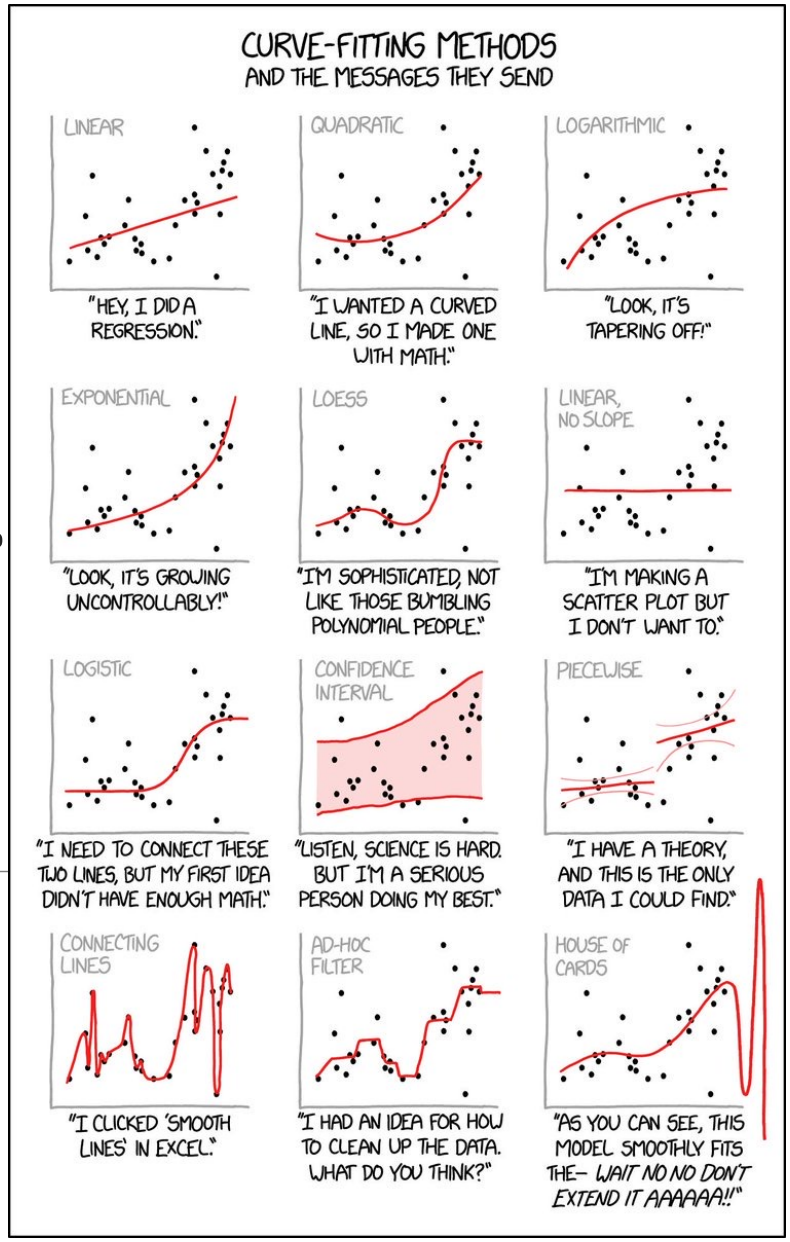
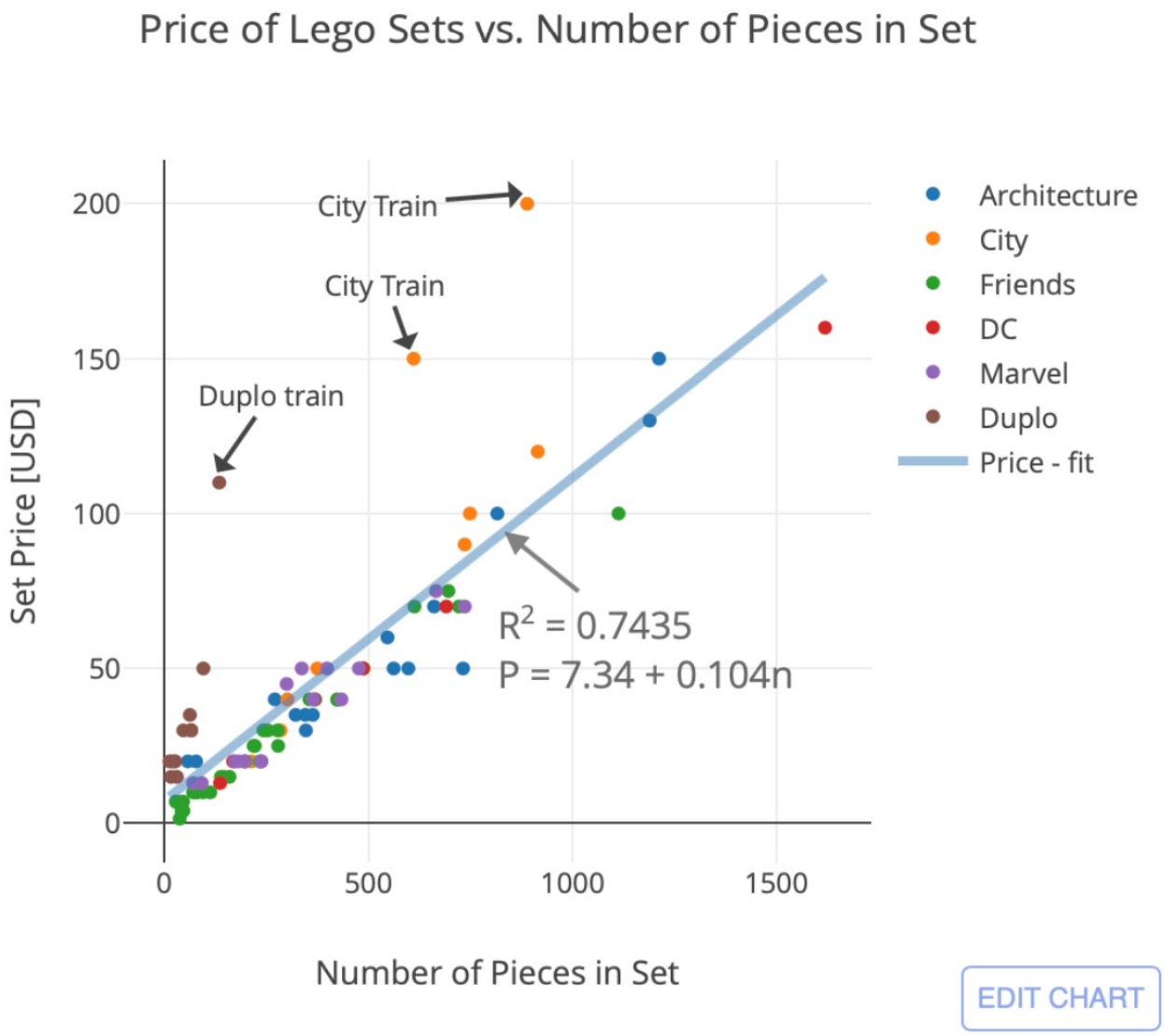
Curve-fitting, Train-validate-test

MIT Department of Electrical Engineering and Computer Science



Announcements

Distributions Alone Lack Explanatory Power

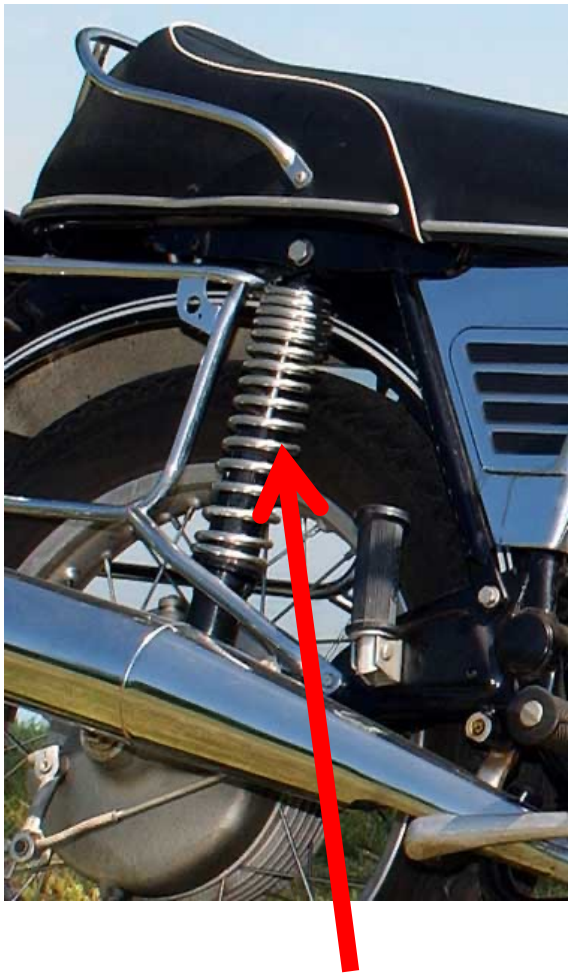


https://www.wired.com/2014/08/lego-cost

Curve Fitting: Hooke's Law

- Robert Hooke
  - 1635-1703
  - Curator of experiments of the Royal Society
  - Surveyor for City of London after the Great Fire
  - **Discovered law of elasticity**
    - Led to invention of balance spring, which led to first accurate watch
  - Observed that gravity was an inverse effect, but didn't know it was inverse square (Newton gets credit for that)
  - Huge believer in running experiments and then building models
    - *"It is commonly believed that anyone who tabulates numbers is a statistician. This is like believing that anyone who owns a scalpel is a surgeon."*
    - *"The truth is, the Science of Nature has been already too long made only a work of the Brain and the Fancy: It is now high time that it should return to the plainness and soundness of Observations on material and obvious things."*

Example: Curve Fitting With Springs



$k \approx 35{,}000\,N/m$

$k \approx 1N/m$

Linear spring: amount of force needed to stretch or compress spring is linear in the distance the spring is stretched or compressed, up to some maximum force

Each spring has a spring constant, k, that determines how much force is needed to achieve a specific compression

Hooke's Law

- $F = -k\delta$

Why the '–'? Because deflection in opposite direction to force

- How much does a rider have to weigh to compress spring 1 cm?

$$F = 0.01m * 35{,}000N/m$$

$$F = 350N$$

$$mass * 9.81m/s^2 = 350N$$

$$mass = \frac{350N}{9.81m/s^2}$$

$$mass = \frac{350}{9.81}\ \text{kg}$$

$$mass \approx 35.68\ \text{kg}\ (or\ about\ 79\ \text{lbs})$$

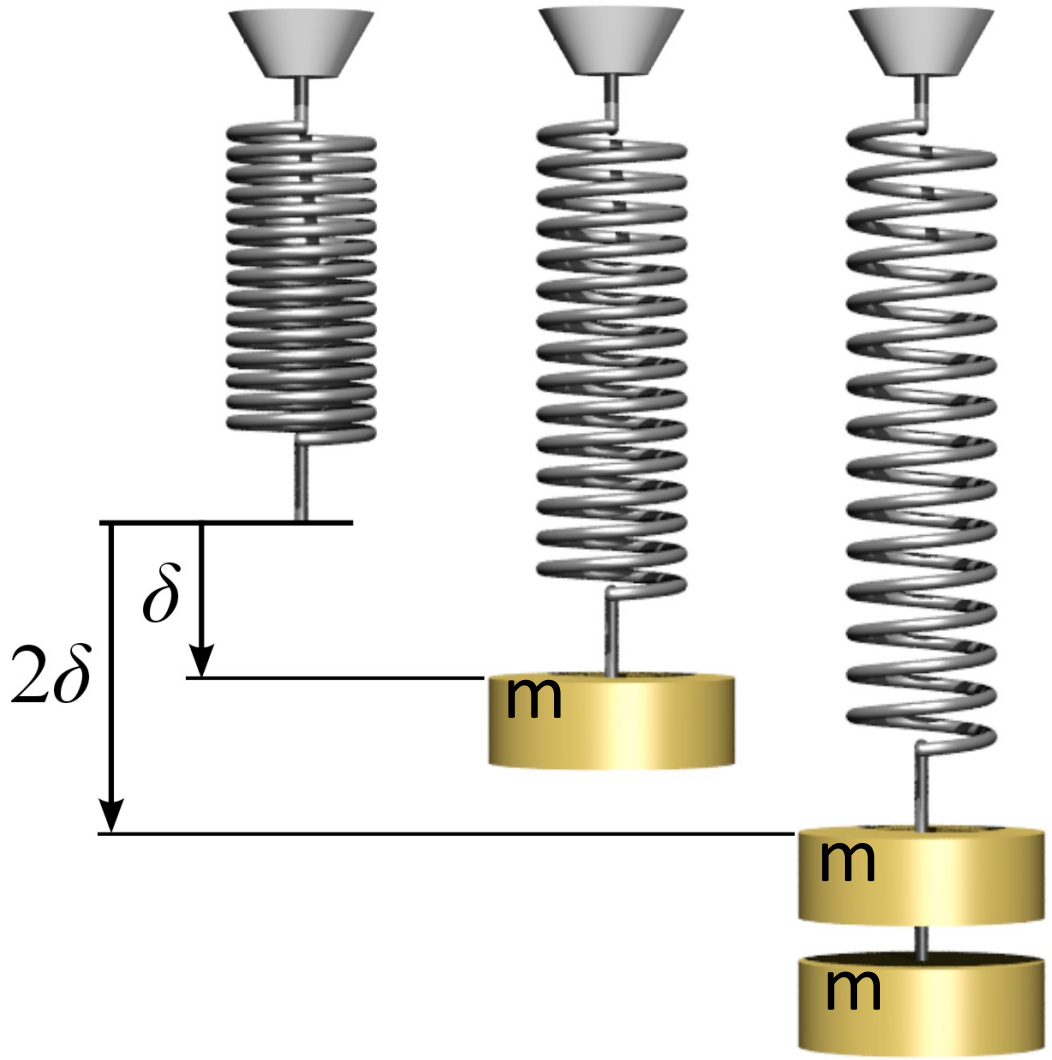
$$F = mass * acc$$

$$F = mass * 9.81m/s^2$$

Finding k

- $F = -k\delta$
- $k = -F/\delta$
- $k = 9.81*m/\delta$

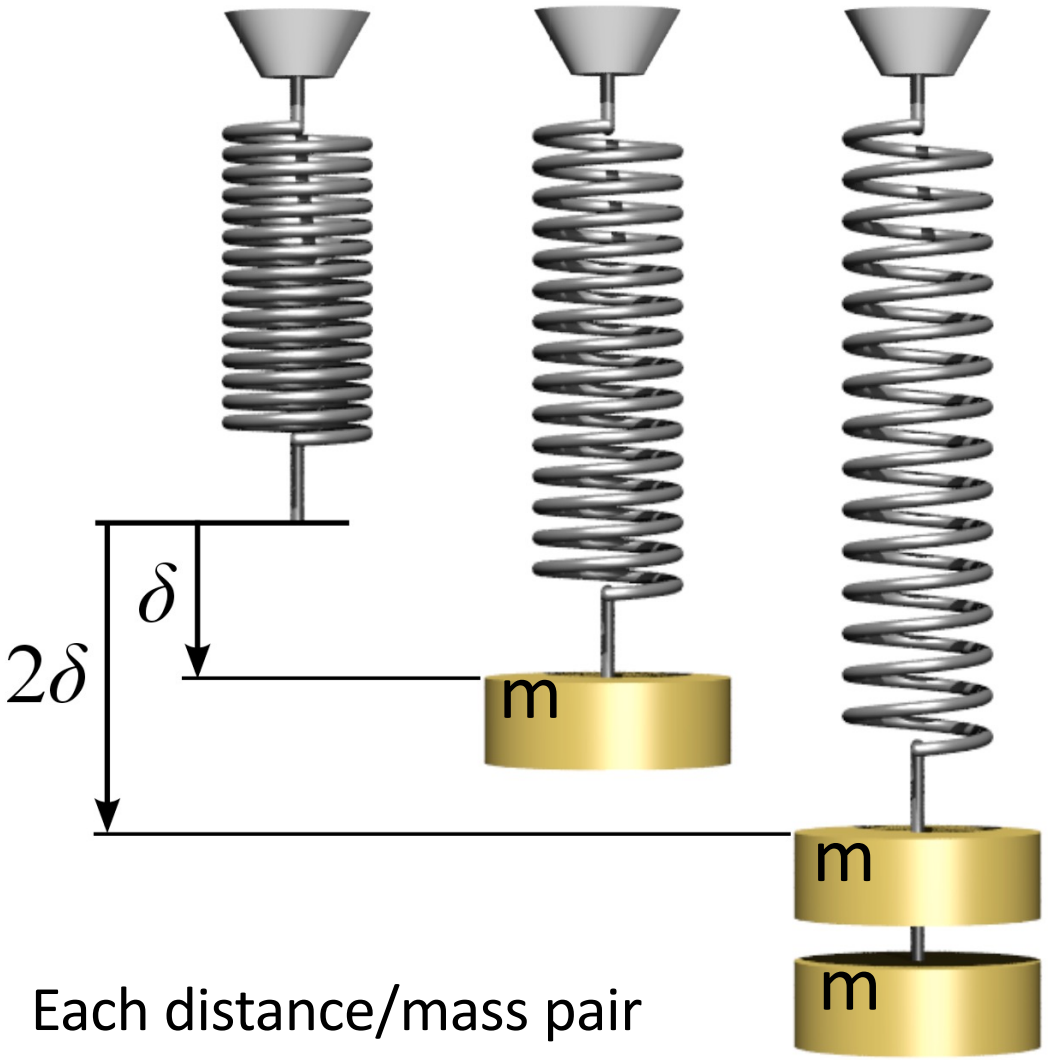
- Each trial estimates k



By Yapparina (Own work) [CC BY-SA 3.0 (http://creativecommons.org/licenses/by-sa/3.0)], via Wikimedia Commons

