

Variants of Linear Regression

Statistics and Data Science

Spring 2025

Goals for today: you should be able to...

- ❖ **Lecture 22/23 notebook:**

- ❖ Apply simple least-squares linear regression in Python
- ❖ Perform Polynomial regression
- ❖ Use the chi-squared statistic to assess fit quality
- ❖ Apply information criteria to choose between different models
- ❖ Use bootstrap methods to get errors in parameters from regression

Proposed meeting times

- ❖ Mykola: Monday 4/7 3 PM
- ❖ Finian + Yoki: Monday 4/7 4:30 PM (4 if colloquium is not an issue)
- ❖ Alia + Amanda + Marcos: Tuesday 4/8 4 PM
- ❖ Mira: Tuesday 4/8 4:30 PM
- ❖ Cullen: Wednesday 4/9 4 PM
- ❖ Amelia + Mohamed: Wednesday 4/9 4:30 PM
- ❖ Yunchong: Thursday 4/10 4 PM
- ❖ Jake: Friday 4/11 1 PM
- ❖ Lauren + Nathalie: Friday 4/12 2 PM
- ❖ Julissa: Friday 4/12 2:30

Review

- ❖ Suppose we have a set of measurements y_i that should be expressible as a function of some other set of variables, x_i , with parameters α, β, γ etc. (so the i th y will depend on the i th vector of variables x_i) with random scatter:

$$y_i = f(x_i; \alpha, \beta, \gamma \dots) + \varepsilon_i$$

where ε_i is a random variable of mean 0, independent of all the parameters

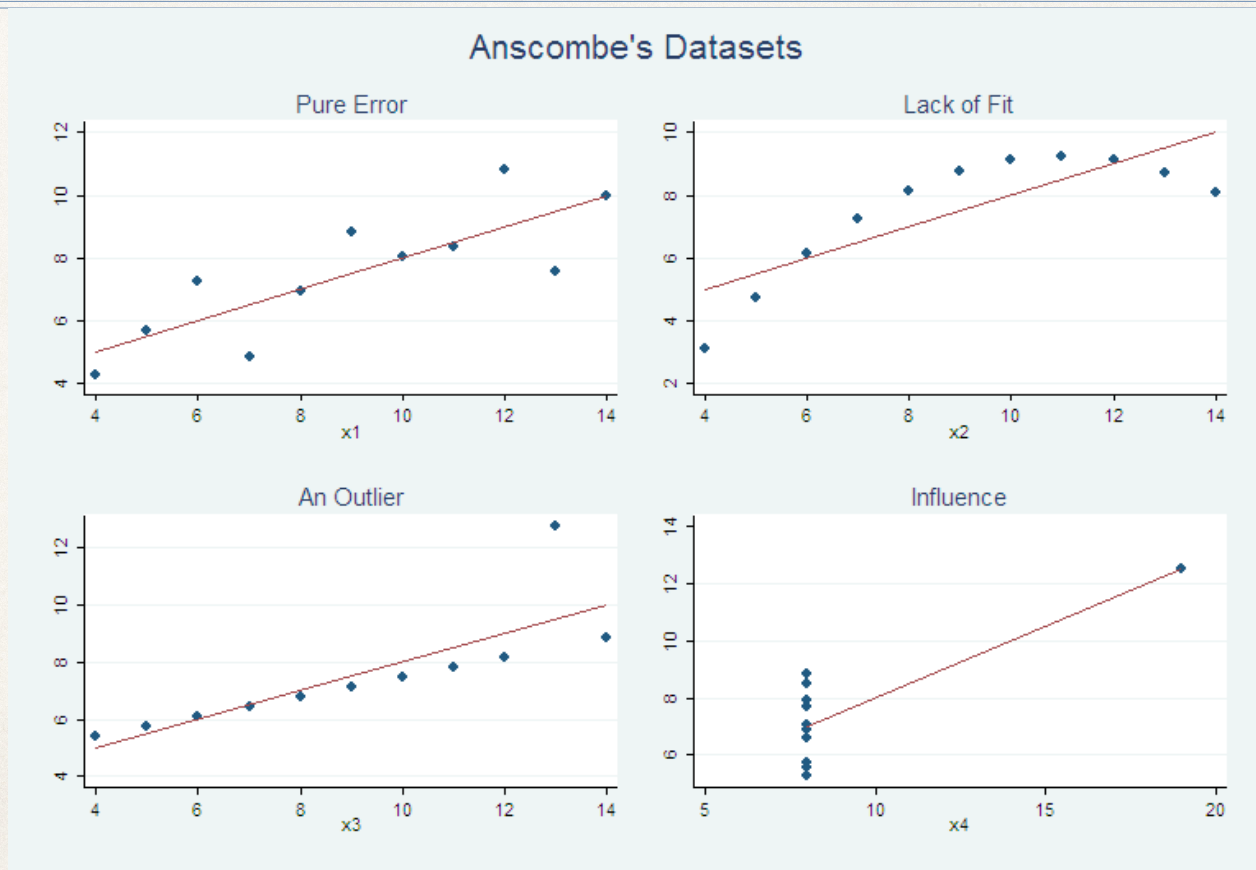
- ❖ If ε_i is Gaussian, $L(\text{data} \mid \text{params}) = \prod (2\pi \sigma_i^2)^{-1/2} \exp(-(y_i - f(x_i; \alpha, \beta, \gamma \dots))^2 / 2\sigma_i^2)$
- ❖ So maximizing likelihood is minimizing $\chi^2 = \sum (y_i - f(x_i; \alpha, \beta, \gamma \dots))^2 / \sigma_i^2$
- ❖ Determining the parameters from observed x_i and y_i is the problem of *regression*

