[1]:
```
%matplotlib inline
# make figures better for projector:
import matplotlib
font = {'size':18}
matplotlib.rc('font', **font)
matplotlib.rc('figure', figsize=(9.0, 6.0))

import warnings
warnings.filterwarnings('ignore')
```

[2]:
```
%config InlineBackend.figure_format = 'retina'
```

DS1 Lecture ~~10~~ 11

Jim Bagrow

Last time:

1. Data cleaning:
   - Rejecting bad data, combining data, filtering and processing data
2. Histograms as data "microscopes"
   - See the distribution of the data
   - Binning is key!
   - "broad" and log-vs-linear scales

Today's plan:

1. Scatter plots

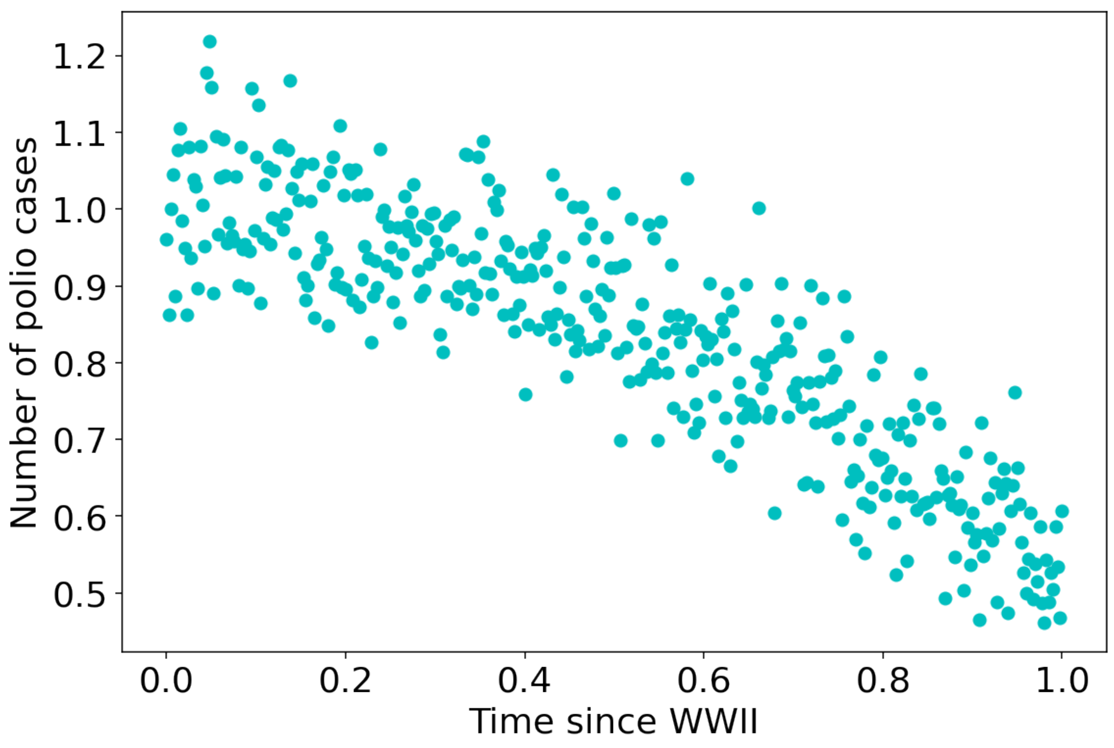
XY-data - Scatter plots

Often times you have pairs of (x, y) values. We've all seen plots of functions like $y = \cos(x)$. That's exactly what this is.

- When exploring data and looking at *pairs* of variables

When you have a set of "paired" data (or what I call XY-data), and you want to see if they exhibit a **trend** or are otherwise **related**, bust out the trusty old

[3]:
```
# Let's gussy up some fake data:

# 400 points evenly spaced in $[0,1]$:
X = np.linspace(0,1,400)

# y = cos(x) + some small noise:
Y = np.cos(X) + 0.075*(np.random.randn(*X.shape))

plt.plot(X,Y, 'o', color='c')

plt.xlabel("Time since WWII")
plt.ylabel("Number of polio cases") # not really
plt.show()
```



Just a quick scatter plot and we immediately see a nice **decreasing** trend.

- As simple as this is, this is one of the **fastest** and **most powerful** ways to explore your data.
- The two biggest tools in your data science toolbelt: histograms and scatter plots.

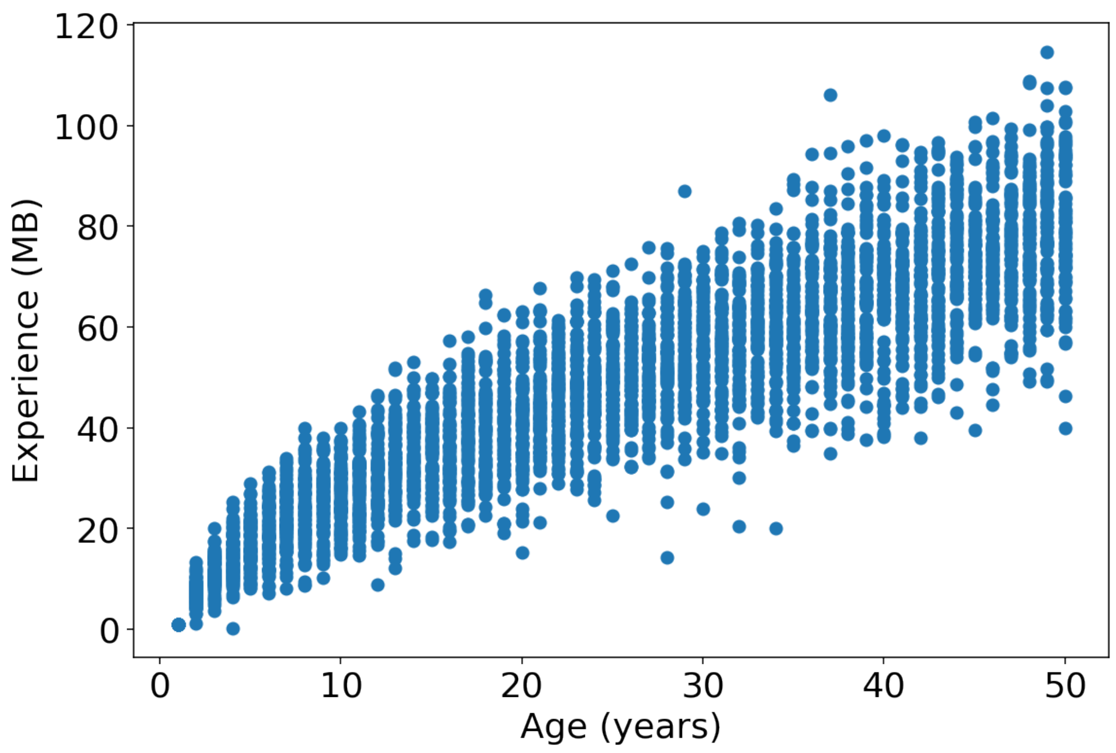
Some of the ideas behind **histograms** for X-data also translate well to working with scatter plots.

- We've got some noisy scatter. Let's compute a **smoother trendline** by averaging.

There's many ways to do this, a simple way is a **generalization** of the histogram.