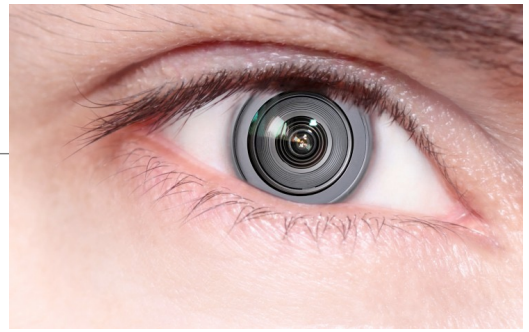
Some Data

Mass (kg)	Distance (m)
0.1	0.0865
0.15	0.1015
0.2	0.1106
0.25	0.1279
0.3	0.1892
0.35	0.2695
0.4	0.2888
0.45	0.2425
0.5	0.3465
0.55	0.3225
0.6	0.3764
0.65	0.4263
0.7	0.4562
0.75	0.4502
0.8	0.4499
0.85	0.4534
0.9	0.4416
0.95	0.4304
1.0	0.437



Each distance/mass pair provides an estimate of k

Taking a Look at the Data



```
def plotData(fileName):
    xVals, yVals = getData(fileName)
    xVals = np.array(xVals) #masses
    yVals = np.array(yVals) #distances/displacements
    xVals = xVals*9.81  #acc. due to gravity; forces
    plt.plot(xVals, yVals, 'bo',
             label = 'Measured displacements')
    labelPlot()
```

A reminder/primer about numpy arrays:

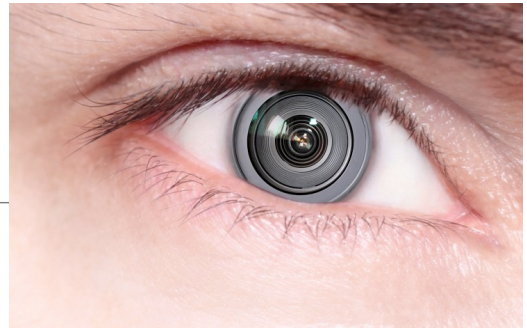
- Converts a list into a linear data structure
- Can treat arrays algebraically; e.g., if a and b are arrays, then:
  - a*2 multiplies every element of a by 2
  - a + 3 adds 3 to every element of a
  - a – b subtracts each element of b from corresponding element of a
  - a*b multiplies each element of a by corresponding element of b

Using Arrays


```
testArray = np.array([1,2,3,4,5,6,7,8,9])
print(testArray)
[1 2 3 4 5 6 7 8 9]
secondArray = np.array([3,2,6,4,1,8,7,5,9])
print(secondArray)
[3 2 6 4 1 8 7 5 9]

print(testArray * 2)
[ 2  4  6  8 10 12 14 16 18]
print(testArray + 3)
[ 4  5  6  7  8  9 10 11 12]
print(testArray - secondArray)
[-2  0 -3  0  4 -2  0  3  0]
print(testArray * secondArray)
[ 3  4 18 16  5 48 49 40 81]
```

Note: operations on arrays do not mutate the array, they create a new one

Taking a Look at the Data

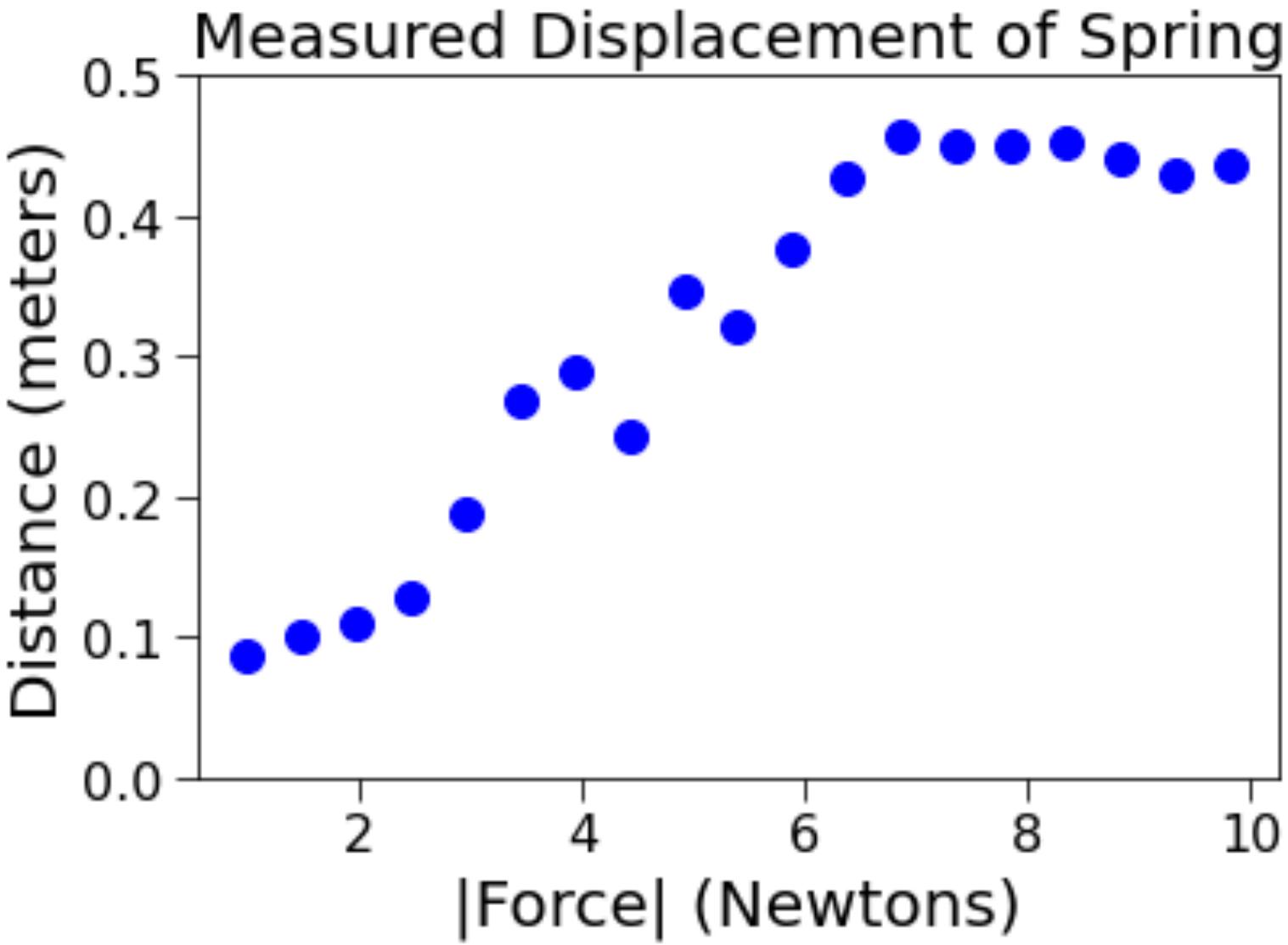


```
def get_data(filename):
    with open(filename, 'r') as data_file:
        y_vals, x_vals = [], []
        data_file.readline() # discard header
        for line in data_file:
            y, x = line.split()
            y_vals.append(float(y))
            x_vals.append(float(x))
        return np.array(x_vals), np.array(y_vals)

x_vals, y_vals = get_data('springData.txt')
x_vals = x_vals*9.81 # force due to gravity
plt.plot(x_vals, y_vals, 'bo', label='Measured displacements')
```

A reminder/primer about numpy arrays:

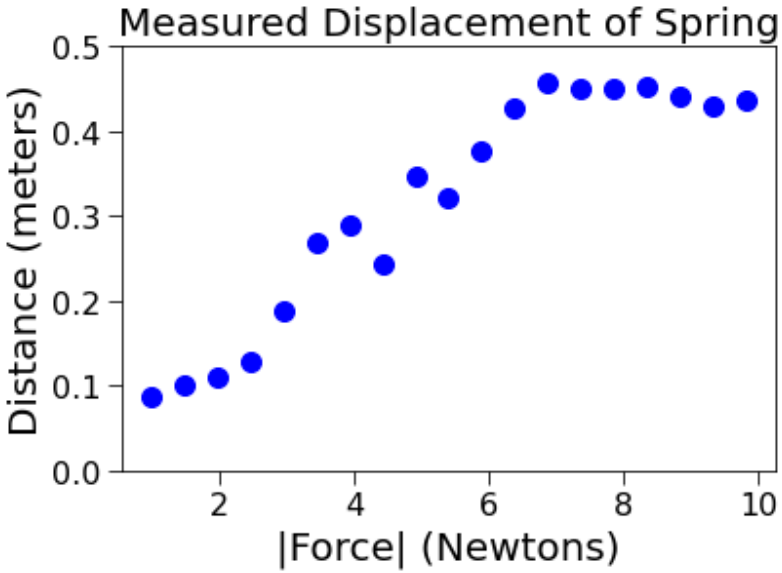
- Converts a list into a linear data structure
- Can treat arrays algebraically; e.g., if a and b are arrays, then:
  - a*2 multiplies every element of a by 2
  - a + 3 adds 3 to every element of a
  - a – b subtracts each element of b from corresponding element of a
  - a*b multiplies each element of a by corresponding element of b

Taking a Look at the Data



What Can We Do With This Data?

- We've run an experiment
- We can relate observations to trials (distance d vs. force F – or |F| in this case)
- Theory predicts a relationship between observations and trials
  - |F| = kd
- Can we use these measurements to determine k and to validate model?
- Try to fit a curve to data, and use to deduce relationship between observation and trial
- Notice that points don't lie on a line. Experiments are noisy!



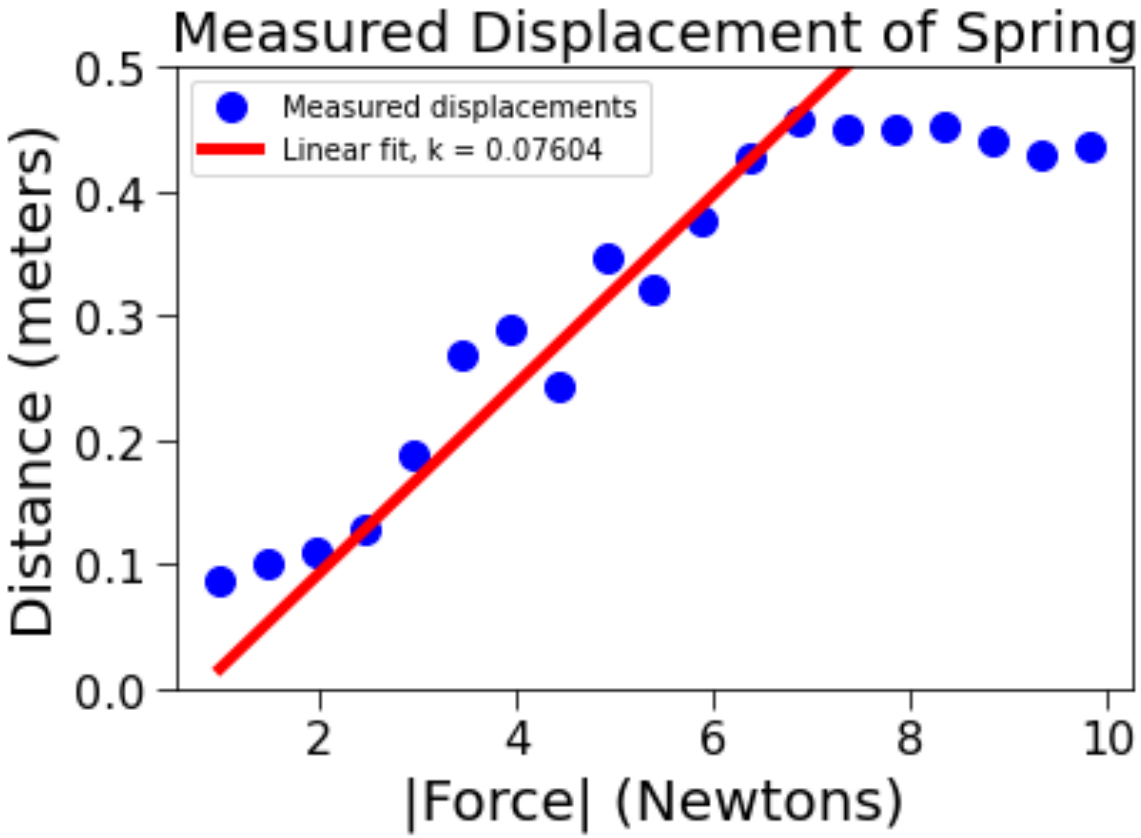
Why not just compute "average" k?

Mass (kg)	Distance (m)
0.1	0.0865
0.15	0.1015
0.2	0.1106
0.25	0.1279
0.3	0.1892
0.35	0.2695
0.4	0.2888
0.45	0.2425
0.5	0.3465
0.55	0.3225
0.6	0.3764
0.65	0.4263
0.7	0.4562
0.75	0.4502
0.8	0.4499
0.85	0.4534
0.9	0.4416
0.95	0.4304
1.0	0.437

- Naïve solution : For each data point, compute k = m/d
  Then take the average
- Obvious problem : what if we had a measurement for m = 0 ?
- Even measurements for small m are not as precise
- Only works for a single parameter and a simple law like Hooke's law
- Doesn't tell us how good the law is

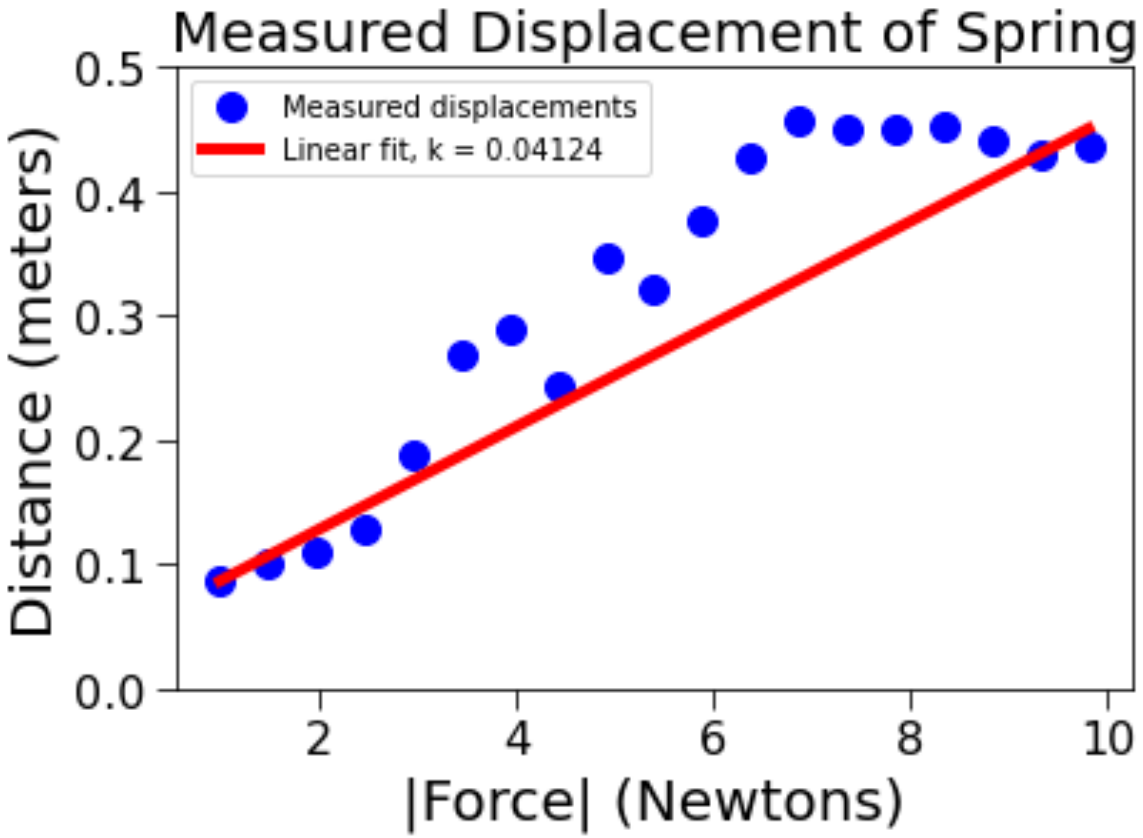
Could Just Try Guessing Line?

