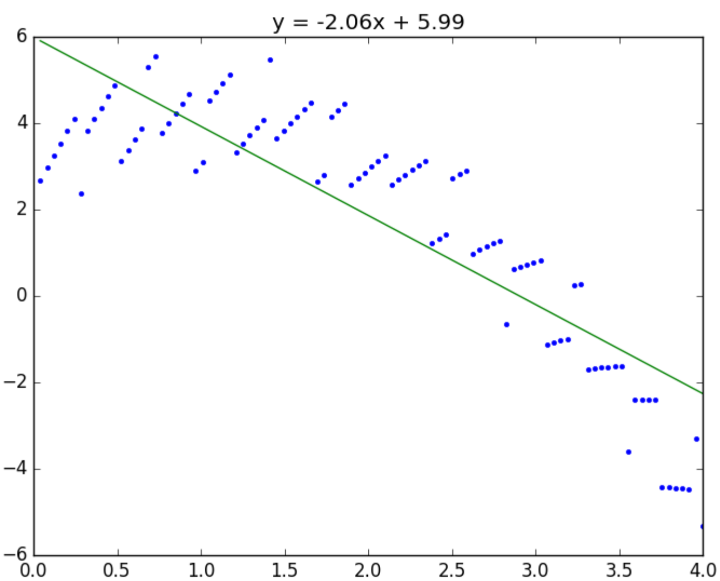
Homework 3

CS 4964 - Math for Data
Nick Porter

November 26, 2016

Exercise 1. *Let the first column of the data set be the explanatory variable x, and let the fourth column be the dependent variable y.*

- (a) Run simple linear regression to predict y from x. Report the linear model you found. Predict the value of y for new x values 1, for 2, and for 3.



After running a simple linear regression I found the model to be:

$$f(x) = -2.06x + 5.99$$
$$f(1) = 3.93$$
$$f(2) = 1.87$$
$$f(3) = -0.19$$

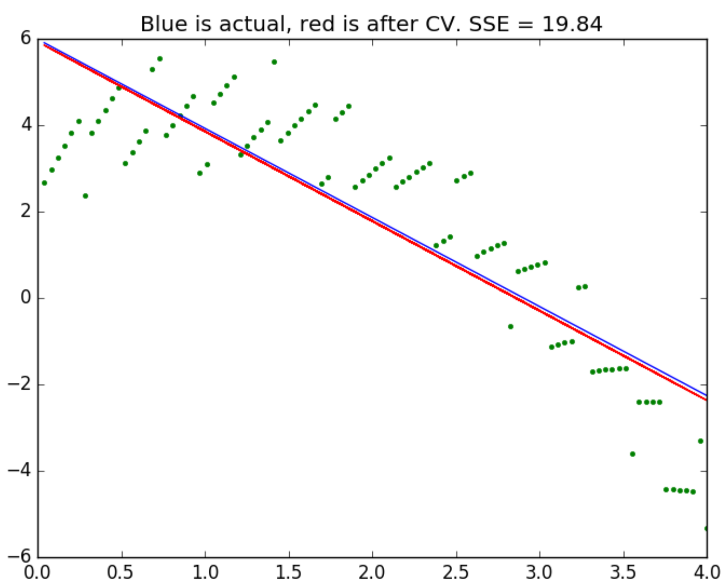
- (b) Use cross-validation to predict generalization error, with error of a single data point (x, y) from a model M as $(M(x) - y)^2$. Describe how you did this, and which data was used for what.

  To split my data into training and testing sets I used a SKLearn function, and used 10% as testing data and left the other 90% to train my new model. Then I computed a new model based on the training data. After that I computed the SSE based off the testing data.

```
# Split data randomly into train and test sets
X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0.1)

# Build our new model from the training data
trainM, trainB = np.polyfit(X_train.values.flatten(), y_train.values.flatten(), 1)
p = np.poly1d([trainM, trainB])

# Evaluate our new model on our test data.
sse = 0
for i in range(0, len(X_test)):
    xValue = X_test.iloc[i][0]
    yValue = y_test.iloc[i][0]
    sse = sse + (p(xValue) - yValue)**2
```



- (c) On the same data, run polynomial regression for p = 2, 3, 4, 5. Report polynomial models for each. With each of these models, predict the value of y for a new x values of 1, for 2, and for 3.

```
def poly_regression(x, y, degree):
    coefs = np.polyfit(x.values.flatten(), y.values.flatten(), degree)
    p = np.poly1d(coefs)
```

When $p = 2$

$$\begin{gathered} f(x) = -1.02x^2 + 2.08x + 3.16 \\ f(1) = 4.229 \\ f(2) = 3.236 \\ f(3) = 0.188 \end{gathered}$$

When $p = 3$

$$\begin{gathered} f(x) = 0.025x^3 - 1.18x^2 + 2.33x + 3.08 \\ f(1) = 4.264 \\ f(2) = 3.237 \\ f(3) = 0.153 \end{gathered}$$

When $p = 4$

$$\begin{gathered} f(x) = -0.034x^4 + 0.30x^3 - 1.91x^2 + 3x + 2.94 \\ f(1) = 4.303 \\ f(2) = 3.190 \\ f(3) = 0.187 \end{gathered}$$

When $p = 5$

$$\begin{gathered} f(x) = 0.03x^5 - 0.035x^4 + 1.44x^3 + 3.64x^2 + 4.02x + 2.79 \\ f(1) = 4.294 \\ f(2) = 3.187 \\ f(3) = 0.202 \end{gathered}$$

- (d) Cross-validate to choose the best model. Describe how you did this, and which data was used for what.