If your x-values are discrete:

[4]:
```
# MOAR fake data:
Xs = []
Ys = []

for x in range(1,50+1):
    for _ in range(75):
        y = x + 3*np.log(x)*(np.random.randn()+2.5)
        Xs.append(x)
        Ys.append(y)

plt.plot(Xs,Ys,'o')
plt.xlabel("Age (years)")
plt.ylabel("Experience (MB)")
plt.show()
```

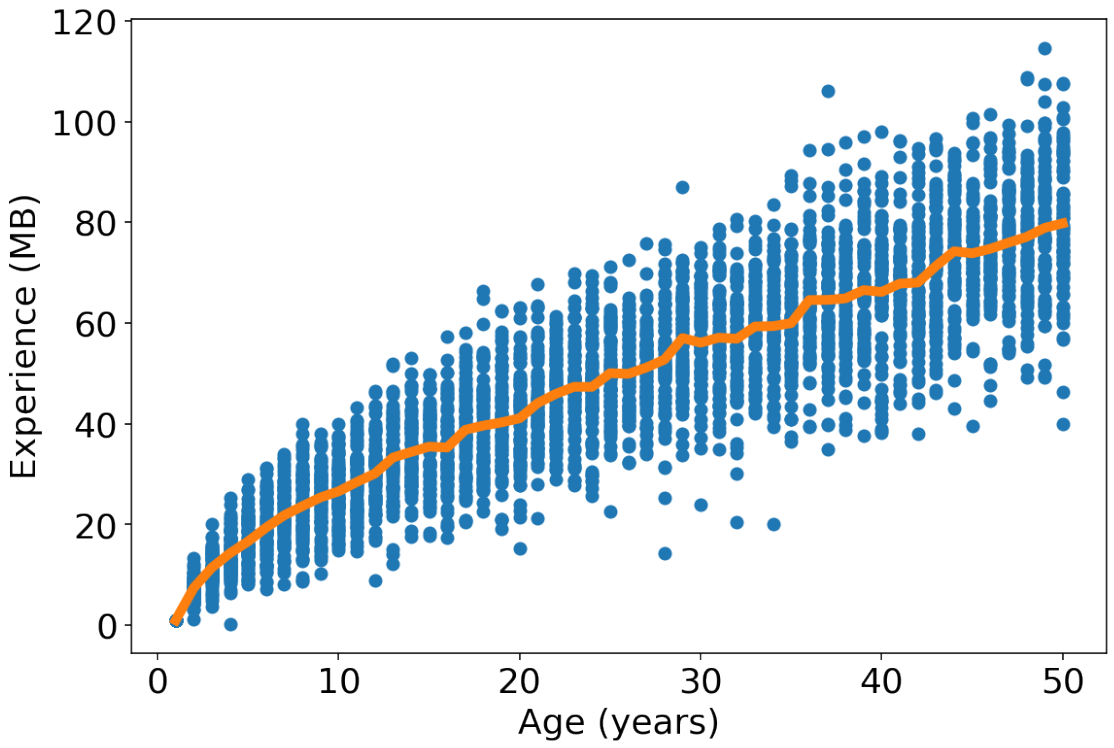


Then you can "bin" the *y*-data based on each unique *x*-value:

- Let's take each unique value of x and make a list of all corresponding y-values, then compute the **mean** of each list:

[5]:
```
# build list of all y's for each x:
x2listY = {}
for x,y in zip(Xs,Ys):
    try:
        x2listY[x].append(y)
    except KeyError:
        x2listY[x] = [y]

# compute mean of each list, break into separate lists
# for plotting:
Xs_line = sorted(x2listY.keys())
Ys_line = []
for x in Xs_line:
    corresponding_ys = x2listY[x]
    meanY = np.mean(corresponding_ys)
    Ys_line.append(meanY)

# faster/shorter/less easy to understand
#x2meanY = { x : np.mean(x2listY[x]) for x in x2listY }
#x_meanY = sorted(x2meanY.items())
#Xs_line,Ys_line = zip(*x_meanY)
```

```python
# plot averaged trend on top of "raw" scatter:
plt.plot(Xs,Ys, 'o')
plt.plot(Xs_line,Ys_line, '-', linewidth=5)
plt.xlabel("Age (years)")
plt.ylabel("Experience (MB)")
plt.show()
```



(This is also a great time for a `groupby` if you are so inclined.)

(And of course, in practice we wouldn't repeat these code blocks, we would update the previous code block to include the trend line. These code blocks are *unpacked* for the lecture.)

If your x-values are continuous
(or the data are noisy or you only have few points) you can **bin them** just like a histogram.

Let's take all the pairs of data and **bin them** by their x-value. Then instead of counting how many pairs fall into a bin (which would be a histogram) let's compute the means of their y-values.

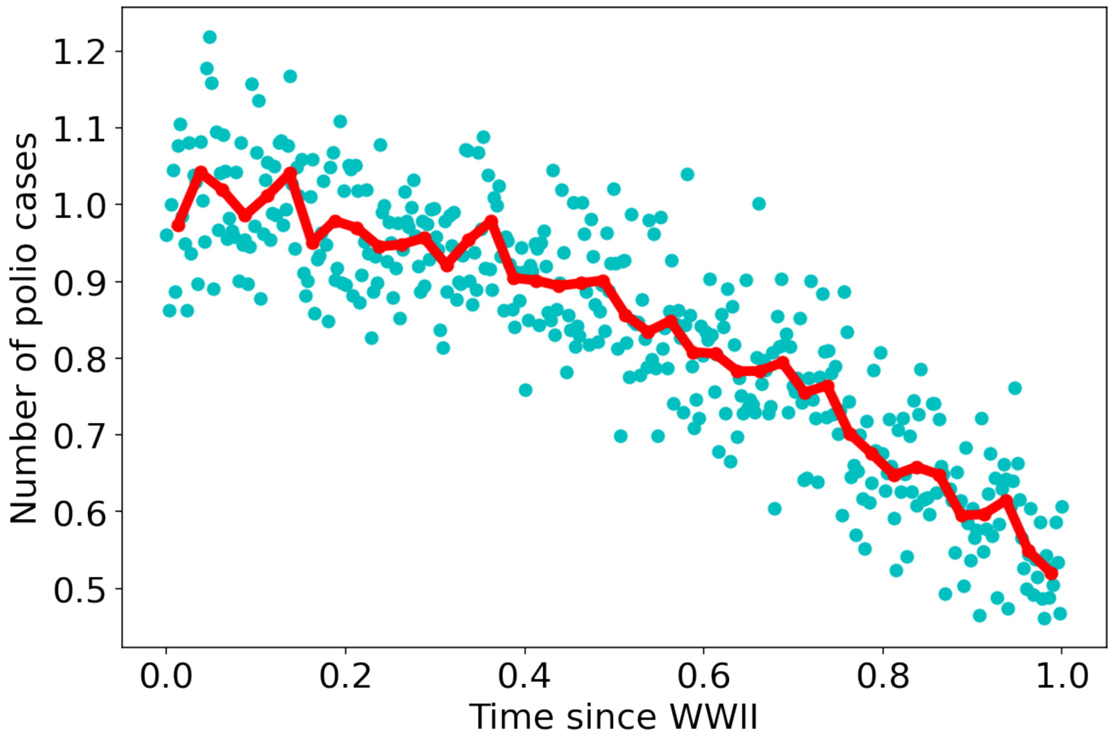
In *pseudocode*, for one bin:

```python
data = [(x1,y1),(x2,y2), ...]

this_bin_l = 0.0
this_bin_r = 0.1

y_values_for_this_bin = []
for x,y in data:
    if this_bin_l <= x < this_bin_r:
        y_values_for_this_bin.append(y)
```

```
x_this_bin = (this_bin_l + this_bin_r)/2 # bin center
y_this_bin = mean(y_values_for_this_bin)

# repeat for all bins
```

