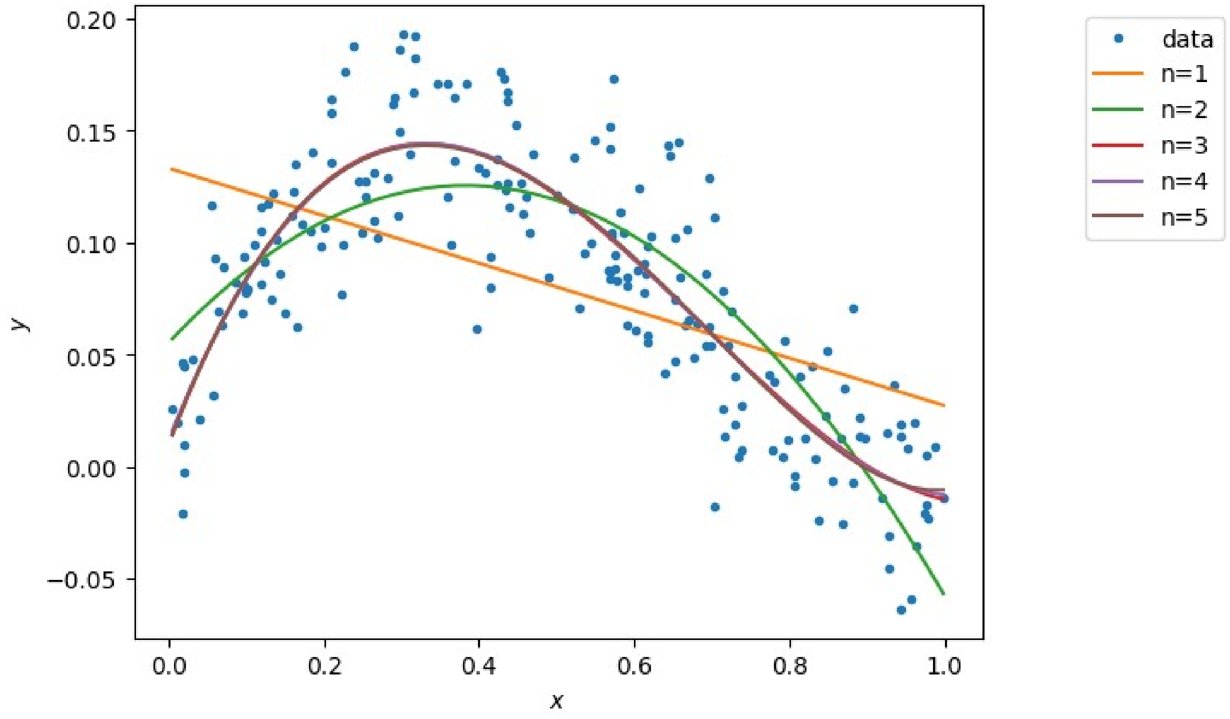
Fitting data to the model (5 points)

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.

```
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.
for i in range(1, N+1):
    xhat = np.ones_like(x)
    for j in range(1, i+1):
        xhat = np.vstack((x**j, xhat))
    x_trans = xhat.T
    theta =
np.linalg.inv(x_trans.T.dot(x_trans)).dot(x_trans.T.dot(y))
    xhats.append(x_trans)
    thetas.append(theta)
#theta = np.linalg.inv((xhat.T).T.dot(xhat.T)).dot((xhat.T).T.dot(y))
# ================== #

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

# 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)
```



Calculating the training error (5 points)

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

```
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 i in range(N):
    difference_val = y - xhats[i].dot(thetas[i])
    training_errors.append(difference_val.T.dot(difference_val)
/num_train)

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

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