- Pick two sample points to define line and fit, measure slope



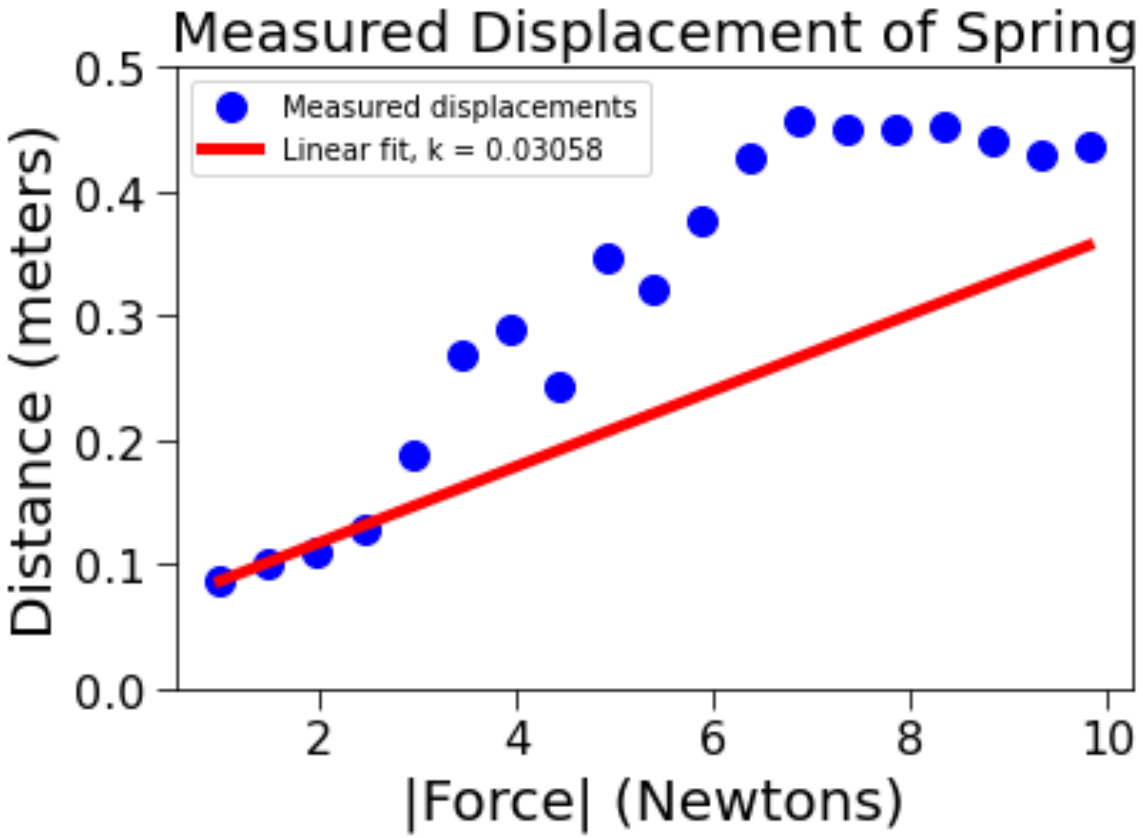
Could Just Try Guessing Line?

- Pick two sample points to define line and fit, measure slope



Could Just Try Guessing Line?

- Pick two sample points to define line and fit, measure slope



Need some way to evaluate how well the line fits the data

Also need a robust algorithm that finds "good" fits

Fitting Curves to Data – back to optimization

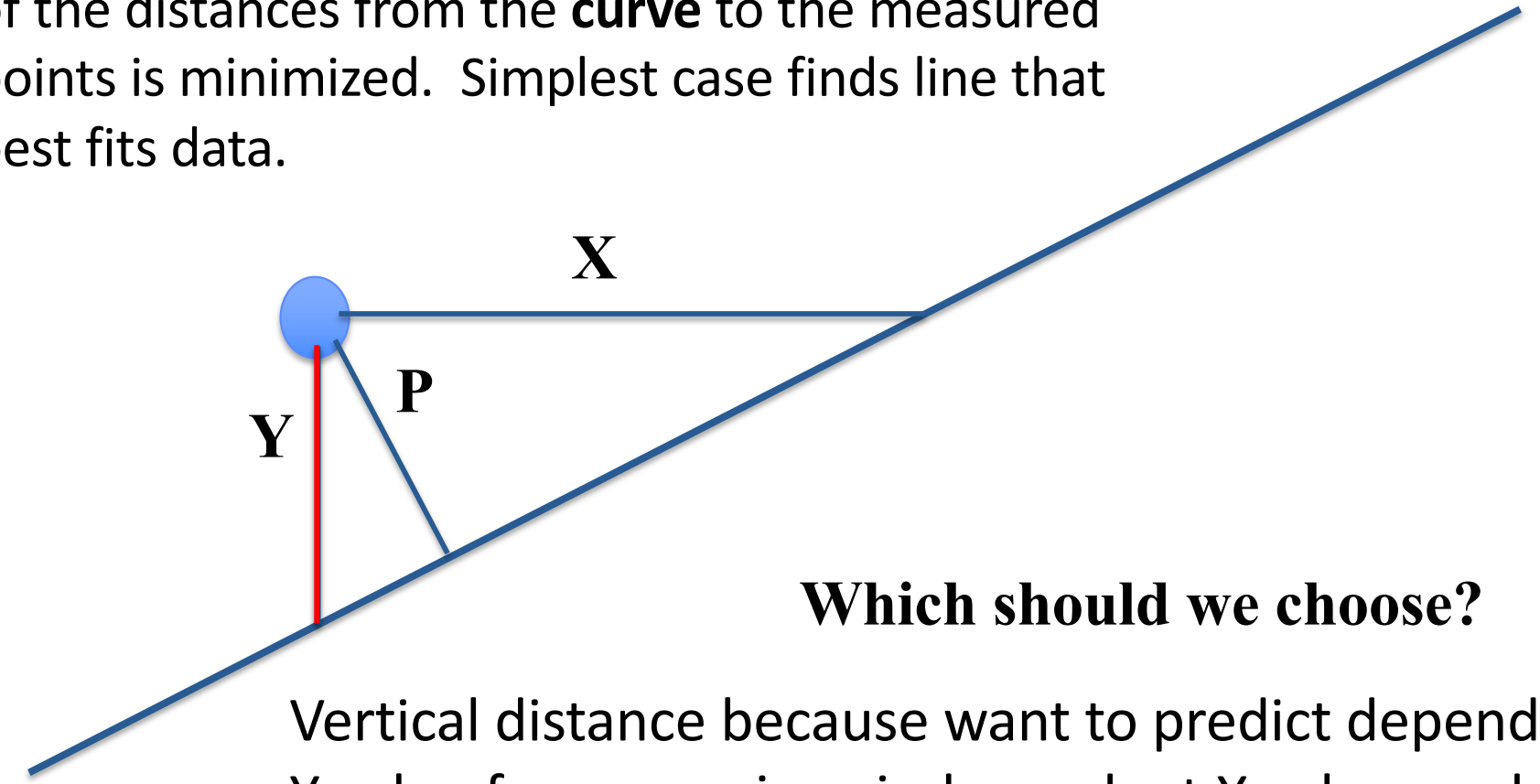
- When we fit a curve to a set of data, we are finding a fit that relates an independent variable (the mass or force) to an estimated value of a dependent variable (the distance or displacement)

Fitting Curves to Data – back to optimization

- When we fit a curve to a set of data, we are finding a fit that relates an independent variable (the mass or force) to an estimated value of a dependent variable (the distance or displacement)
- To decide how well a curve fits the data, we need a way to measure the goodness of the fit – called the **objective function (aka loss)**
- Once we define the objective function, we also need an algorithm to find the curve that minimizes it
- Theory says find a **curve** such that some function of the distances from the **curve** to the measured points is minimized. Simplest case finds line that best fits data.
- So need **objective function** that measures distances from curve, and **algorithm** to find best curve

Measuring Distance

Theory says find a **curve** such that some function of the distances from the **curve** to the measured points is minimized. Simplest case finds line that best fits data.



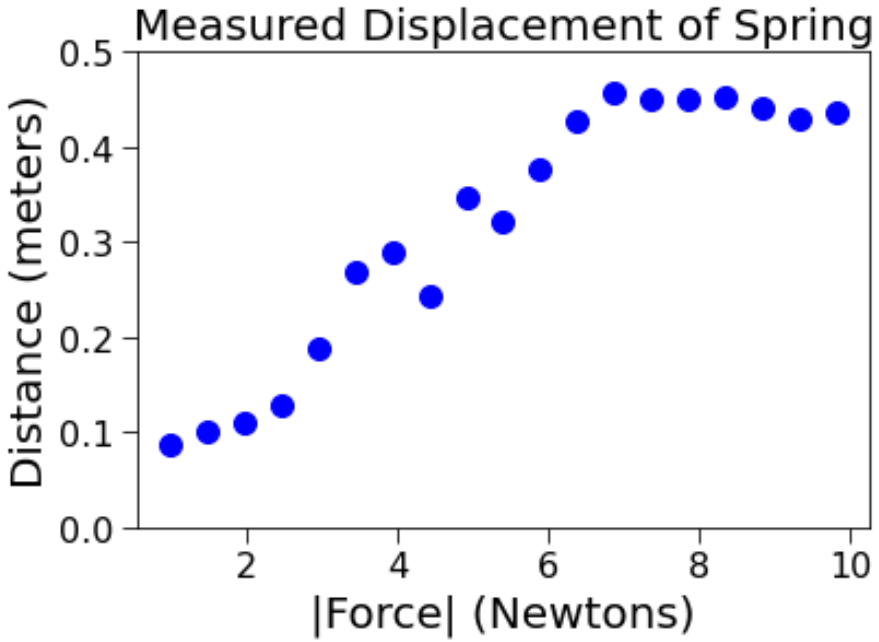
Which should we choose?

Vertical distance because want to predict dependent Y value for every given independent X value, and vertical distance measures error in that prediction

Least Squares Objective Function

$$\sum_{i=0}^{len(observed)-1} (observed[i] - predicted[i])^2$$

- *observed*: actual measurements for each trial
- *predicted*: value that model suggests for each trial (value on the fitted curve)



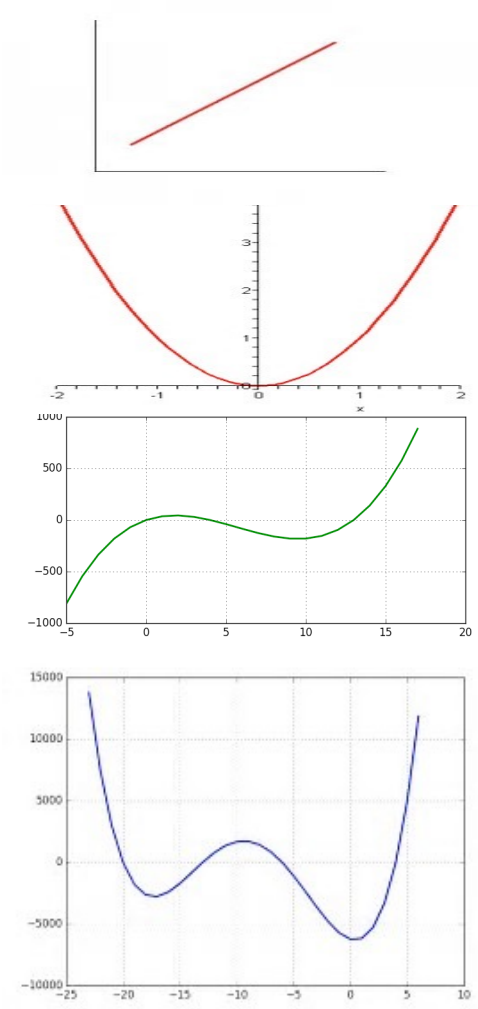
Solving for Least Squares

$$\sum_{i=0}^{len(observed)-1} (observed[i] - predicted[i])^2$$

- To minimize this objective function, want to find a curve for the **predicted** observations that leads to minimum value of sum of squared differences
  - Remember that curve will define, for each independent variable value, the associated predicted dependent variable value
- Need to make a choice on kinds of curves – we will use polynomials of one variable
- Need to find the best curve – use **linear regression** to find a polynomial representation for the predicted model that minimizes the objective function

Aside: Polynomials with One Variable (x)

- Definition: 0 or sum of finite number of non-zero terms
- Each term of the form cx^p
  - c, the coefficient, a real number
  - p, the degree of the term, a non-negative integer
- The degree of the polynomial is the largest degree of any non-zero term
- Examples
  - Line: $ax + b$
  - Parabola: $ax^2 + bx + c$
  - Cubic: $ax^3 + bx^2 + cx + d$
  - Quartic: $ax^4 + bx^3 + cx^2 + dx + e$



Solving for Least Squares

$$\sum_{i=0}^{len(observed)-1} (observed[i] - predicted[i])^2$$

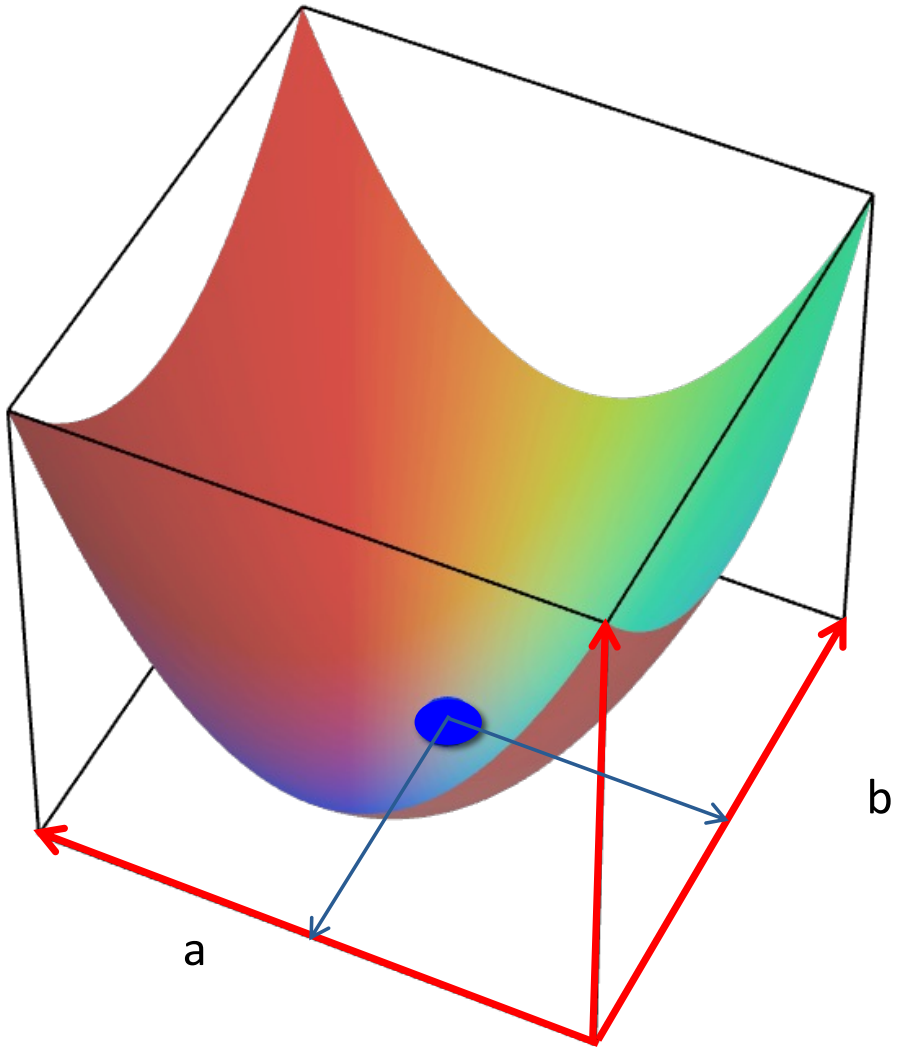
- Simple example:
  - Use a degree-one polynomial, $y = ax+b$, as model of data (best fitting line)
- Want to find values of a and b such that

$$\sum_{i=0}^{len(observed)-1} (observed[i] - a * x[i] - b)^2$$

is minimized, where *x[i]* is the ith data point, and *observed[i]* is the corresponding measured value

Finding the Best Curve (simplest case)

- Set of all possible lines represented by points in (*a, b*) space
  - Point defines line ax+b
- Imagine a surface in this space, where height is value of the objective function
- The objective function is quadratic in terms of a and b, so this surface is a two-dimensional parabola
- The minimum point can be found by “walking downhill”, or by analytic differentiation



Some Properties of Linear Regression

- Objective surface using sum-of-squared-differences is differentiable
  - Means we can compute gradient direction analytically and use to efficiently compute next step to take in searching space for optimal objective function value
- Objective surface in this case has a global minimum
  - Means there is always a unique best fit
- Easy to solve for (linear system)
  - Minimum: we want derivative to be zero
  - Derivative of a quadratic is linear

Derive it!

- For y = ax + b where y is dependent variable and x is independent variable
- yi, xi are measurements
- Find a & b that minimize $\Sigma(yi - axi - b)^2$

Derive it!