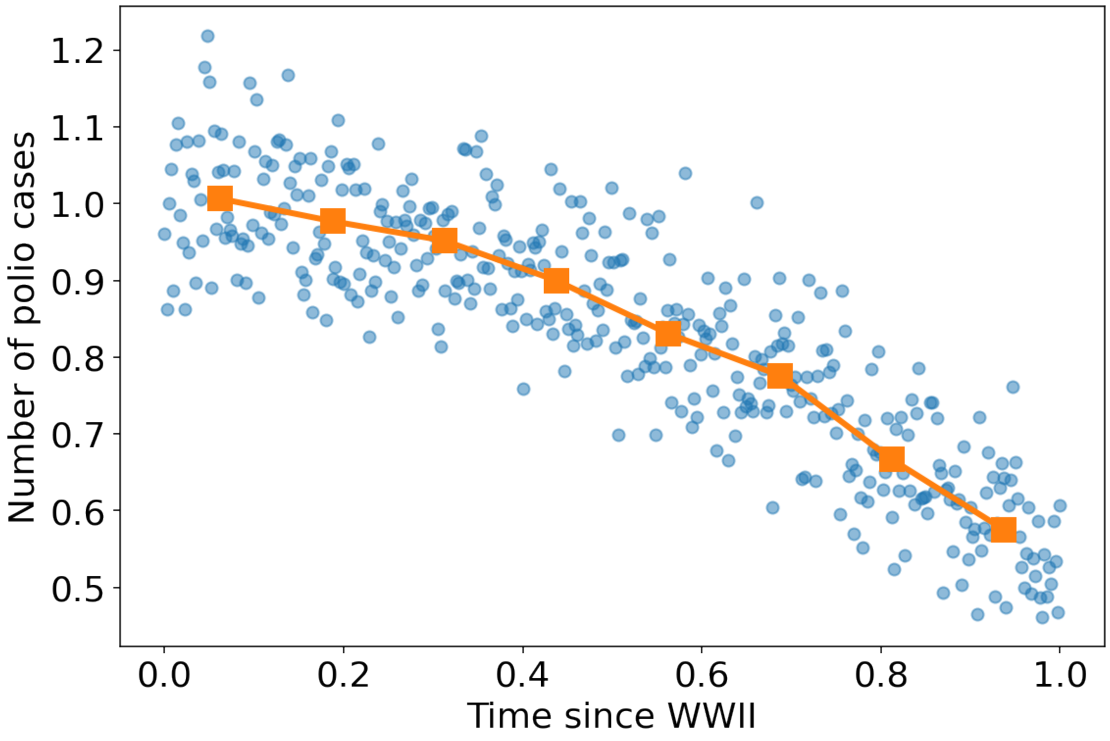
And to do this **for real**:

```
[6]: import scipy.stats

binned_stat = scipy.stats.binned_statistic

# nice builtin function!
y_bins,bin_edges, misc = binned_stat(X,Y, statistic="mean", bins=40)

# but this function doesn't return the bin CENTERS:
x_bins = (bin_edges[:-1]+bin_edges[1:])/2


plt.plot(X,Y, 'co')
plt.plot(x_bins, y_bins, "ro-", linewidth=5)


plt.xlabel("Time since WWII") # fake!
plt.ylabel("Number of polio cases")
plt.show()
```



OK, that's a little noisy still, let's **dial back** the number of bins...

[7]:
```
y_bins,bin_edges, misc = binned_stat(X,Y, statistic="mean", bins=8)
x_bins = (bin_edges[:-1]+bin_edges[1:])/2



plt.plot(X,Y, 'o', alpha=0.5)
plt.plot(x_bins, y_bins, "s-", linewidth=3, markersize=12)

plt.xlabel("Time since WWII")
plt.ylabel("Number of polio cases")
plt.show()
```



There exist many other techniques, especially when the x-values are NOT noisy, but this works well in many situations.

- **Moving** (or rolling) **averages** are also common, although they can sometimes introduce artifacts at the edges of a dataset's domain, especially if data are sparse there.
  - I prefer a LOWESS fit over a moving average when possible.
- It's especially important when there's so many points you just see a cloud of data. When the points are **so dense they overlap**, you can't see the **underlying density** with a scatterplot, as we'll see below.

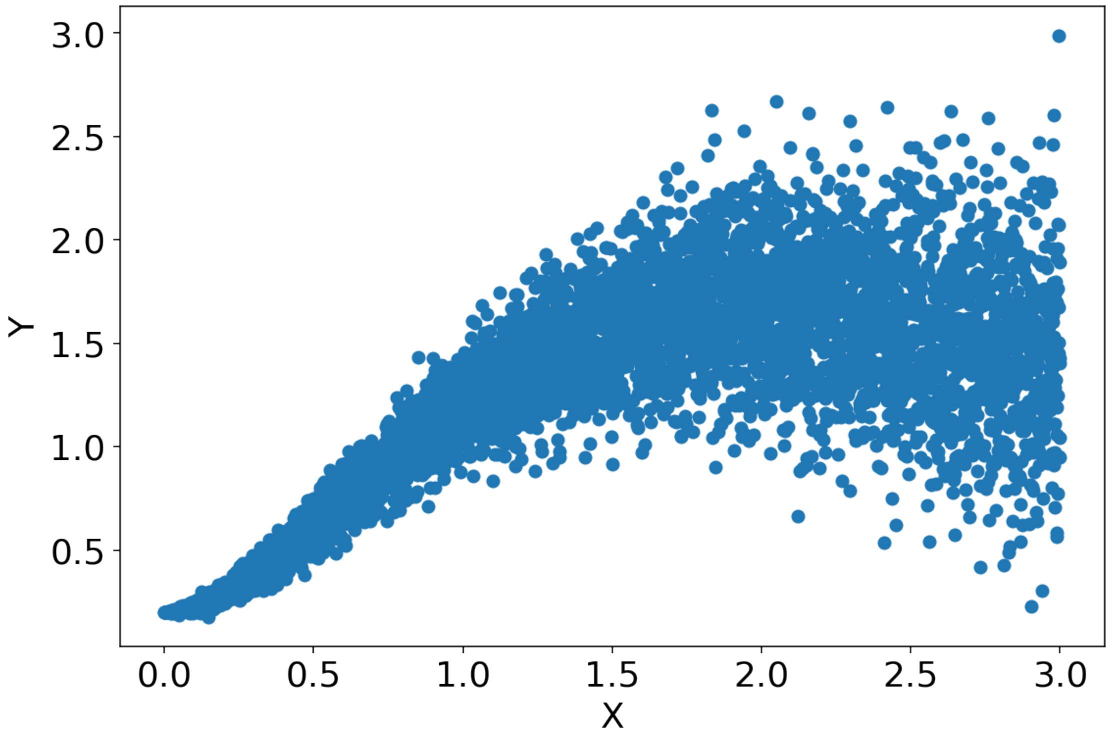
Binning on the x also lets us use our one-dimensional data analysis **toolbox** repeatedly on small segments of the y-data:

- When the x-values are discrete we can look at all the **unique** x-values.
- We can compute the mean of all y-values in an x-bin, but we can also use the *median*, *percentiles*, or really any measure of *central tendency*.