```
Training errors are:
 [0.0023799610883627003, 0.0010924922209268528, 0.0008169603801105375,
0.000816535373529698, 0.0008161479195525295]
```

QUESTIONS

(1) What polynomial has the best training error?

(2) Why is this expected?

ANSWERS

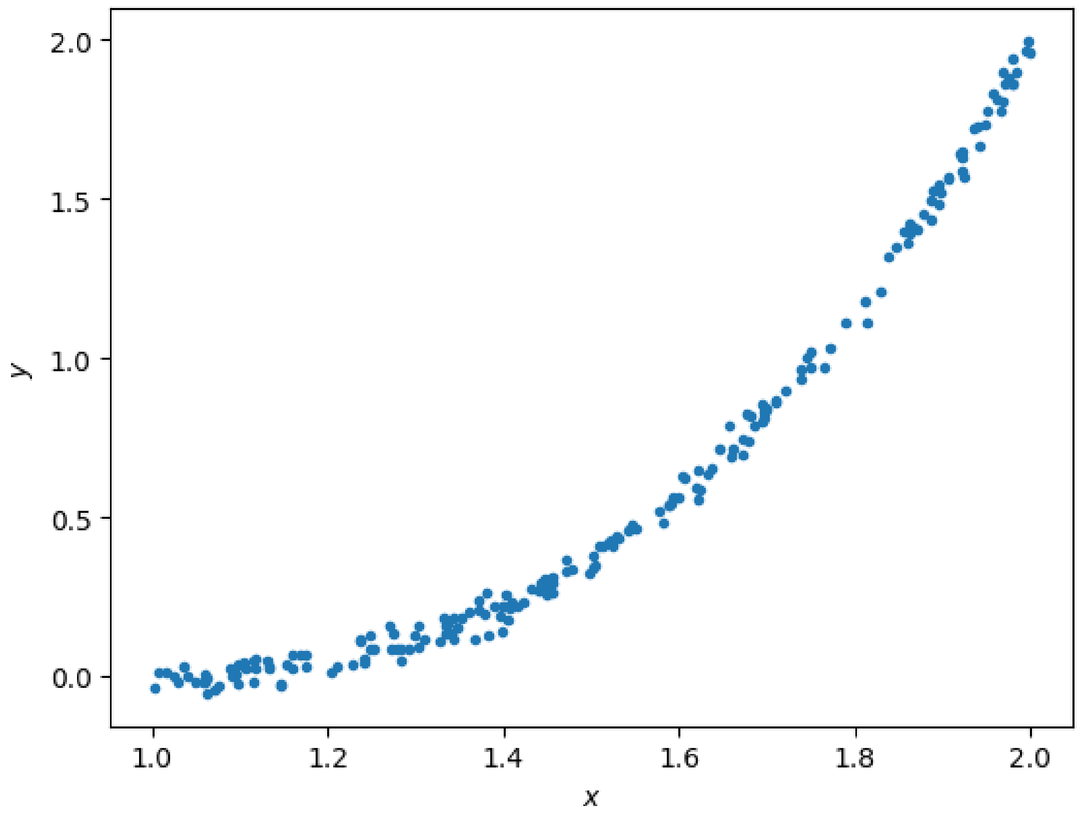
(1) The polynomial model of the order 5 has the lowest training error, of 0.0008161479195525295 compared to every other lower order polynomial

(2) This is because the compared to a previous (low order)polynomial, the higher order polynomial model has an extra order of x that can be used to reduce the loss. Consider the polynomial f(x) = bx**(n-1) + cx**(n--1).... C. The next polynomial could have the same constants as the previous model and will have the same loss, if g(x) = ax**(n) + bx**(n-1) + cx**(n--1).... C is the higher order polynomial and the constant 'a' is zero. However now this extra term can be used to drive the loss even further down

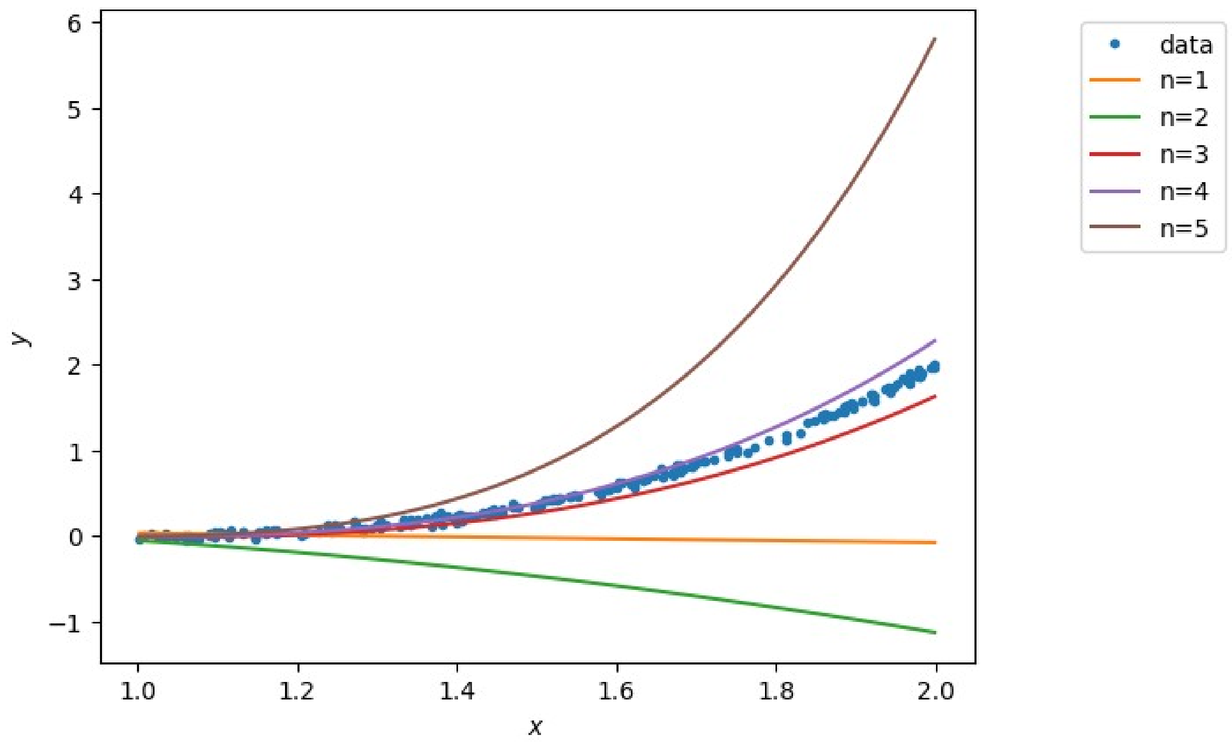
Generating new samples and testing error (5 points)

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