- For y = ax + b where y is dependent variable and x is independent variable
- yi, xi are measurements
- Find a & b that minimize $\Sigma(yi - axi - b)^2$
- Set derivative wrt a and wrt to b to zero
- Derivative wrt a: $\Sigma - 2\text{xi}(yi - axi - b)$ = 0
- Wrt b: Σ -$(yi - axi - b)$ = 0
- The xi and yi are known.
- We have two linear equations in a & b.
  - Easy to solve !

polyfit

- You could certainly write your own linear regression code
- Good news is that numpy provides built in functions to find these polynomial fits
- `np.polyfit(observed_x, observed_y, n)`

finds coefficients of a polynomial, of degree n, that provides a best least squares fit for the observed data

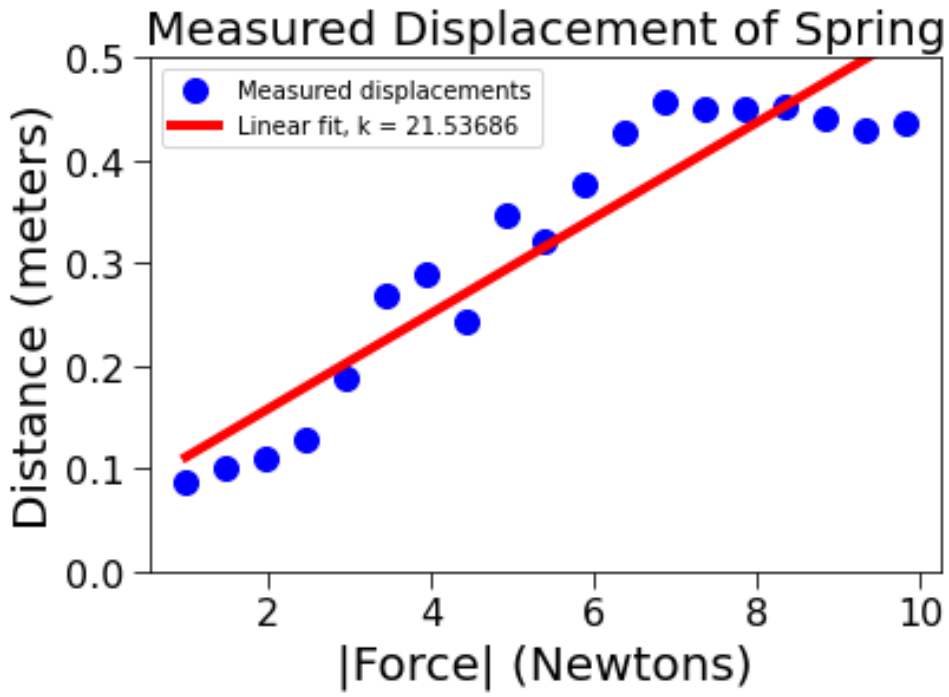
- n = 1 → best line $y = ax + b$
- n = 2 → best parabola $y = ax^2 + bx + c$
- n = 3 → best cubic $y = ax^3 + bx^2 + cx + d$

Using polyfit

```
a, b = np.polyfit(x_vals, y_vals, 1)
y_pred = a*x_vals + b
k = 1/a
print(f'a = {a:.5f}, b = {b:.5f}')
plt.plot(x_vals, y_pred)
```

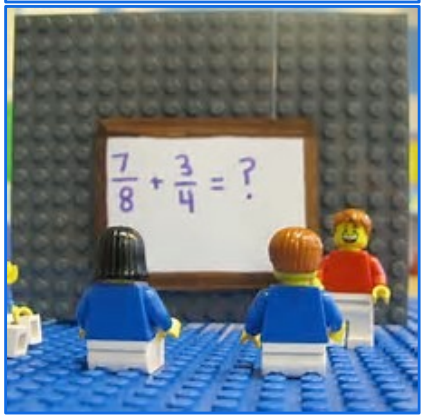
Remember Hooke:
F = kd
Here plotting d = aF
So k = 1/a

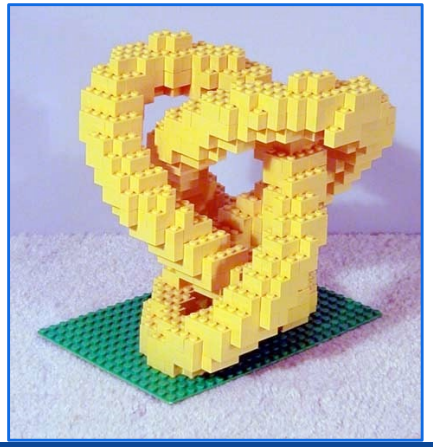
```
model = np.polyfit(x_vals, y_vals, 1)
y_pred = np.polyval(model, x_vals)
k = 1/model[0]
print(f'a = {model[0]:.5f},
      b = {model[1]:.5f}')
plt.plot(x_vals, y_pred)
```



Quick Summary So Far

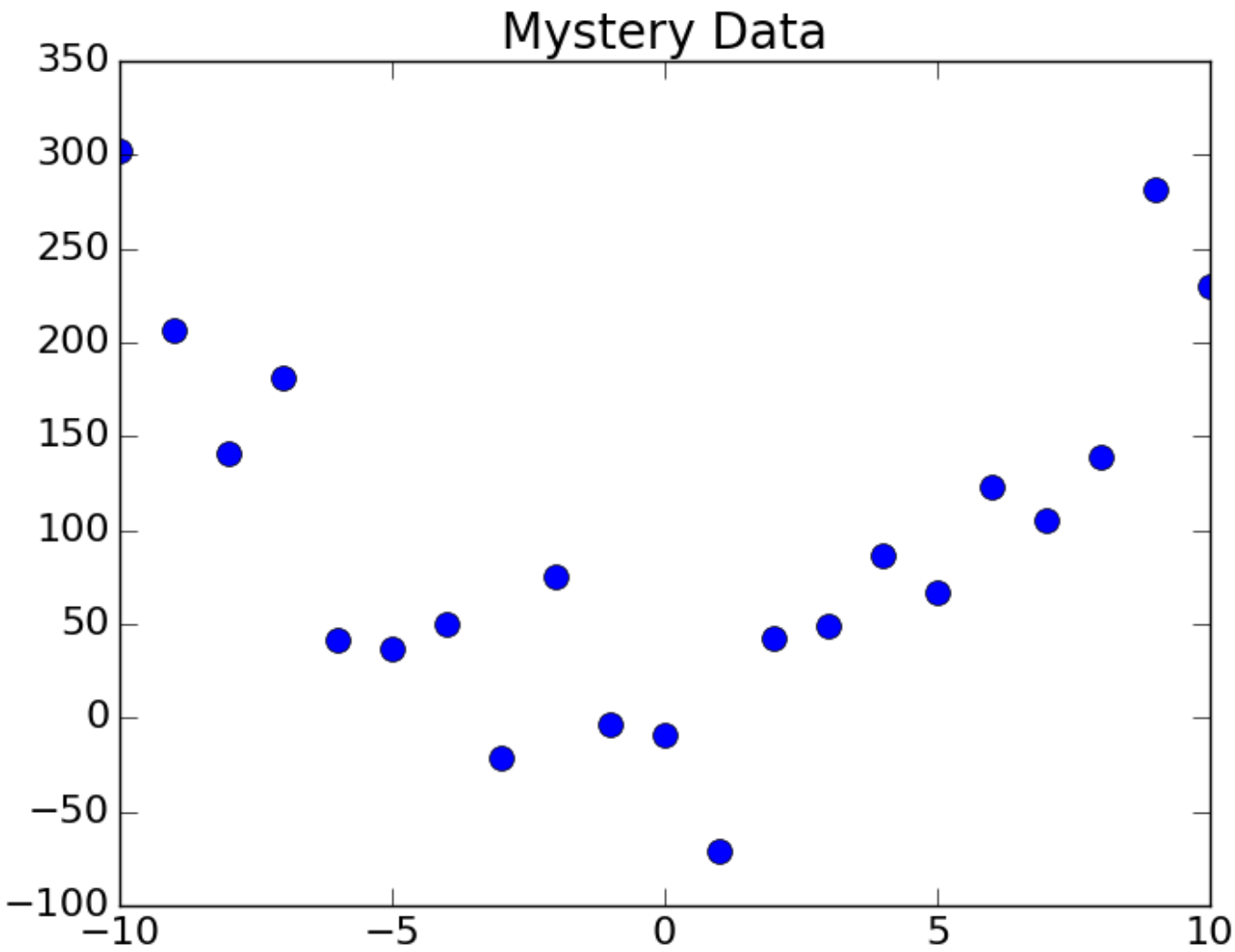
- Ran an experiment to gather data
- Theory predicts relationship between measurements (displacements) and experimental parameters (masses or forces)
- Linear regression lets us fit best model (line in our case) to observed data
  - Best here means minimize sum squared error between observed and predicted values
- So, let's apply this idea to other data…



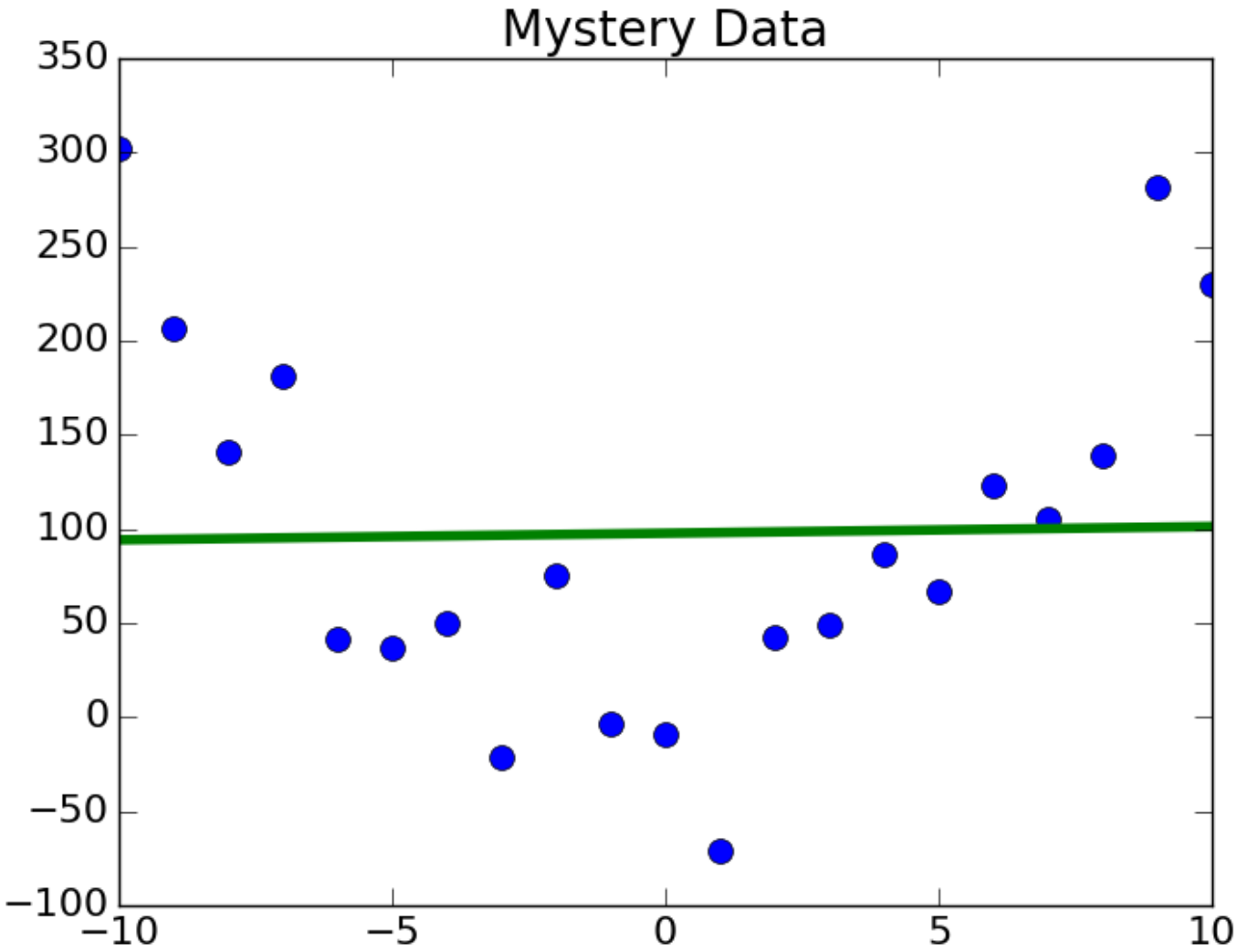


Five Minute Break

Another Experiment



Fit a Line



Let's Try a Higher-degree Model

```
model2 = np.polyfit(x_vals, y_vals, 2)
plt.plot(x_vals, np.polyval(model2, x_vals))
```

- Now we are fitting a best **parabola** instead of a best line
- Sometimes, this is still called ***linear regression***, because the model is linear in the **parameters** (i.e. coefficients of the polynomial)
- Others call it ***polynomial regression***

$$\sum_{i=0}^{len(observed)-1} (observed[i] - \boldsymbol{a} * x[i]^2 - \boldsymbol{b} * x[i] - \boldsymbol{c})^2$$

Two meanings of linear in linear regression

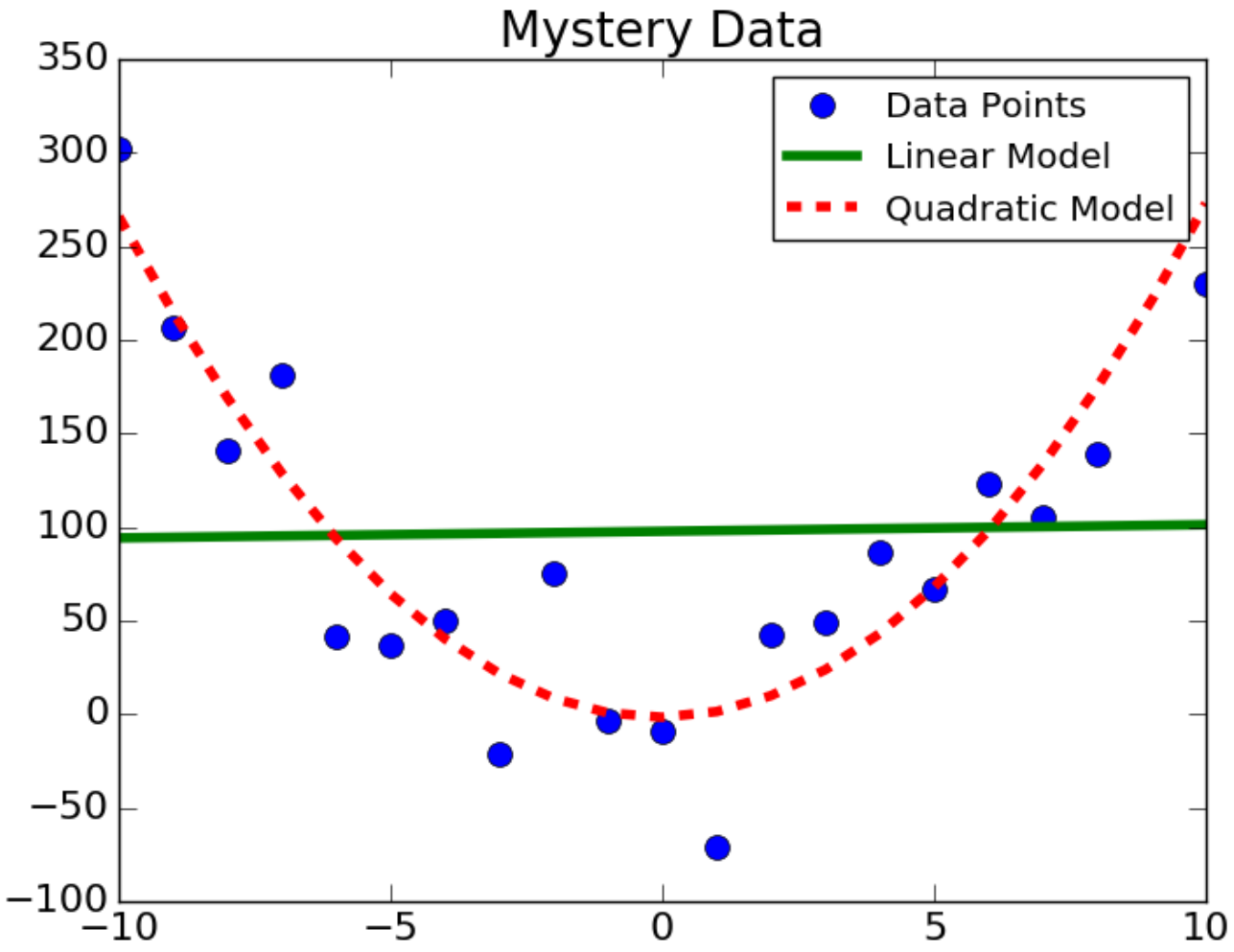
- 1/ the model we find is a linear polynomial
  - Linearity wrt to *x* or *m* here

Two meanings of linear in linear regression

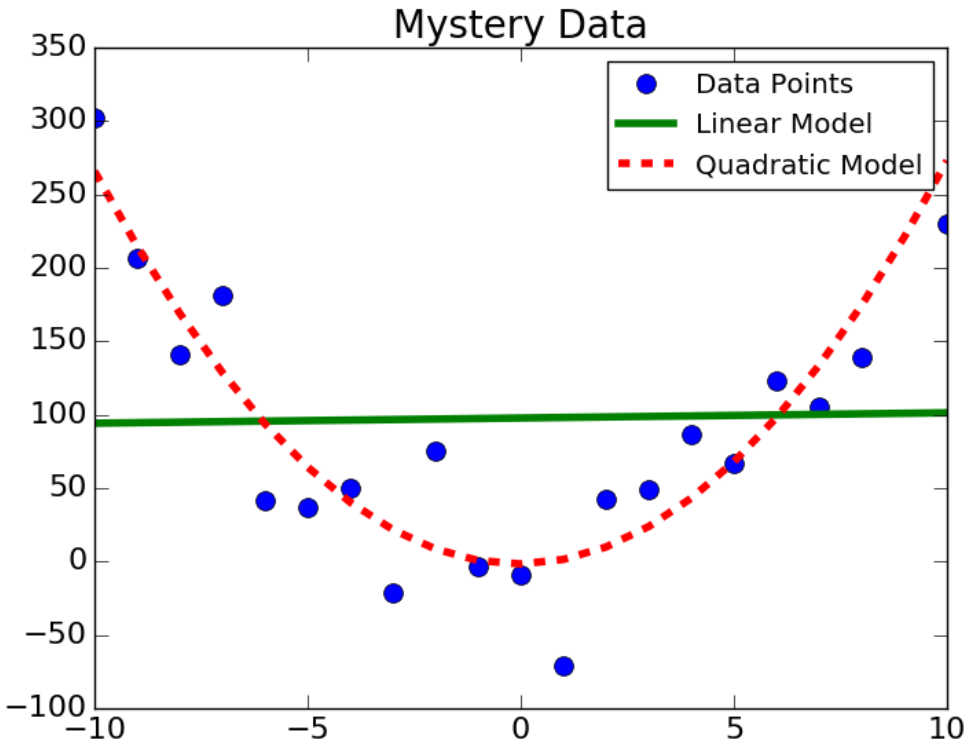
- 1/ the model we find is a linear polynomial
  - Linearity wrt to x or m here
- 2/ The equations we solve are linear in the model parameters
  - Linearity wrt k or a and b here
  - True for any polynomial : i.e. even when we fit a quadratic $y = ax^2 + bx + c$ it's still linear in a, b, c even though it's not linear in x.

- The world can be confusing.

Quadratic Appears to be a Better Fit



How Good Are These Fits?



- How good are they relative to each other?
- How good are they in an absolute sense?