Error bars: measures of the *spread* of the y-values in a bin can tell us how broad the data are.

First, let's make a scatter plot:

[8]:
```
n = 5000
x = np.linspace(0,3,n)

x_data = x
y_data = np.exp(-x+1)*x**2 + 0.15*(np.random.randn(n))*x + 0.2

plt.plot(x_data,y_data, 'o')

plt.xlabel("X")
plt.ylabel("Y")
plt.show()
```

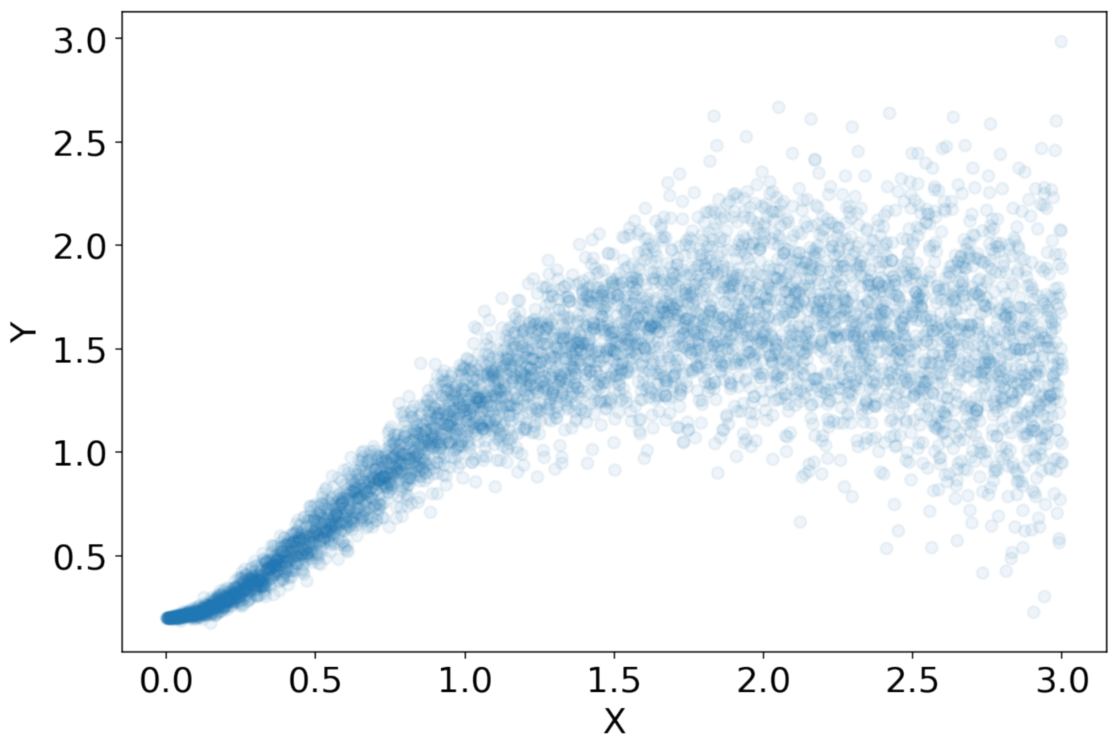


Notice how the data spread out more as x increases?

Also notice how it's hard to see what's going on inside the plot—it's just a wall of points.

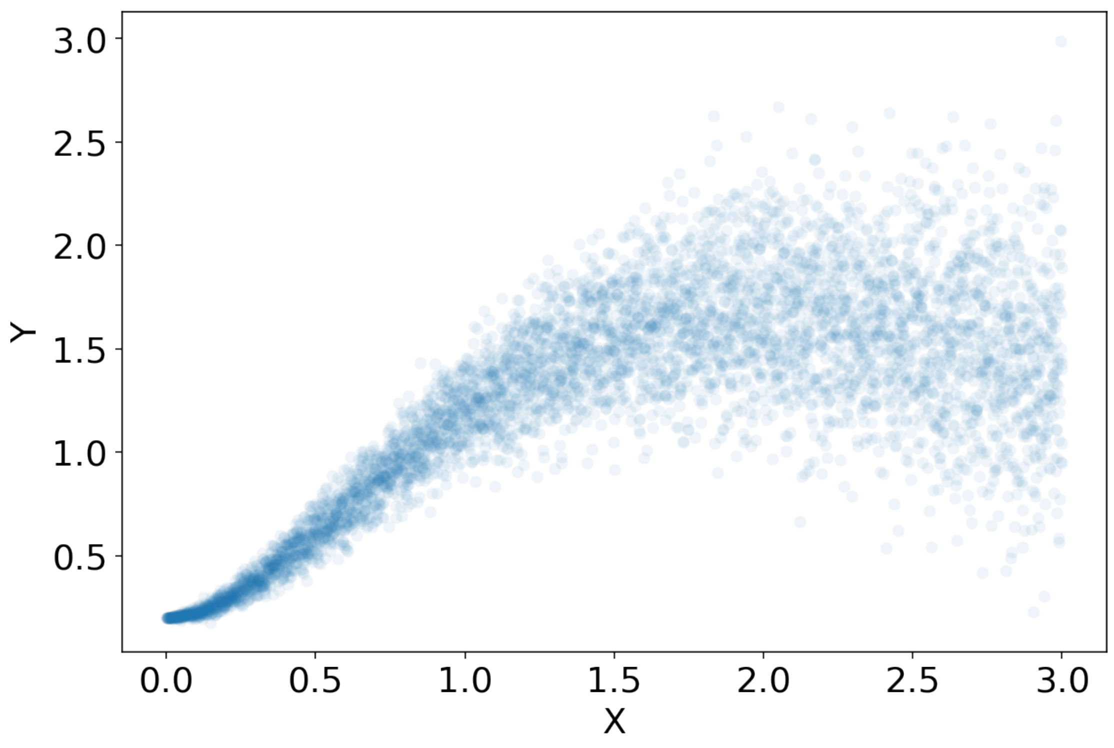
- A bit of ***transparency*** can sometimes help with so much scatter overlap:

[9]:
```
plt.plot(x_data,y_data, 'o', alpha=0.075)
#                            ^^^^^^^^^^
plt.xlabel("X")
plt.ylabel("Y")
plt.show()
```



Keep in mind that markers have *edges* and *faces* and sometimes `alpha` (in this case) will affect one and not the other. Let's turn off the marker edges using a special matplotlib command:

[10]:
```
plt.plot(x_data,y_data, 'o', alpha=0.075, mec='none')
#                                         ^^^^^^^^^^
plt.xlabel("X")
plt.ylabel("Y")
plt.show()
```