  To find the p which provides the best model we will use cross validation to help us gauge the effectiveness of each model.

  We will perform the operations described below 10,000 times to get a good average.

  First use train test split method to split our data 90% train, 10% test for each iteration 1 - 10,000.
  Then for each value of p $[1 - 5]$ we will evaluate the SSE and add it to the running sum for that value of p.

  Next we will compute the average by taking SSE of each p / 10,000.

Degree	SSE
1	19.643
2	4.633
3	4.625
4	4.605
5	4.593

As we can see the degree 5 polynomial has the lowest SSE, making it the most accurate out of all our models. However to confirm this we may want to plot the lines and ensure that a degree 5 polynomial is not doing anything too extreme at the ends.

Exercise 2. *Now let the first three columns of the data set be separate explanatory variables x_1, x_2, x_3. Again let the fourth column be the dependent variable y.*

- (a) Run linear regression simultaneously using all three explanatory variables. Report the linear model you found. Predict the value of y for new (x1,x2,x3) values (1,1,1), for (2, 0, 4), and for (3, 2, 1).

```
x3 = pandas.read_csv('D3.csv', usecols = [0,1,2])
x3 = sm.add_constant(x3)
model = sm.OLS(y,x3).fit()
```

$$f(x_1, x_2, x_3) = -2.042x_1 + 0.561x_2 - 0.292x_3 + 5.4137$$

$$f(1, 1, 1) = 3.640$$
$$f(2, 0, 4) = 0.161$$
$$f(3, 2, 1) = 0.117$$

- (b) Use cross-validation to predict generalization error, with error of a single data point (x_1, x_2, x_3, y) from a model M as $(M(x_1, x_2, x_3)y)^2$. Describe how you did this, and which data was used for what.

To split my data into training and testing sets I used a SKLearn function, and used 10% as testing data and left the other 90% to train my new model. Then I computed a new model based on the training data. After that I computed the SSE based off the testing data.

I ran the following operation on all the testing data.

```
sse += ((xValue_1 * params[1] + xValue_2 * params[2] + xValue_3 * params[3] +
params[0]) - yValue)**2
```

After running the operation I was given a SSE of 14.084

Exercise 3. *Consider two functions*
$f_1(x, y) = (x - 2)^2 + (y - 3)^2$
$f_2(x, y) = (1 - (y - 3))^2 + 20((x + 3) - (y - 3)^2)^2$
Starting with $(x, y) = (0, 0)$ *run the gradient descent algorithm for each function. Run for* T *iterations, and report the function value at the end of each step.*

- (a) First, run with a fixed learning rate of $\gamma = 0.5$.

  For both functions I ran the gradient descent function for $T = 10$.

$f_1(x, y)$

i	(x, y)
0	(0,0)
1	(2,3)
2	(2,3)
3	(2,3)
4	(2,3)
5	(2,3)
6	(2,3)
7	(2,3)
8	(2,3)
9	(2,3)

We ended up with a final gradient of $< 0, 0 >$

$f_2(x, y)$

i	(x, y)
0	(0,0)
1	(120,724)
2	(1.03944800e+07, -1.49886671e+10)
3	(4.49320284e+21, 1.34694243e+32)
4	(3.62850784e+65, -9.77478235e+97)
5	(1.91092740e+197, 3.73577989e+295)
6	(inf, -inf)
7	(nan,nan)
8	(nan,nan)
9	(nan,nan)

We ended up with integer overflow. Unsure how to handle this.

```
def gradient_descent(gradf, x, y, learning_rate, iterations):

    v_init = np.array([x, y])
    values = np.zeros([iterations, 2])
```

```
    values[0, :] = v_init
    v = v_init

    for i in range(1, iterations):
        print v
        v = v - learning_rate * gradf(v[0], v[1])
        values[i, :] = v

print LA.norm(gradf(v[0], v[1]))
```

- (b) Second, run with any variant of gradient descent you want. Try to get the smallest function value after T steps.

  For $f_1(x, y)$ I ran the same gradient descent algorithm, however I used a learning rate of 0.01, a starting point of $(5, 5)$ and $T = 100$

  After 100 iterations I was given back a value of 0.975 for the norm of the gradient at the point $(2.414, 3.276)$

  We noticed that the original parameters resulted in a far better result, using few iterations as well.

  On $f_2(x, y)$ I ran the same gradient descent algorithm however with the starting point of $(6, 6)$, learning rate at 0.0007 and set for $T = 100$

  The smaller learning rate allowed this run to not overflow and converge.

  After 100 iterations I was given back a value of 0.656 for the norm of the gradient at the point $(5.955, 5.989)$