Relative to Each Other

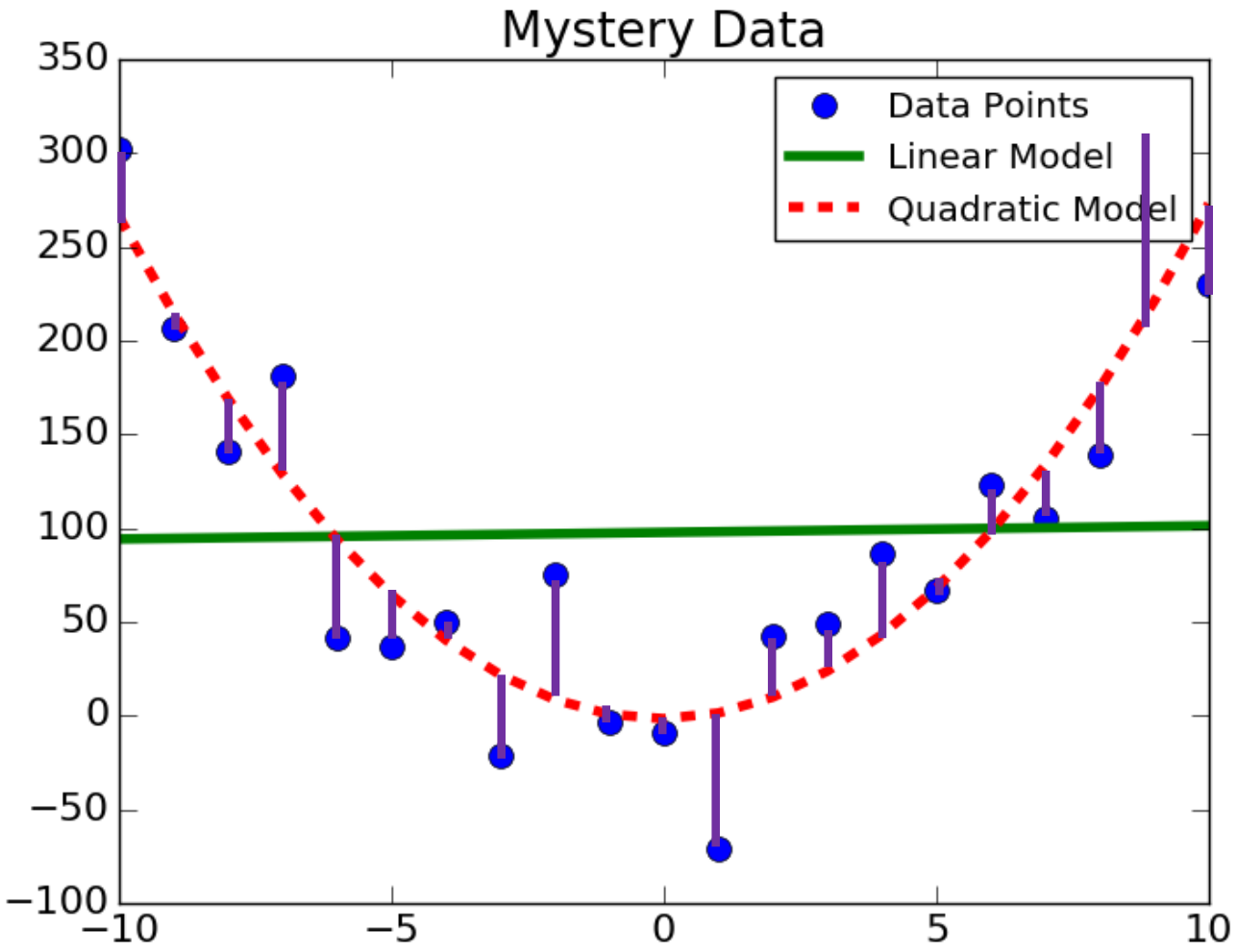
- Fit is a function from the independent variable to the dependent variable
  - Given an independent value, provides an estimate of the dependent value
- Which model provides better estimates?
  - Since we found fit by minimizing sum of square error, could just evaluate goodness of fit by looking at that error

We Can Look at Mean Squared Error

Measure error of each data point

Take square of difference

Find average across all of these squared differences



Comparing Mean Squared Error

```
def mean_squared_error(data, predicted):
    error = 0.0
    for i in range(len(data)):
        error += (data[i] - predicted[i])**2
    return error/len(data)
```

Comparing Mean Squared Error

```
def mean_squared_error(data, predicted):
    error = 0.0
    for i in range(len(data)):
        error += (data[i] - predicted[i])**2
    return error/len(data)

y_pred = np.polyval(model1, x_vals)
print('Mean squared error for linear model =',
      mean_squared_error(y_vals, y_pred))

y_pred = np.polyval(model2, x_vals)
print('Mean squared error for quadratic model =',
      mean_squared_error(y_vals, y_pred))
```

Mean squared error for linear model = 9372.73
Mean squared error for quadratic model = 1524.02

Given this improvement in mean squared error from linear to quadratic model, is there something even better?

In an Absolute Sense

- Mean squared error useful for comparing two different models for the same data
- Is it also useful for getting a sense of absolute goodness of fit?
  - Is 1524 good?
- Hard to know – no bound on values; not scale independent
  - For example, if we double the masses, get double the error

In an Absolute Sense

- Mean squared error useful for comparing two different models for the same data
- Is it also useful for getting a sense of absolute goodness of fit?
  - Is 1524 good?
- Hard to know – no bound on values; not scale independent
  - For example, if we double the masses, get double the error
- Instead we use coefficient of determination, R^2,

$$R^2 = 1 - \frac{\sum_i (y_i - p_i)^2}{\sum_i (y_i - \mu)^2}$$

← Error in estimates

← Variability in measured data

y_i are measured values
p_i are predicted values
μ is mean of measured values

If You Prefer Code

$$R^2 = 1 - \frac{\sum_i (y_i - p_i)^2}{\sum_i (y_i - \mu)^2}$$

```
def r_squared(observed, predicted):
    error = ((predicted - observed)**2).sum()
    mean_error = error/len(observed)
    return 1 - mean_error / np.var(observed)
```

If You Prefer Code

$$R^2 = 1 - \frac{\sum_i (y_i - p_i)^2}{\sum_i (y_i - \mu)^2}$$

Subtracting two arrays component-wise

Squaring each element of an array

```
def r_squared(observed, predicted):
    error = ((predicted - observed)**2).sum()
    mean_error = error/len(observed)
    return 1 - mean_error / np.var(observed)
```

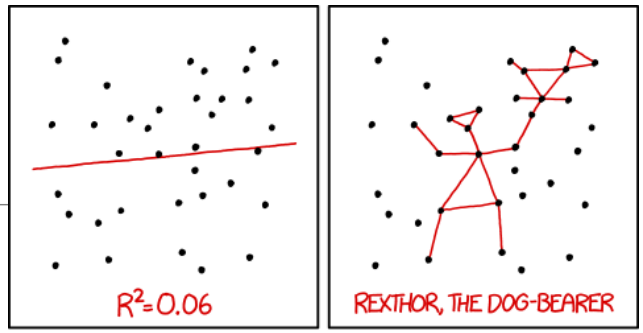
Need to invoke function

Add all elements

Note:

- Numerator is sum of squared errors
- Dividing by number of samples gives mean squared error
- Denominator is variance times number of samples
- So mean SSE/variance is same as R^2 ratio

R^2



- By comparing the estimation errors (the numerator) with the variability of the original values (the denominator), R^2 is intended to capture the proportion of variability in a data set that is accounted for by the statistical model provided by the fit
  - Said differently: compare model to a constant model
  - The mean is the best constant estimate under least squares

R^2

$$R^2 = 1 - \frac{\sum_i (y_i - p_i)^2}{\sum_i (y_i - \mu)^2}$$

- Always between 0 and 1 when fit generated by a linear regression* and tested on training data
  - If $R^2 = 1$, the model explains all of the variability in the data.
    - $\sum_i (y_i - p_i)^2 = 0$
  - If $R^2 = 0$, there is no relationship between the values predicted by the model and the actual data. (No better than constant prediction)
    - $\sum_i (y_i - p_i)^2 = \sum_i (y_i - \mu)^2$
  - If $R^2 = 0.5$, the model explains half the variability in the data.

* Assuming the model has a free constant term

Testing Goodness of Fits

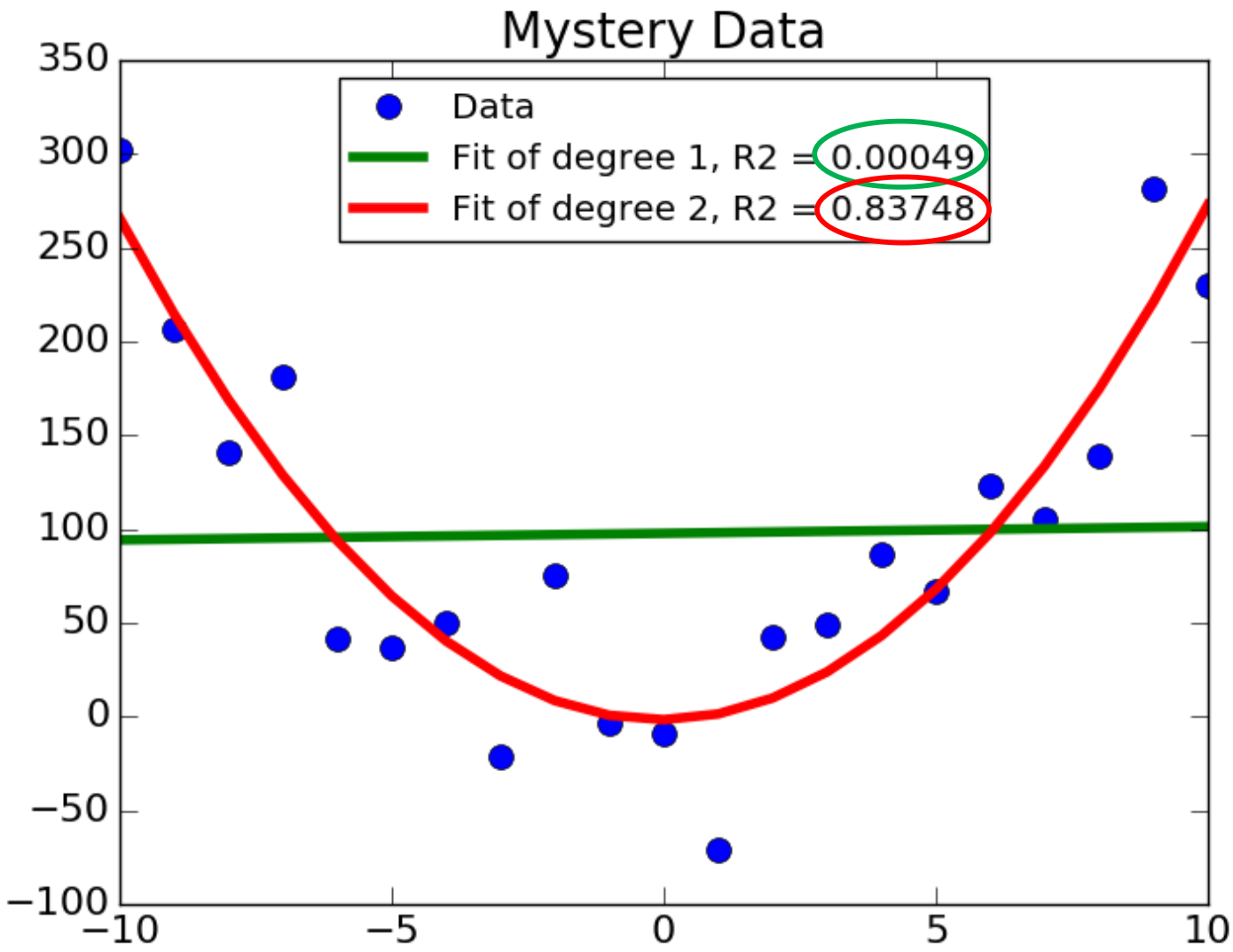
```
def gen_fits(x_vals, y_vals, degrees):
    models = []
    for d in degrees:
        model = np.polyfit(x_vals, y_vals, d)
        models.append(model)
    return models
```

List of ints: different order models to try

List of fitted polynomial models, each a list of coefficients

```
def test_fits(models, degrees, x_vals, y_vals, title):
    plt.figure()
    plt.plot(x_vals, y_vals, 'o', label='Data')
    for i in range(len(models)):
        y_pred = np.polyval(models[i], x_vals)
        error = r_squared(y_vals, y_pred)
        plt.plot(x_vals, y_pred,
                 label=f'Fit of degree {degrees[i]}')
    plt.legend()
    plt.title(title)
```

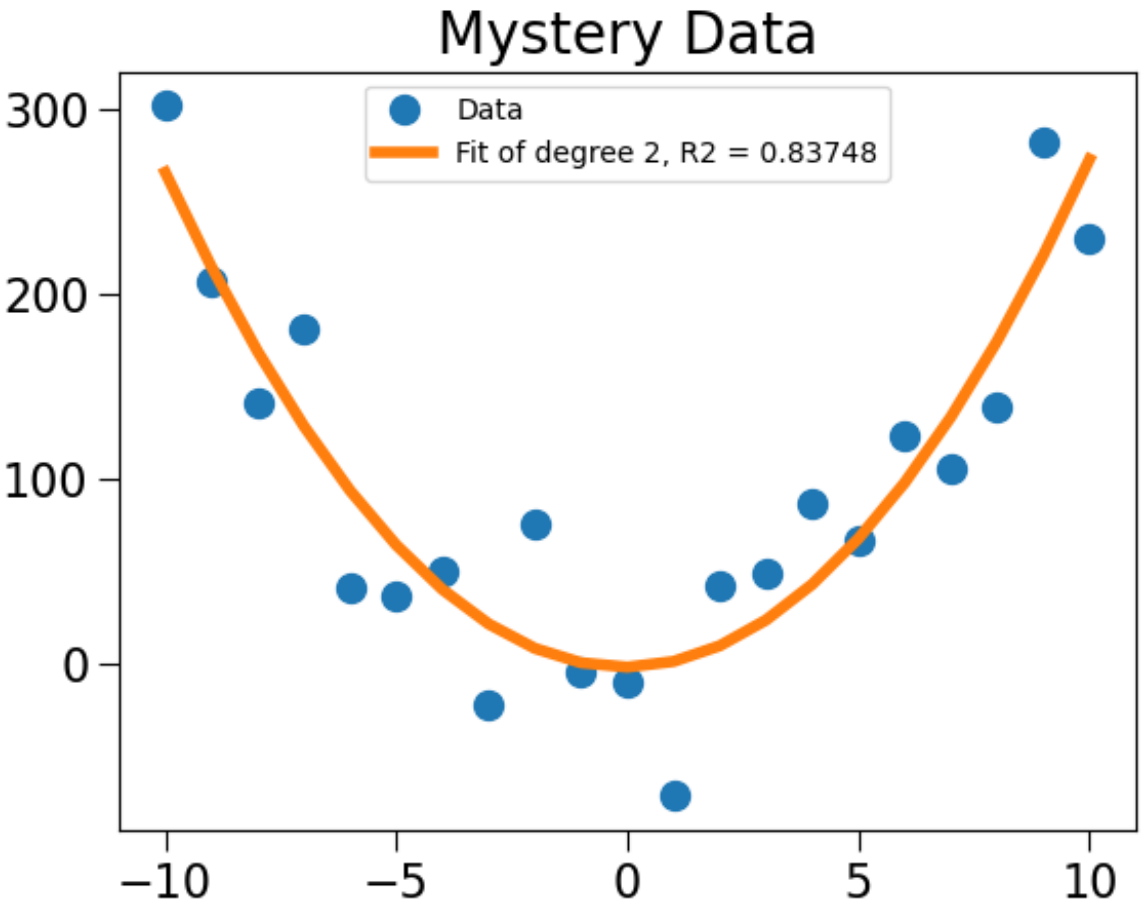
How Well Do Fits Explain Variance?



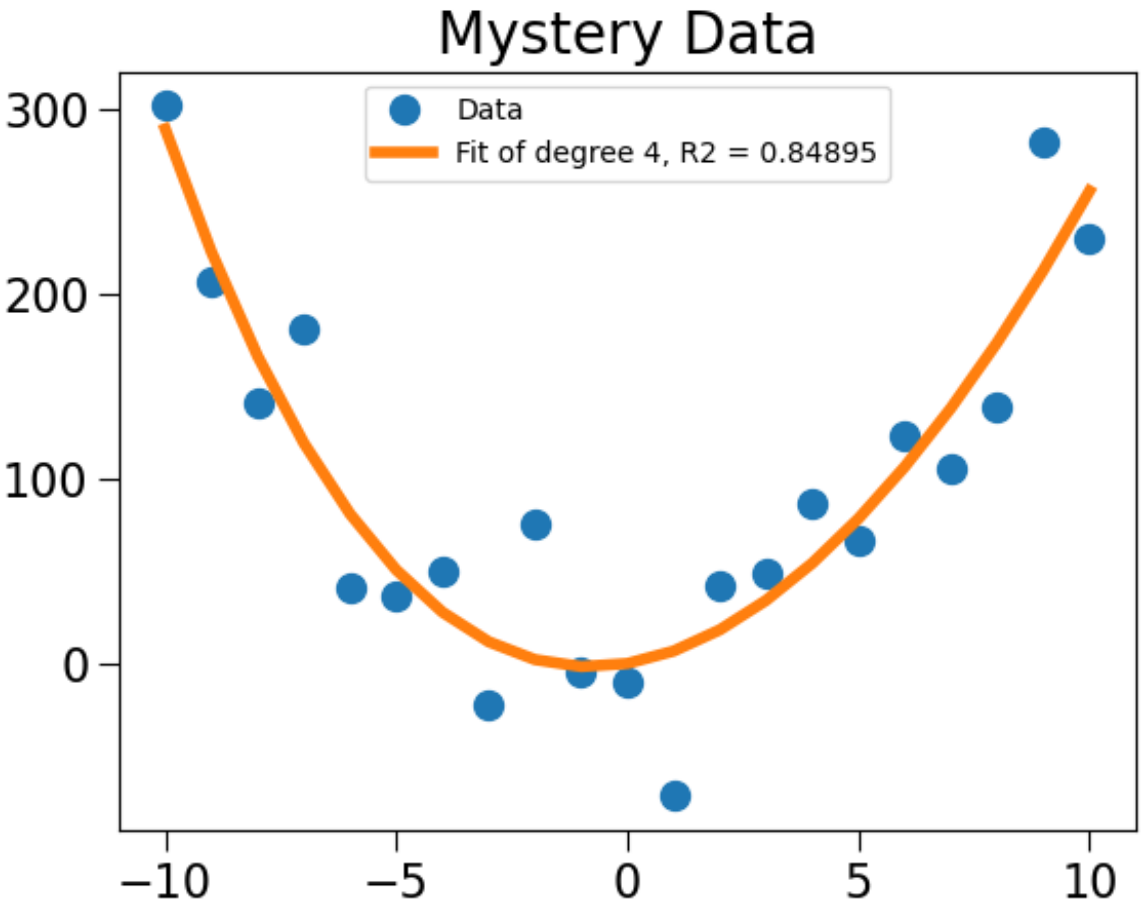
Can We Do Better?