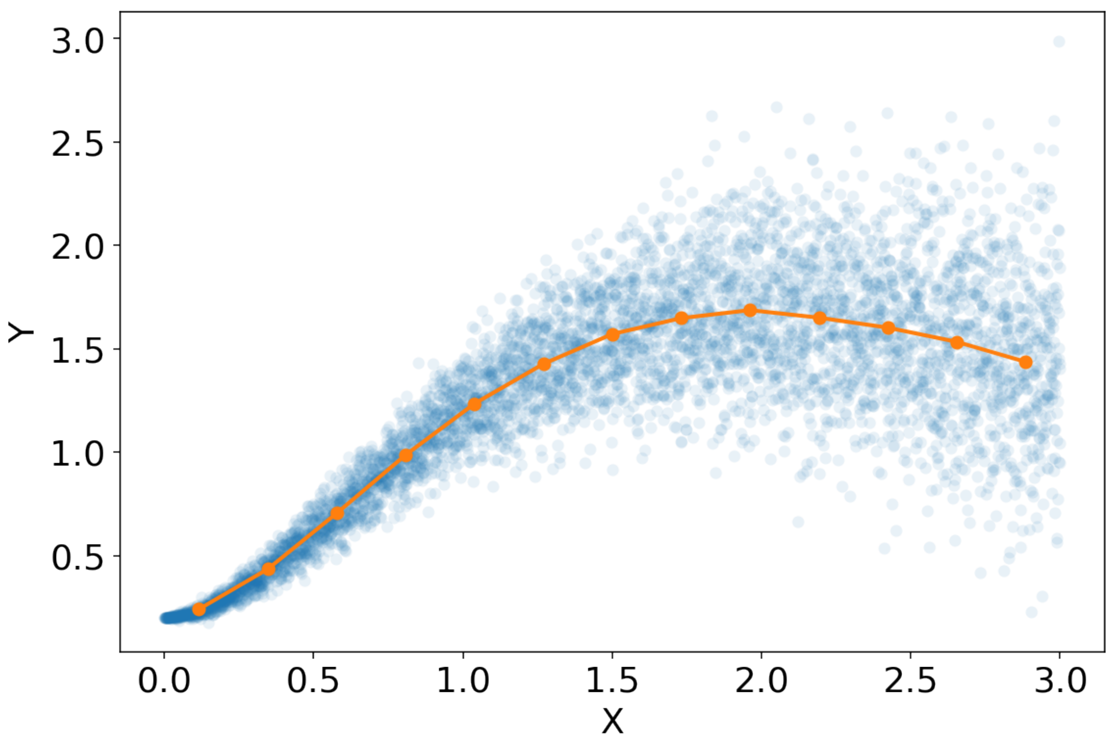


It looks like there are more points in the middle of the spread. We can emphasize this higher density in the middle by adding the **binned trend**:

- Binned trend is also helpful when we want to avoid transparency, for whatever reason.

(Again, we wouldn't duplicate this code in practice—this is just for the lecture.)

[11]:
```
y_bins,bin_edges, misc = binned_stat(x_data,y_data, statistic="mean", bins=13)
x_bins = (bin_edges[:-1]+bin_edges[1:])/2


plt.plot(x_data,y_data, 'o', alpha=0.1, mec='none')
plt.xlabel("X"); plt.ylabel("Y")

plt.plot(x_bins, y_bins, "o-", linewidth=2);
```



Now let's do the same thing but instead of computing the **mean** of each bin, let's compute the **standard deviation**.

One can represent the mean, or average, or **expected value** of a random variable X as:

$$\mathrm{E}(X) \text{ or } \langle X \rangle .$$

For a discrete collection of values x this becomes the familiar sum

$$\langle x \rangle = \frac{1}{N} \sum_{i=1}^{N} x_i,$$

where N is the number of data points.

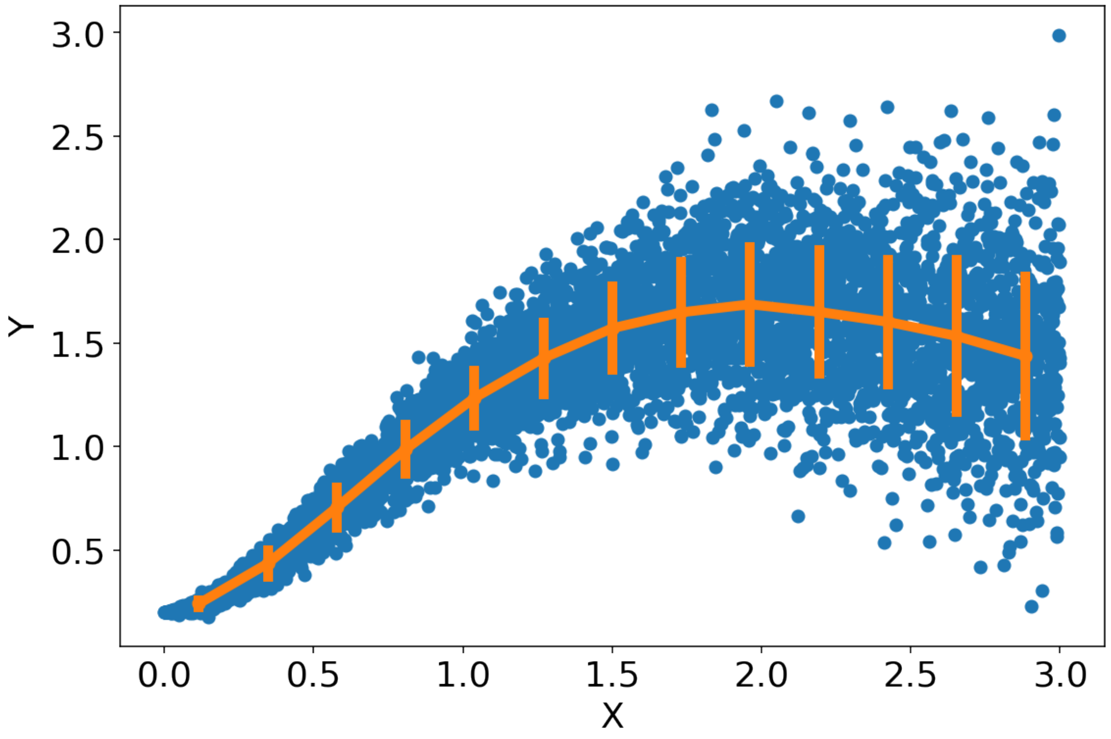
The **standard deviation** σ of that data is related to the mean of the square of the data:

$$\sigma = \sqrt{\left\langle \left(x - \langle x \rangle \right)^2 \right\rangle} = \sqrt{\langle x^2 \rangle - \langle x \rangle^2}$$

To the codez!

```
[12]: s_bins,bin_edges, misc = binned_stat(x_data,y_data,
                                             statistic=np.std, bins=13)
                                                       ^^^^^^
      #
      x_bins = (bin_edges[:-1]+bin_edges[1:])/2

      # plot the raw data
```