```
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('$y$')

Text(0, 0.5, '$y$')
```



```
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)

# 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)
```



```
testing_errors = []

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

# GOAL: create a variable testing_errors, a list of 5 elements,
# where testing_errors[i] are the testing loss for the polynomial fit
of order i+1.
for i in range(N):
    difference_val = y- xhats[i].T.dot(thetas[i])
```

```
testing_errors.append((difference_val.T.dot(difference_val))/num_train
)
# ================== #
# END YOUR CODE HERE #
# ================== #

print ('Testing errors are: \n', testing_errors)
```

```
Testing errors are:
 [0.8086165184550592, 2.1319192445058492, 0.03125697108312313,
0.01187076519554373, 2.1491021831759682]
```

QUESTIONS

(1) What polynomial has the best testing error?

(2) Why polynomial models of orders 5 does not generalize well?

ANSWERS

(1) The model with order 4 is the best fitting polynomial model. It has a testing loss of 0.01187076519554373 on the unseen test data which is the lowest out of all the polynomial models.

(2) The polynomial order of 5 has the highest testing error, and this is due to the overfitting of the training data. This is because it overfits the training data, however it doesn't generalize well on the unseen data, thus leading to higher training errors. The polynomial of order 5 tries to fit the noise in the training data, and due to this it overfits and doesn't generalize the data well.