- Saw that linear fit was poor – both visually and through R^2 measure
- Saw that quadratic fit was better – again both visually and through R^2 measure
- What about fitting higher order polynomials to data?
  - Degree 4?
  - Degree 8?
  - Degree 16?

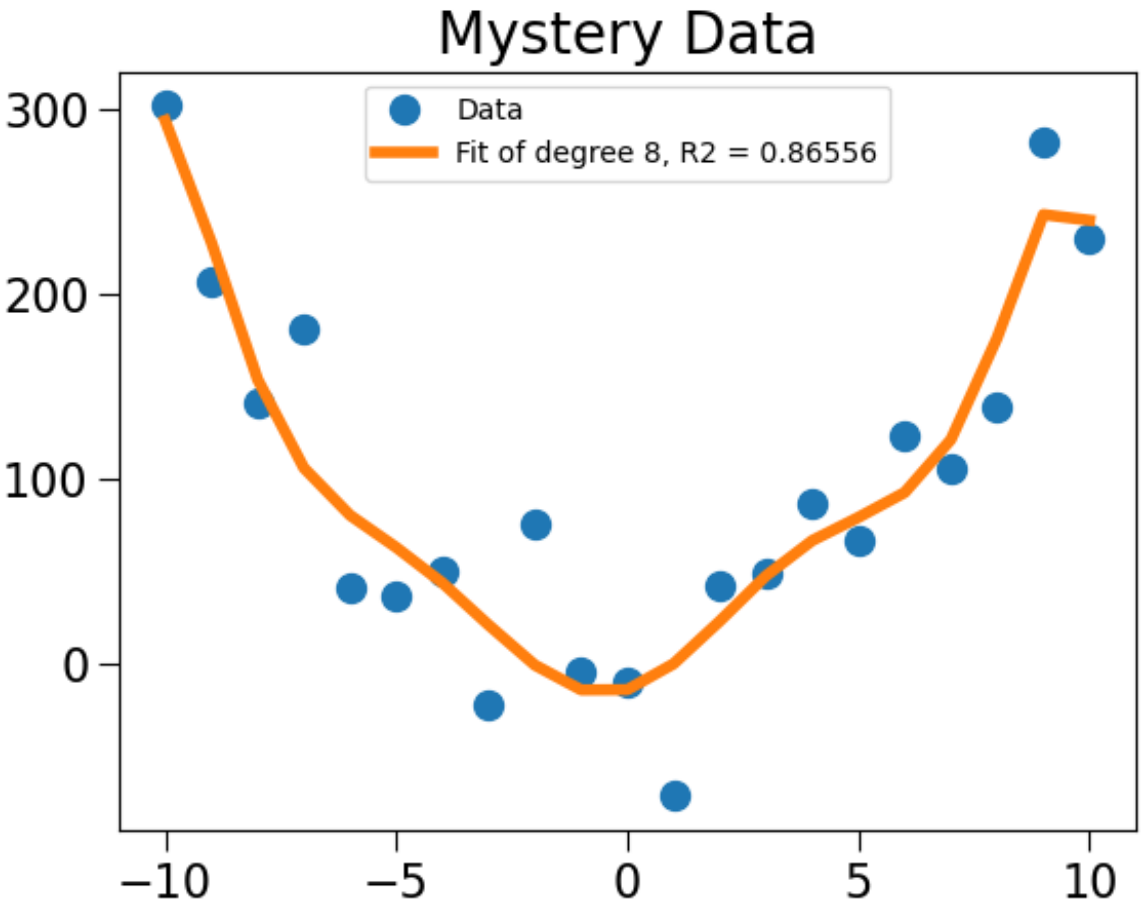
Order 2 Fit


Mystery Data
Data
Fit of degree 2, R2 = 0.83748
300
200
100
0
−10
−5
0
5
10

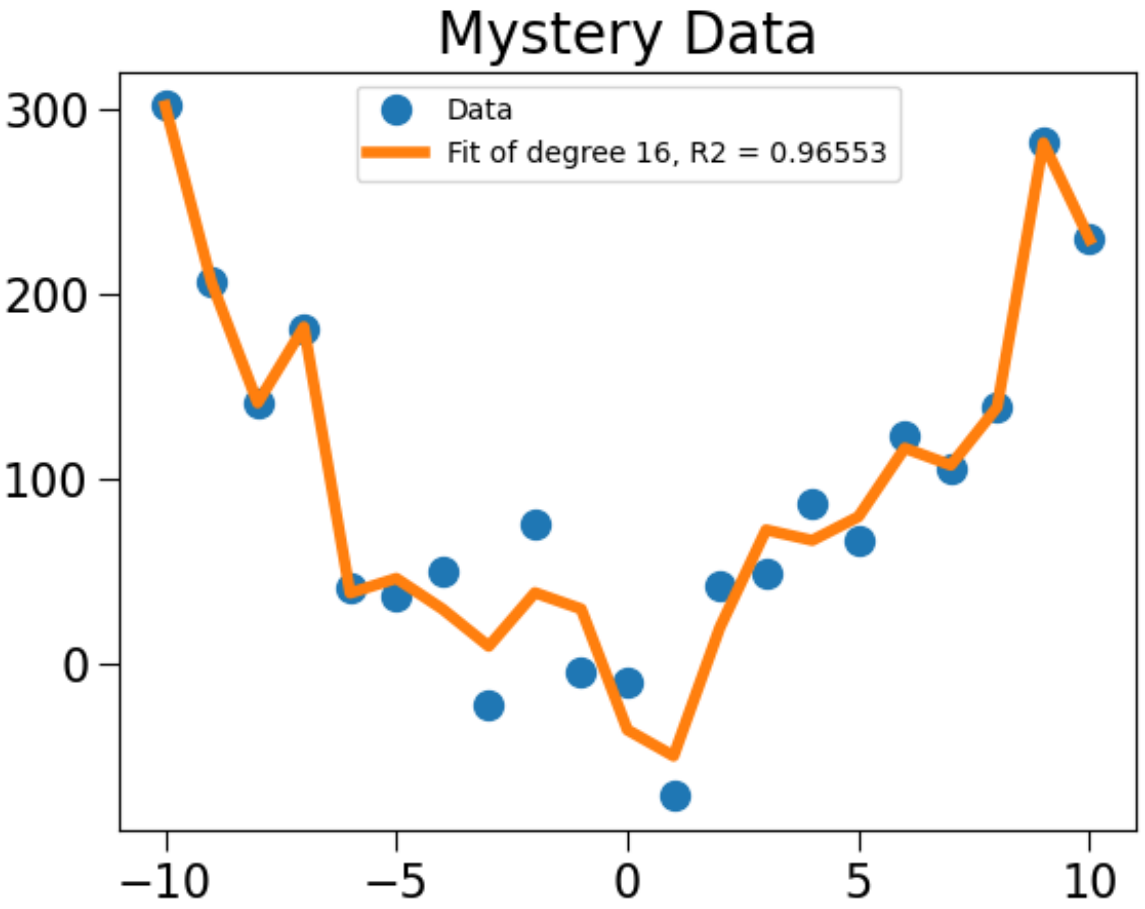
Order 4 Fit


Mystery Data
Data
Fit of degree 4, R2 = 0.84895
300
200
100
0
−10
−5
0
5
10

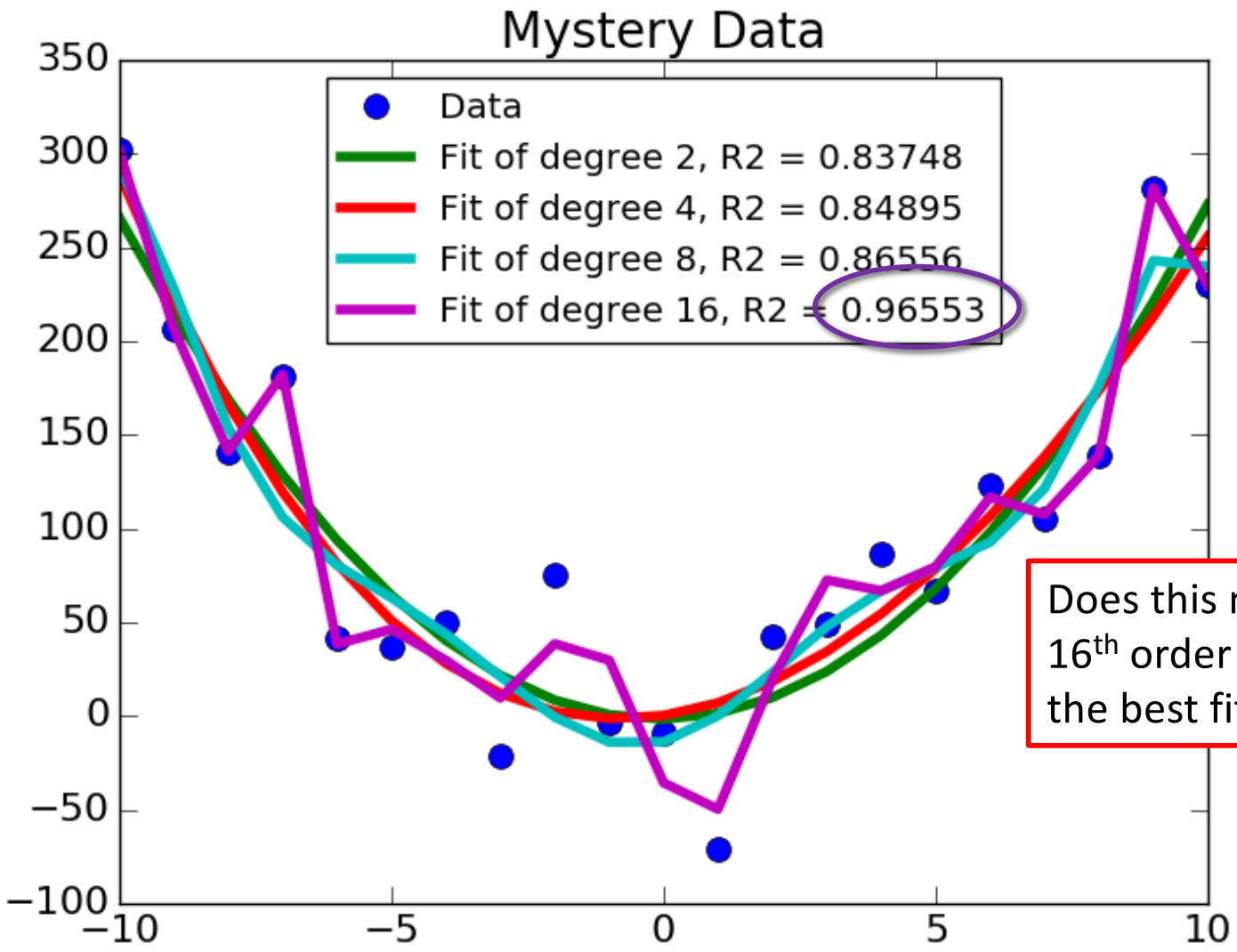
Order 8 Fit


Mystery Data
Data
Fit of degree 8, R2 = 0.86556
300
200
100
0
−10
−5
0
5
10

Order 16 Fit


Mystery Data
Data
Fit of degree 16, R2 = 0.96553
300
200
100
0
−10
−5
0
5
10

Can We Get a Tighter Fit?



Does this mean that a 16th order polynomial is the best fit for the data?

Does Tightest = Best?

- Looks like an order 16 fit is really good – so should we just use this as our model?
  - To answer, need to ask – **why build models in first place?**
- Help us understand process that generated the data
  - E.g., the properties of a particular linear spring
- Help us make predictions about out-of-sample data
  - E.g., predict the displacement of a spring when a force is applied to it
  - E.g., predict the effect of treatment on a patient
  - E.g., predict the outcome of an election
- A good model helps us do **both** of these things
  - Let's specifically look at using second property

How Mystery Data Was Generated

```
def gen_noisy_parabolic_data(a, b, c, x_vals, filename):
    y_vals = []
    for x in x_vals:
        theoretical_val = a*x**2 + b*x + c
        y_vals.append(theoretical_val + random.gauss(0, 35))
    with open(filename,'w') as f:
        f.write('y          x\n')
        for i in range(len(y_vals)):
            f.write(str(y_vals[i]) + ' ' + str(x_vals[i]) + '\n')

x_vals = range(-10, 11, 1)
a, b, c = 3, 0, 0
gen_noisy_parabolic_data(a, b, c, x_vals, 'mysteryData.txt')
```

Zero mean, Gaussian noise

How Mystery Data Was Generated

```
def gen_noisy_parabolic_data(a, b, c, x_vals, filename):
    y_vals = []
    for x in x_vals:
        theoretical_val = a*x**2 + b*x + c
        y_vals.append(theoretical_val + random.gauss(0, 35))
    with open(filename,'w') as f:
        f.write('y          x\n')
        for i in range(len(y_vals)):
            f.write(str(y_vals[i]) + ' ' + str(x_vals[i]) + '\n')

x_vals = range(-10, 11, 1)
a, b, c = 3, 0, 0
gen_noisy_parabolic_data(a, b, c, x_vals, 'mysteryData.txt')
```

Zero mean, Gaussian noise

If data was generated by quadratic, why was 16th order polynomial the "best" fit?

Because it fit the noise

Increasing the Complexity

- Is it just luck that we got a “better” fit on training data with higher order model?
- What happens when we increase order of polynomial during training?
  - Can we get a worse fit to training data?

Increasing the Complexity

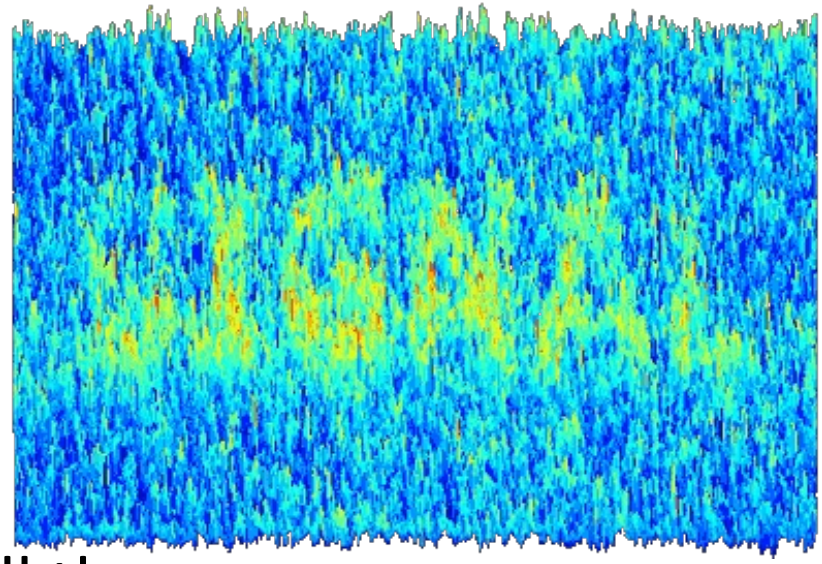
- Is it just luck that we got a "better" fit on training data with higher order model?
- What happens when we increase order of polynomial during training?
  - Can we get a worse fit to training data?
- If extra term is useless, coefficient will merely be zero

Interpolation Theorem:
An order ***n*** polynomial will perfectly fit
n+1 data points

- But if data is noisy, can fit the noise rather than the underlying pattern in the data
  - May lead to a "better" R^2 value, but not really a "better" fit
  - Might yield terrible predictions for unseen data – will look at this shortly

Validate the training

- One way to separate out impact of noise on model is to take advantage of fact that each time we sample a system
  - Signal will be roughly the same
  - Noise will be different



- Use set of data as a **"training"** set to fit a model
- Use a second set of data as a **"validation"** set, and see how well the model from the training set accounts for the validation set

Generate 2 Data Sets from Same Distribution

```
x_vals = range(-10, 11, 1)
a, b, c = 3, 0, 0
gen_noisy_parabolic_data(a, b, c, x_vals, 'parabola1.txt')
gen_noisy_parabolic_data(a, b, c, x_vals, 'parabola2.txt')
```

Generate two data sets

```
degrees = (1, 2, 16)
```

```
x_vals1, y_vals1 = get_data('parabola1.txt')
models1 = gen_fits(x_vals1, y_vals1, degrees)
test_fits(models1, degrees, x_vals1, y_vals1, 'Parabola 1')
```