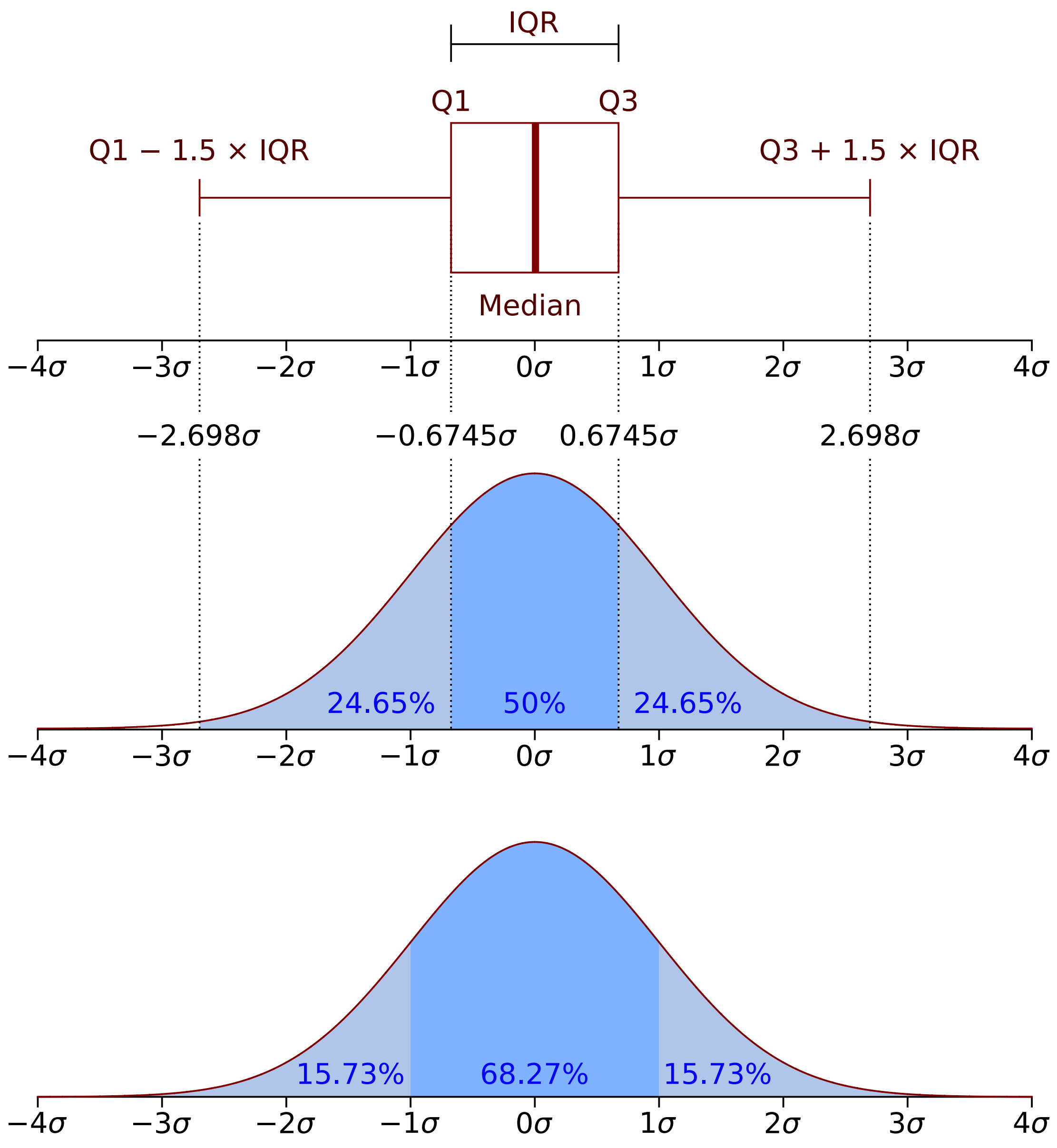
```
plt.plot(x_data,y_data, 'o')

# plot the means of each bin:
plt.plot(x_bins, y_bins, "o-", linewidth=5)

# Now, plot the std as errorbars! (check out the docstring)
plt.errorbar( x_bins, y_bins, yerr=s_bins,
              fmt="none",     # only show errorbars
              elinewidth=5,
              ecolor='C1',
              zorder=5     # plot bars ABOVE scatter
             )

plt.xlabel("X")
plt.ylabel("Y")
plt.show()
```



This idea of using **standard deviation** for errorbars (whiskers) brings box plots and the normal distribution back into play:

[13]:
```
from IPython.display import Image, SVG
SVG("figures/Boxplot_vs_PDF.svg")
```

[13]:



We've combined the **mean** of the data with the **standard deviation** in an **errorbar plot**. This is a great way for seeing both trends and spreads at once, assuming you can interpret it.

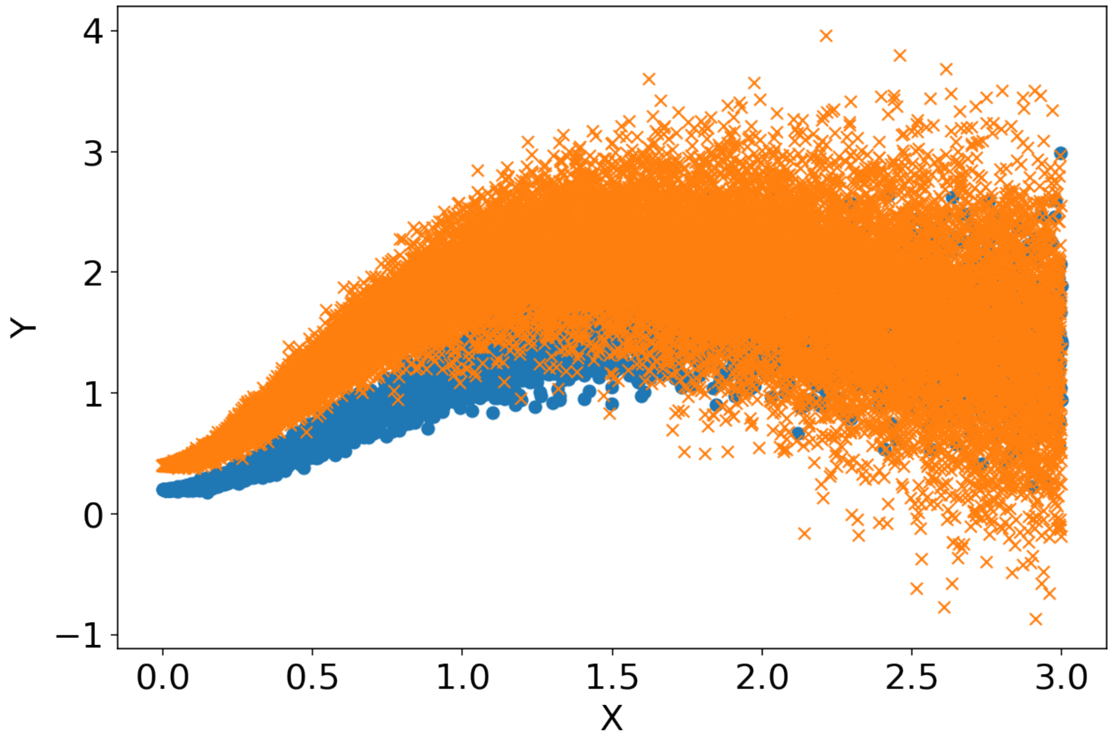
- We can intepret an errorbar of ± 1 std as representing the middle ≈ 68 of the data ***if*** the data within each bin are normally distributed.
- Also, a trend line composed of **box plots** can make a great choice for visualization purposes (see below).

Summarizing multiple scatter plots

Now, in the scatter plot above, all the trend line really shows us is a different view of the same information. The spread of the data as x increases is already obvious from the scatter. But the reduction from a huge **blob** of points let's us avoid mess.