Model first data set

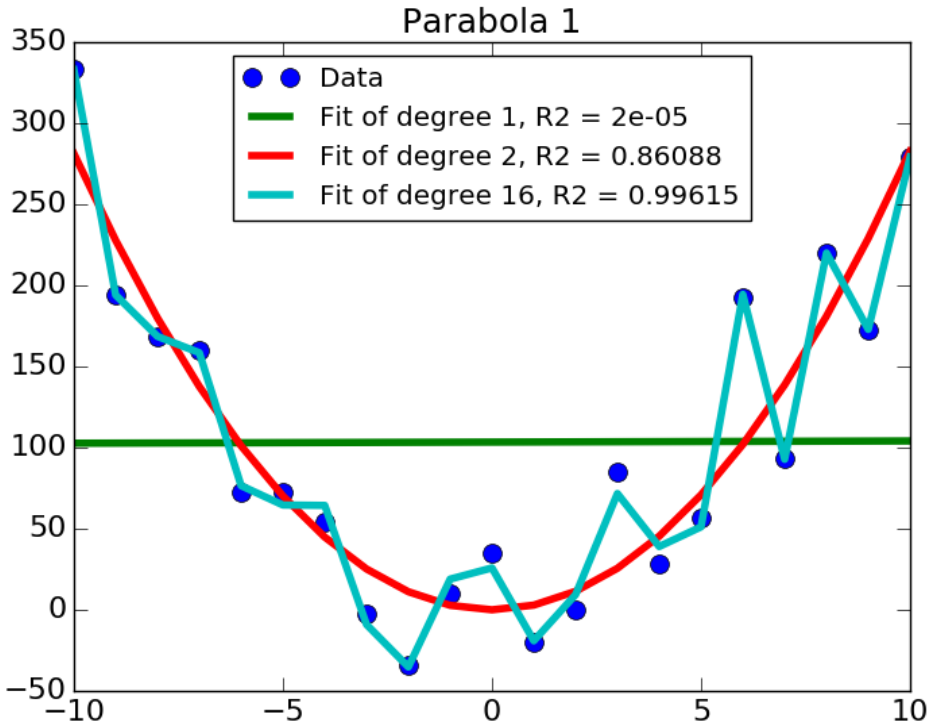
```
x_vals2, y_vals2 = get_data('parabola2.txt')
models2 = gen_fits(x_vals2, y_vals2, degrees)
test_fits(models2, degrees, x_vals2, y_vals2, 'Parabola 2')
```

Model second data set

Have two different models, one for each data set, where x_vals are the same, but y_vals differ

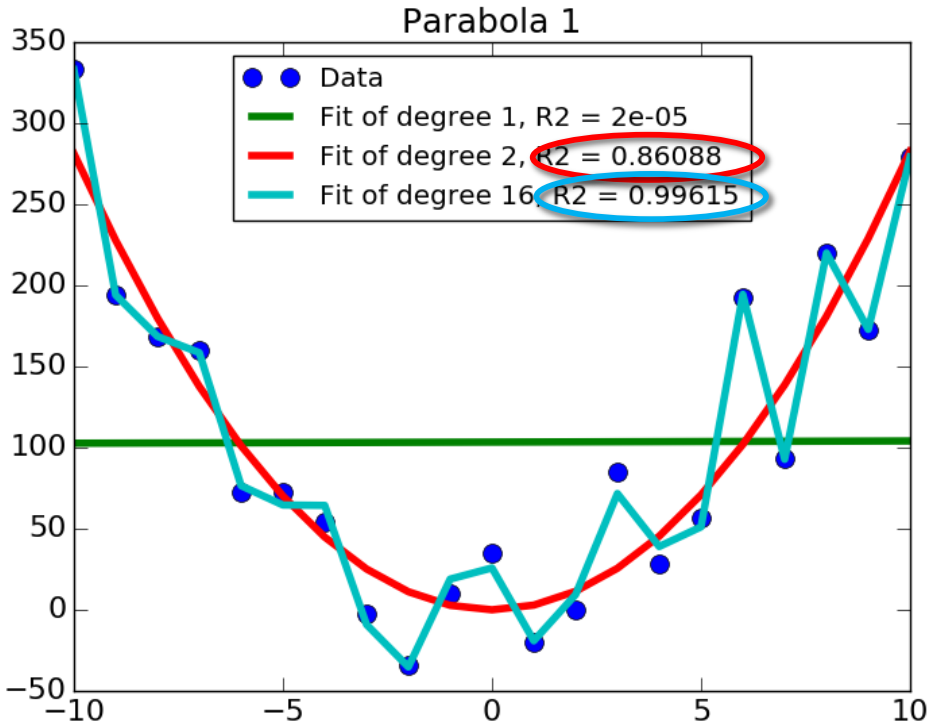
Training and Validation Errors

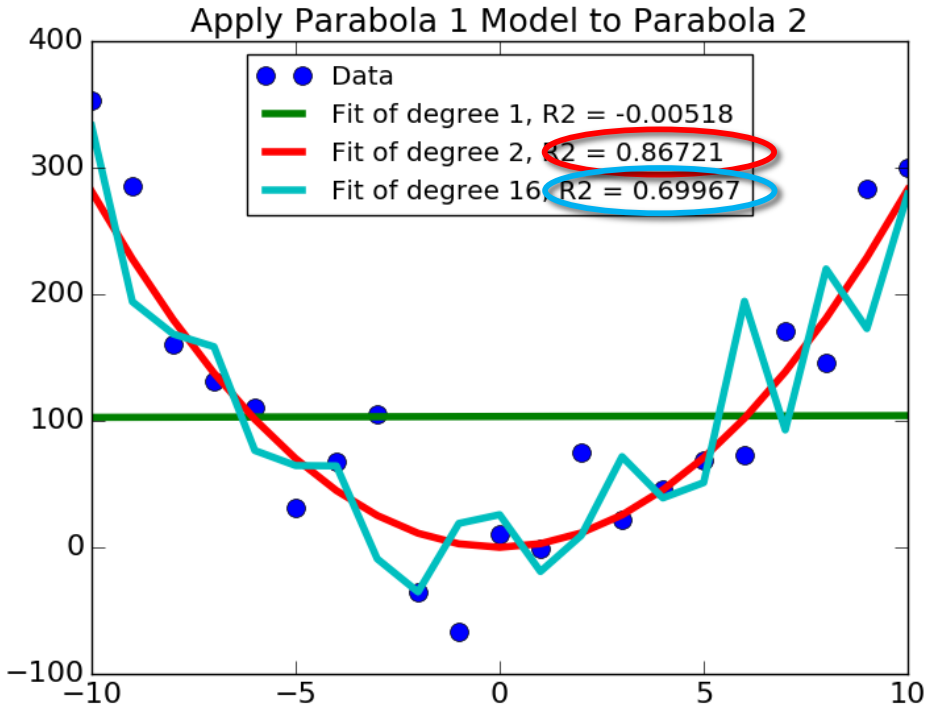
```
test_fits(models1, degrees, x_vals1, y_vals1, 'Parabola 1')
```



Training and Validation Errors

```
test_fits(models1, degrees, x_vals1, y_vals1, 'Parabola 1')

test_fits(models1, degrees, x_vals2, y_vals2,
          'Apply Parabola 1 Model to Parabola 2')
```

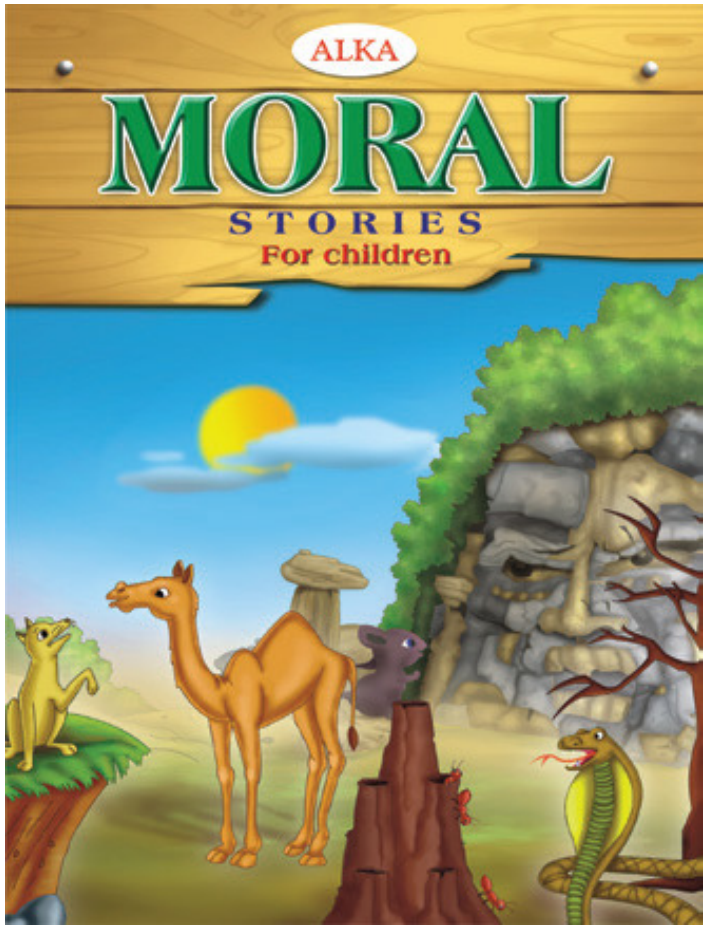




Now we see that the quadratic model actually is much better fit when applied to a new data set

The Moral of the Story

- 16-degree polynomial is an example of overfitting to the data
- If we only look at how well model fits training data, we may not detect that model is too complex
- Need to cross-validate: Train on one data set, then validate on a different set



If We Can't Run Another Experiment?

- If we only have one data set, an alternative way to cross-validate is to split the existing data:
  - A **training subset** (which we use to build a model)
  - A **validation subset** (on which we apply the model)
- Could split data evenly, or use more to train than to validate
- Ideally select validating data at random
  - So that it has the same "distribution" as the training data
  - Many ways of doing this

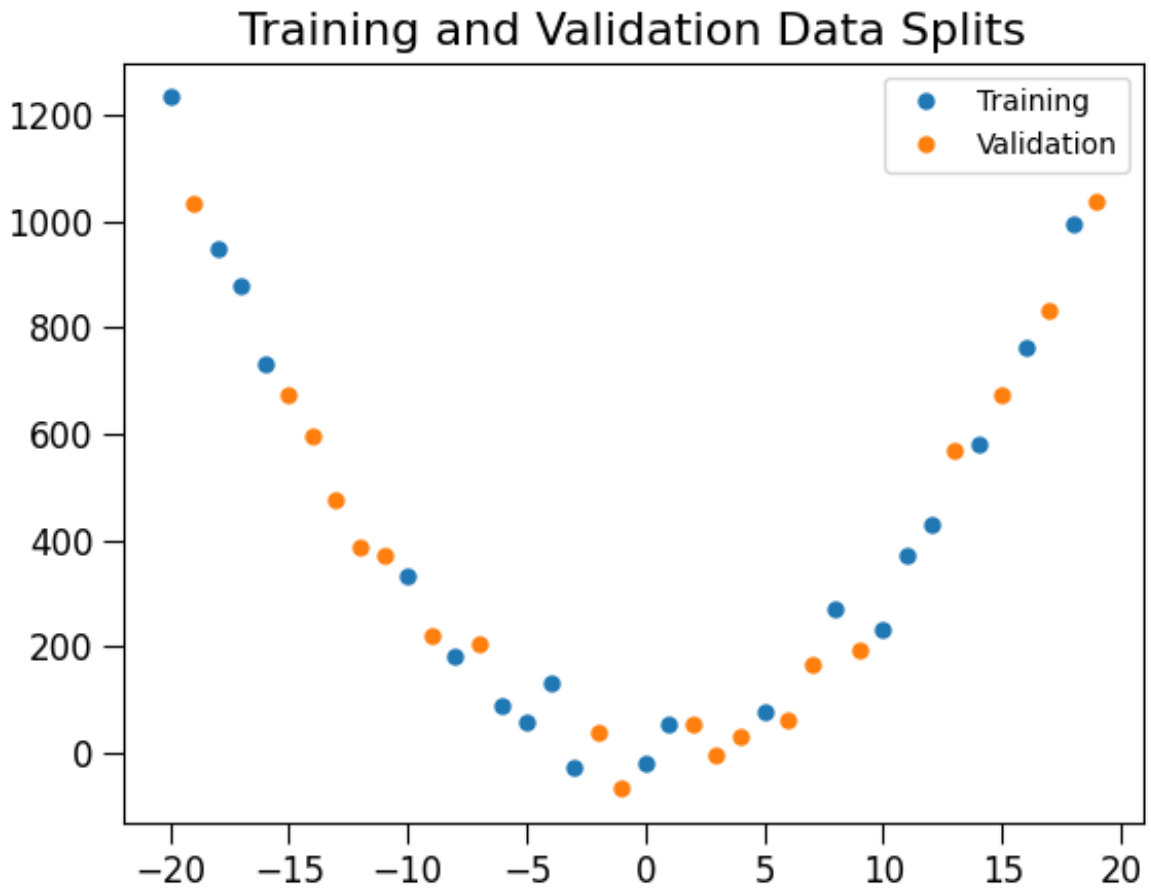
Split a Data Set into Train/Validate Subsets

```
def split_data(x_vals, y_vals, frac_training, plot=True):
    train_size = int(len(x_vals) * frac_training)
    train_indices = random.sample(range(len(x_vals)), train_size)
    train_x, train_y, validate_x, validate_y = [], [], [], []
    for i in range(len(x_vals)):
        if i in train_indices:
            train_x.append(x_vals[i])
            train_y.append(y_vals[i])
        else:
            validate_x.append(x_vals[i])
            validate_y.append(y_vals[i])
    if plot:
        plt.figure()
        plt.plot(train_x, train_y, '.', label='Training')
        plt.plot(validate_x, validate_y, '.', label='Test')
        plt.legend()
        plt.title('Training and Validation Data Splits')
    return (train_x, train_y), (validate_x, validate_y)
```

Sampling without replacement, unique indices

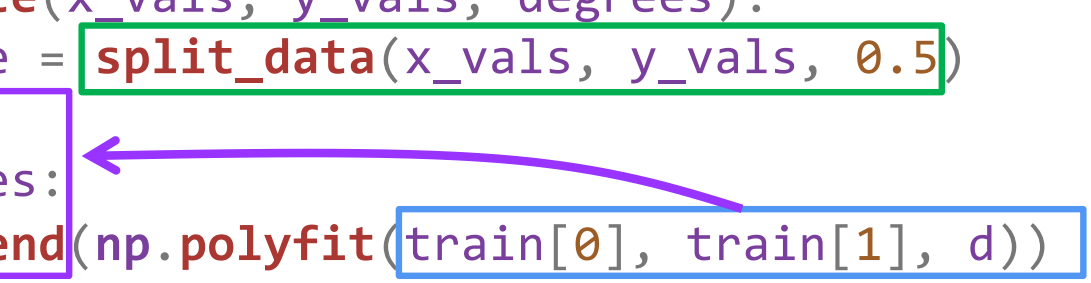
Partition based on training-selected indices

Split a Data Set into Train/Validate Subsets



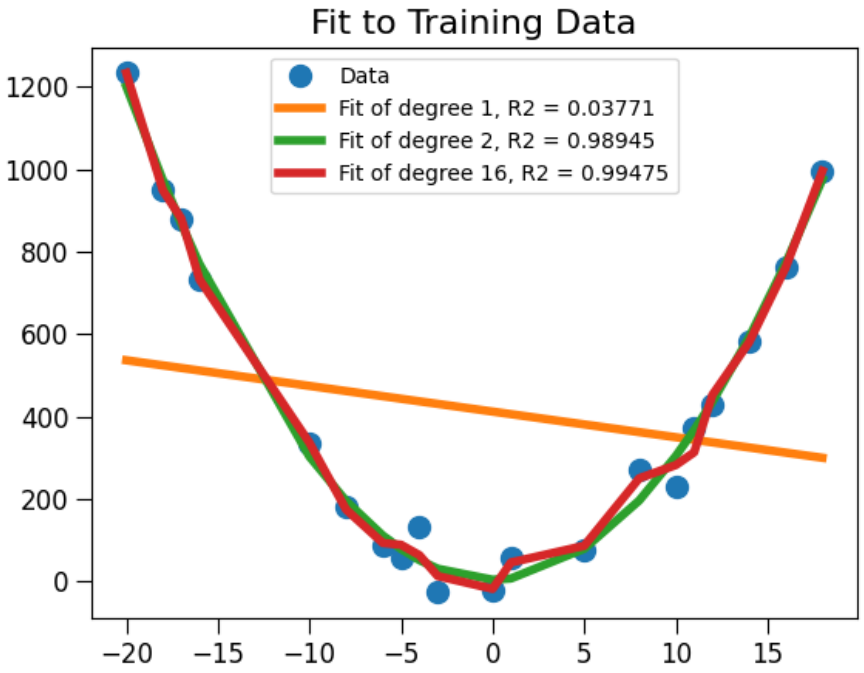
Cross-Validation on a Single Data Set

```
def fit_and_validate(x_vals, y_vals, degrees):
    train, validate = split_data(x_vals, y_vals, 0.5)
    models = []
    for d in degrees:
        models.append(np.polyfit(train[0], train[1], d))
    for m in models:
        print([round(coeff, 2) for coeff in m])

    evaluate_fits(models, degrees, train[0], train[1],
                  'Fit to Training Data')

    evaluate_fits(models, degrees, validate[0], validate[1],
                  'Applied to Validation Data')
```



```
x_vals, y_vals = get_data('parabola.txt')
degrees = (1, 2, 16)
fit_and_validate(x_vals, y_vals, degrees)
```