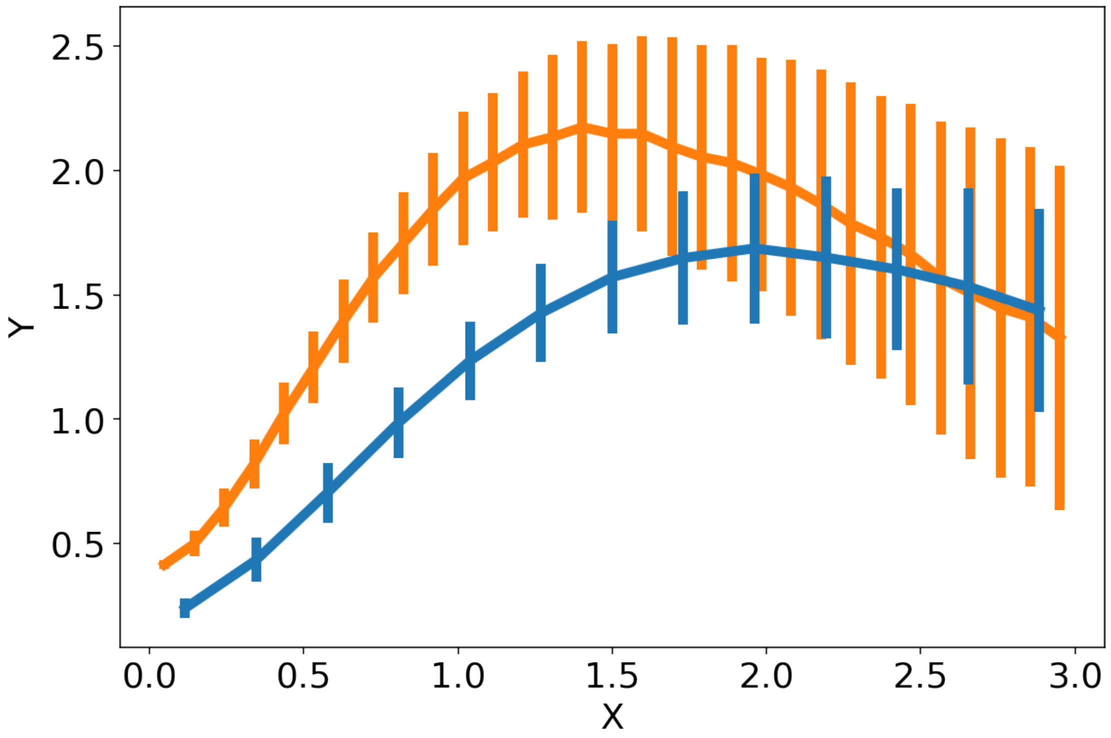
- For example, compare this plot:

[14]:
```python
# second set of points:
x_data2 = np.linspace(0,3,20000)
x2 = x_data2
y_data2 = np.exp(-x2*1.4+1)*(x2*1.5)**2.05 + \
                  0.25*(np.random.randn(20000))*x2 + 0.4

# plot two scatter plots
plt.plot(x_data, y_data , 'o', x_data2,y_data2, 'x')

plt.xlabel("X")
plt.ylabel("Y")
plt.show()
```



with this plot

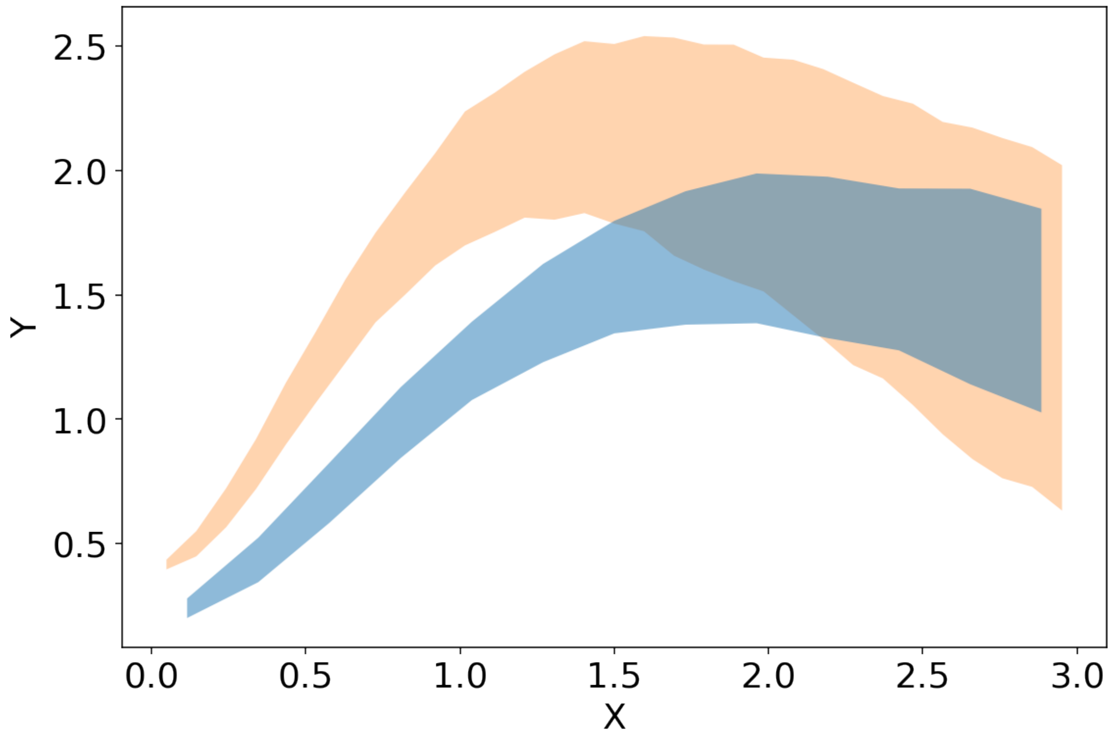
[15]:
```python
def helper_binned_trendline(x_data, y_data, bins=10, ystat='mean', sstat='std'):
    if sstat == "std":
        sstat = np.std

    y_bins,bin_edges, misc = binned_stat(x_data,y_data,
                                         statistic=ystat, bins=bins)
    s_bins,bin_edges, misc = binned_stat(x_data,y_data,
                                         statistic=sstat, bins=bins)
    x_bins = (bin_edges[:-1]+bin_edges[1:])/2
    return x_bins, y_bins, s_bins


# redo the stats for the original blue data:
x_bins, y_bins, s_bins = helper_binned_trendline(x_data, y_data,
                                                 bins=13)

# now the stats for the new orange data:
x_bins2, y_bins2, s_bins2 = helper_binned_trendline(x_data2, y_data2,
                                                    bins=31)


# plot original (blue) curve
plt.errorbar( x_bins, y_bins, yerr=s_bins,
               linewidth=5,
               color='C0', zorder=15 )

# now add the new (orange) curve
plt.errorbar( x_bins2, y_bins2, yerr=s_bins2,
               linewidth=5,
               color='C1', zorder=10 )

plt.xlabel("X")
plt.ylabel("Y")
plt.show()
```



More readable now that we can see the curve in the back

And instead of using vertical bars, we can used **filled curves**:

[16]:
```
# blue data
plt.fill_between(x_bins, y_bins - s_bins, y_bins + s_bins,
                   facecolor='C0', alpha=0.5, linewidth=0
                  ) # 50% transparent blue, no edge

# orange data
plt.fill_between(x_bins2, y_bins2 - s_bins2, y_bins2 + s_bins2,
                   facecolor='C1', alpha=0.33, linewidth=0,
                   zorder=-3
                  )

plt.xlabel("X")
plt.ylabel("Y")
plt.show()
```



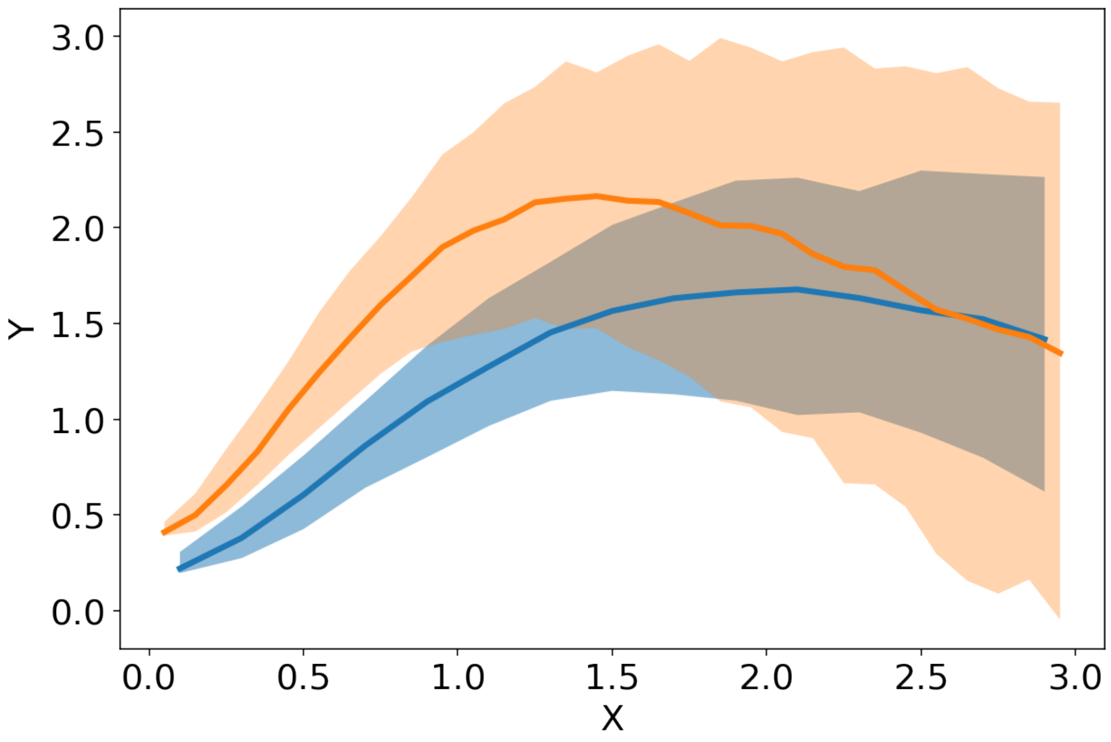
(This is just a visual flourish; a different way to present the same information.)

This plot lets us see the mean ± one standard deviation. But *we are not limited to these statistics*.

- Let's look at the median and 95% confidence intervals (CI):

[17]:
```
def prctile_025(data):
    return np.percentile(data,2.5)

def prctile_975(data):
    return np.percentile(data,97.5)
```

```
def xyBin_md_95ci(xdata,ydata, num_bins):
    """Return xbincenters,ymedian,y025,y975."""
    y_md, be, m = binned_stat(xdata,ydata, statistic="median",    bins=num_bins)
    y_025,be, m = binned_stat(xdata,ydata, statistic=prctile_025, bins=num_bins)
    y_975,be, m = binned_stat(xdata,ydata, statistic=prctile_975, bins=num_bins)
    x_bins = (be[:-1]+be[1:])/2
    return x_bins, y_md, y_025, y_975 # another helper function


# now do the calc:
x1,y1_md,y1_025,y1_975 = xyBin_md_95ci(x_data, y_data,  15)
x2,y2_md,y2_025,y2_975 = xyBin_md_95ci(x_data2,y_data2, 30)


# blue data
plt.fill_between(x1, y1_025, y1_975, facecolor='C0', alpha=0.5, linewidth=0 )
plt.plot(x1,y1_md, '-', linewidth=3)

# red data
plt.fill_between(x2, y2_025, y2_975, facecolor='C1', alpha=0.33, linewidth=0 )
plt.plot(x2,y2_md, '-', linewidth=3)

plt.xlabel("X")
plt.ylabel("Y")
plt.show()
```



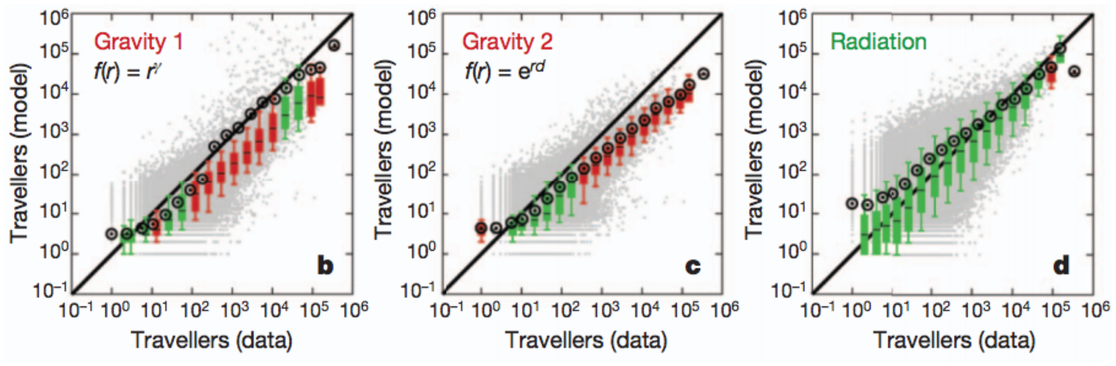
Now the shaded region represents 95% of the (observed) data, regardless of the underlying distribution. Moreover using separate (2.5,97.5%) CI (I'm abusing the term "confidence interval" a bit here) lets us represent **nonsymmetric** or **skewed** data.

Example in practice

Combining scatter plots, box plots, xy-binning and summary statistics with **color** leads to a concise but information-rich graphic:

```
[18]: Image("figures/simini_nature_scatterplot_examples.png", width=800)
```

[18]:



(Simini *et al*, Nature 2012)

Here the predictions for different modeling approaches are compared using a scatter plot. The more accurate the predictions, the closer to the line of equality ($y = x$), drawn in black.

The raw scatter (light gray) shows a lot of spread, so a trend line of box plots was used to show where the "bulk" of the points lie. Black circles represent the mean trend.

Boxes that go through the line of equality are colored **green**, otherwise colored **red**, to show how close the central tendency of the data are to the line of equality.

- Red = bad, green = good makes this choice of colors nice. But red-green colorblindness is the most common form of color blindness, so you do not want to rely upon these two colors that much.

Takeaways

- **Scatter plots** - a primary tool for comparing pairs of variables!
- Important point not discussed: **logarithmic scales**. X, Y, and XY log scales are all common, and useful for understanding the relationship between pairs of variables when one or both are broadly distributed. But all the concerns with data distortion that arose with histograms also arise here with scatterplots. Keep this in mind!

XY-data
If you have XY-data, think about visualizing a trend of relationship between X and Y using a plain old boring scatter plot. ← much more to come along these lines!

To understand the relationship between two variables, you've accomplished a **great deal** by just understanding how the **average** and **spread** of one variable depends on the other.