Validating a Model

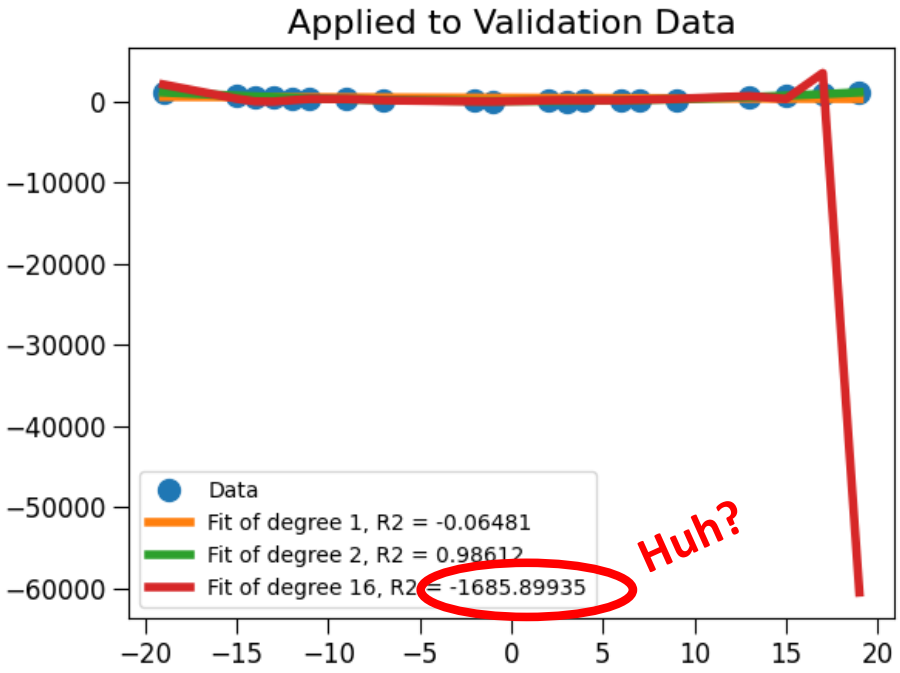


degree = 2 model

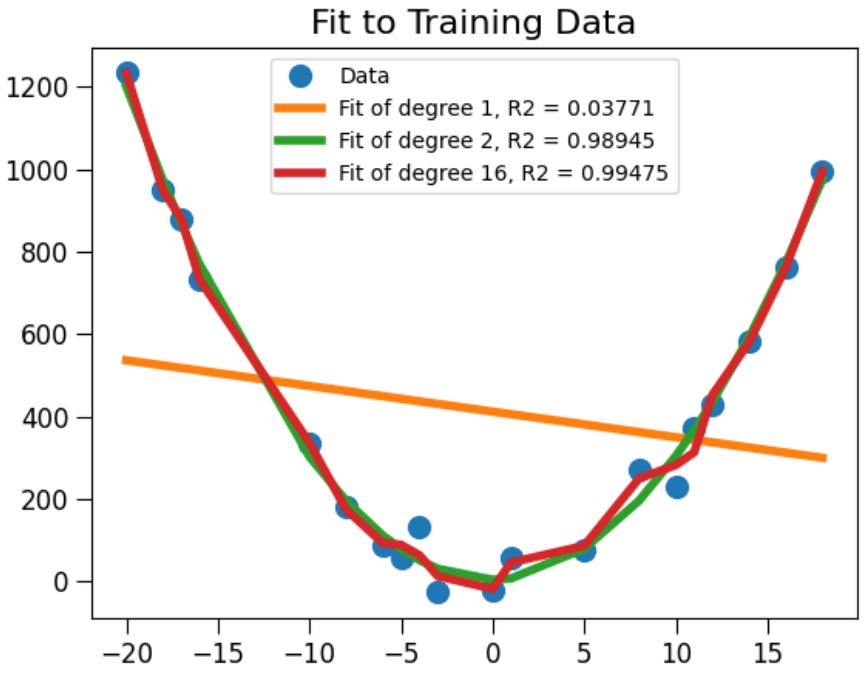
$y = 2.99x^2 + 0.07x + 4.11$

degree = 16 model

$y = -0.03x^6 + 0.23x^5 - 0.95x^4 - 5.82x^3 + 15.94x^2 + 54.55x - 17.95$



Validating a Model

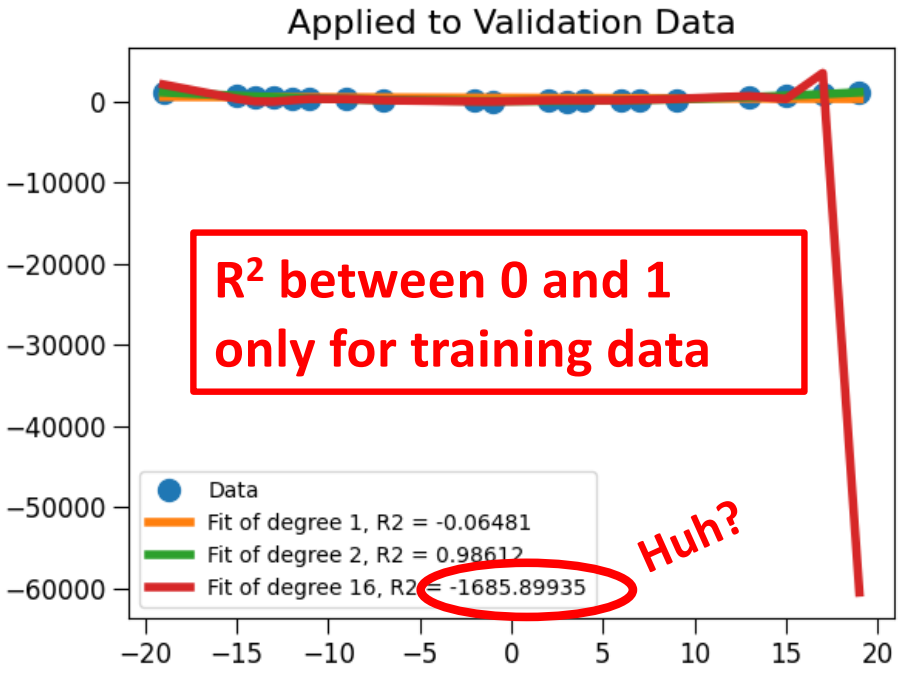


degree = 2 model

$y = 2.99x^2 + 0.07x + 4.11$

degree = 16 model

$y = -0.03x^6 + 0.23x^5 - 0.95x^4 - 5.82x^3 + 15.94x^2 + 54.55x - 17.95$

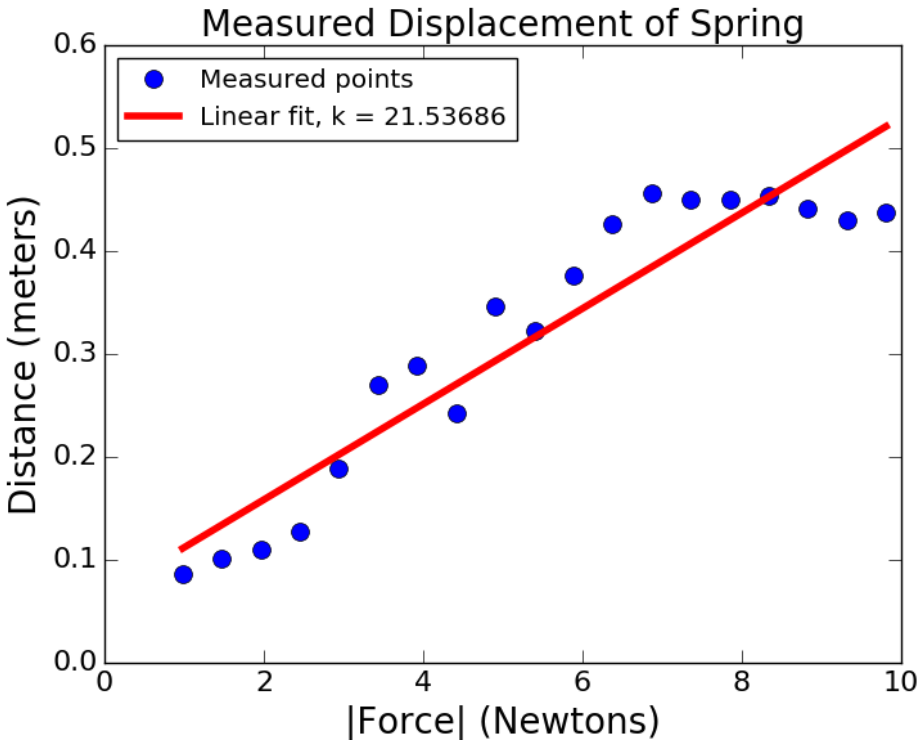


Final evaluation after validation

- Remember the original purpose of fitting a curve to data is to obtain a model that can **predict and explain unseen examples**
- Prepare another ***test data set*** for evaluating the chosen model
  - Same evaluation procedure: apply model to data and evaluate R^2 or related metric
  - But only apply the best validated model
- Summary of data sets
  - **Training data** is for fitting the parameters of a model
  - **Validation data** is for finding the most appropriate model structure
  - **Testing data** is for evaluating the chosen model's predictive power on the data/phenomenon being observed

One Last Thought

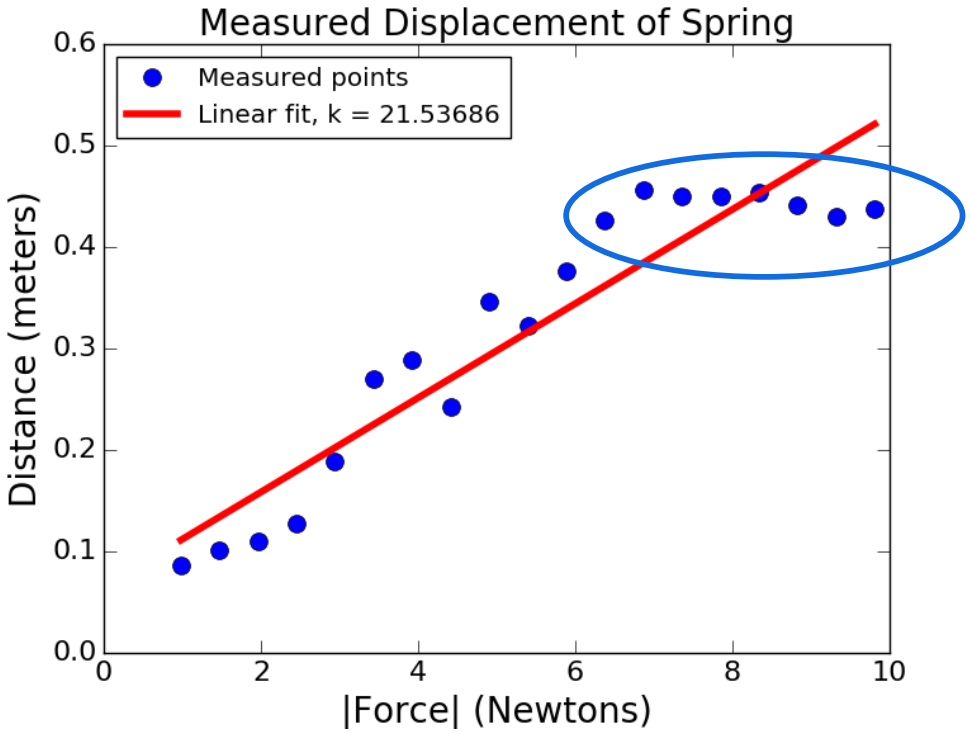
- Combining model information with goodness of fit can provide additional insight
- Consider the fit to the original spring data



- R^2 value is .8815 – which is decent

One Last Thought

- But visual inspection suggests that something might be happening for large forces?

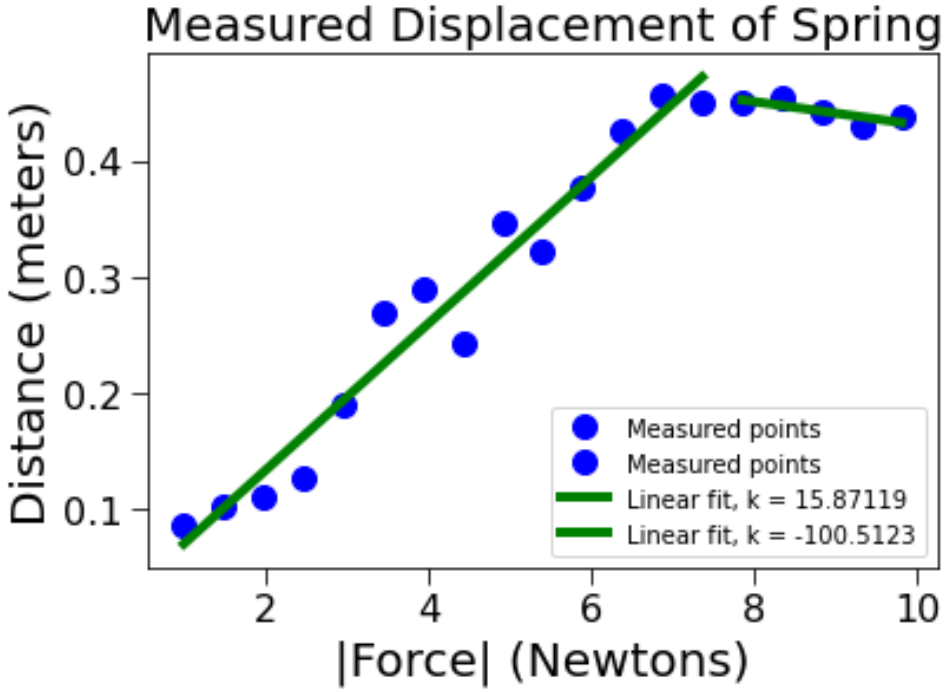


Is there something going on here?

- Remember theory said Hooke's law applied up to some maximum force
  - Have you ever stretched a slinky too far?

One Last Thought

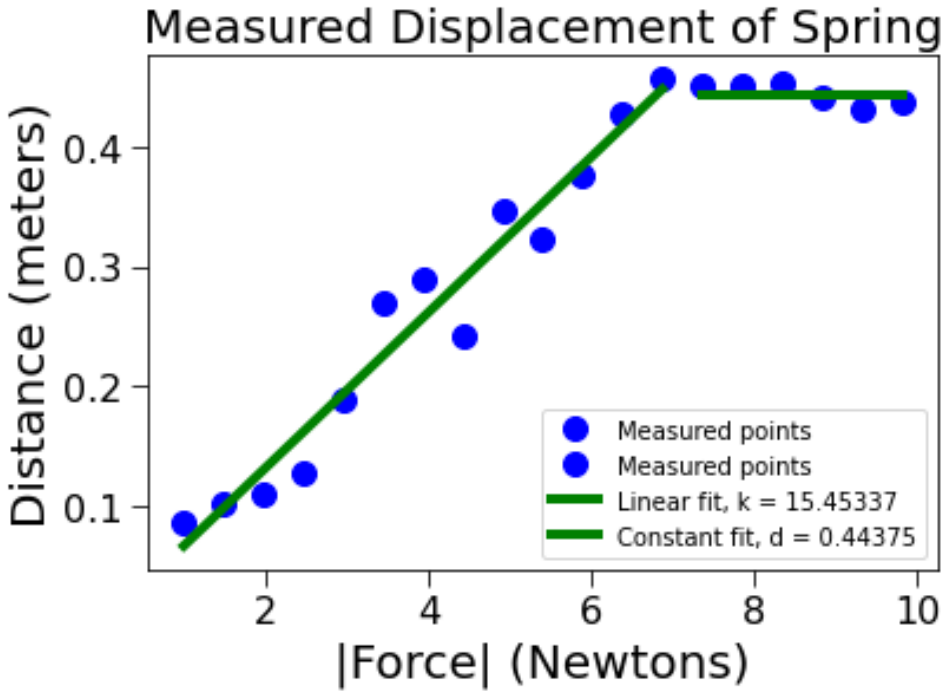
- Could search for **point at which to break data into two modes**, and fit models to both subsets separately; look for break that minimizes sum of residual error in both parts



- R^2 value for first part now .9581; for second part .6784; without break, have R^2 of .8815

One Last Thought

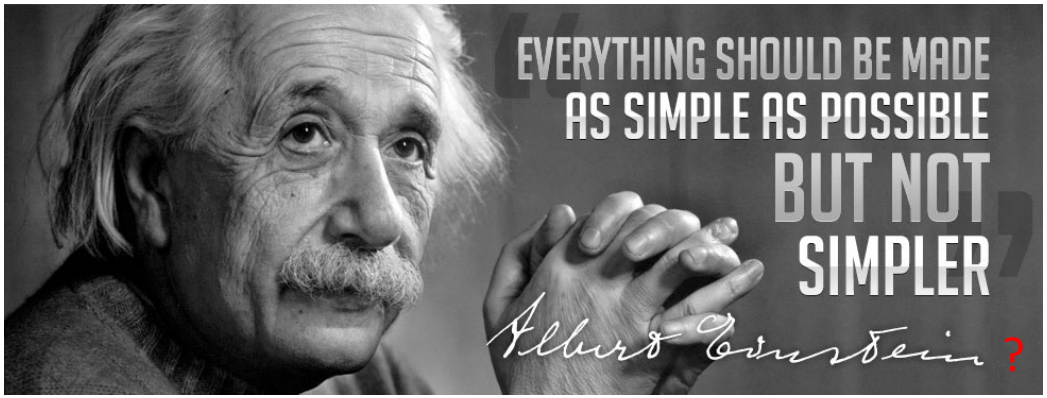
- Alternatively, search for point at which to break data, and fit model to first set **but fit constant line to second set**; look for break that minimizes sum of residual error in both parts



- R^2 value for lower part now .9539; without break, have R^2 of .8815

Take-home message

- We can use linear regression to fit a curve to data
  - Mapping from independent values to dependent values
- That curve is a model of the data that can be used to predict the value associated with independent values we haven't seen (out-of-sample data)
- R-squared used to evaluate model
  - Higher not always "better" because of risk of over fitting
- Choose complexity of model based on
  - Theory about structure of data
  - Cross validation
  - Simplicity



https://quoteinvestigator.com/2011/05/13/einstein-simple/