Questions to answer from last time

- ❖ How well does the standard deviation of the slopes correspond to the mean value of serr in this perfectly specified case?
- ❖ How much do the serr values vary from dataset to dataset?
- ❖ What about the slope errors does the serr value not capture? (Take a look at the plot!)

What can go wrong in regression? "Anscombe's Quartet"

- ❖ These all have the same least-squares fit, same variances in x and y , same Pearson correlation coefficient ρ ...



Regression in Python

- ❖ Python offers a variety of least-squares regression routines, which mostly work similarly:
 - `scipy.stats.linregress`: simplest version, fits a linear relation between 1 dependent variable and 1 independent variable with least-squares
 - `scipy.linalg.lstsq`: performs linear regression between 1 dependent variable and multiple independent variables (note you could just use different functions of the same variable as independent variables -- e.g. fit y vs. $[x, x^2, [x^3...]]$ to fit polynomials, or y vs. $[\sin x, \cos x]$ for sinusoids) .
- ❖ It does **NOT** fit for a constant term; you need to provide an extra independent variable that's an array of all 1's to cover that.

Regression in Python

`numpy.polyfit` : performs least-squares regression for polynomial fits (of specified degree).

`scipy.optimize.curve_fit` : performs nonlinear least-squares regression for arbitrary functions.

`scipy.optimize` also provides a variety of minimization routines, which you can use for fitting by writing a function that calculates the loss you want to use (e.g. chi-squared).

Specialized packages: statsmodels, patsy, and scikit-learn

- ❖ The scipy / numpy fitting routines handle a limited set of cases; e.g., robust fitting isn't covered. Some key tools for more sophisticated modeling are:

`patsy` : A tool for describing models to be fit to data

`statsmodels` : A variety of tools for fitting, using `pandas` and `patsy`

`scikit-learn`: Provides a large variety of tools commonly used in machine-learning, including fitting methods

We'll explore these tools starting next week

Fitting for the Hubble Constant

- ❖ Hubble's Law: $v = H_0 D$, where v is the measured recession velocity and D is distance (only true nearby). H_0 is in units of km/s/Mpc.
- ❖ Taking the log (base 10) of both sides:

$$\log v = \log D + \log H_0$$

- ❖ We commonly provide distances in the form of "distance modulus":

$$\mu = 5 \log (D / 10 \text{ pc})$$

so we expect:

$$\log v = 0.2 \mu + \log (10^{-5} \text{ Mpc}) + \log H_0, \text{ so}$$

$$\mu = 5 \log v - 5 \log (10^{-5} \text{ Mpc}) - 5 \log H_0 = 5 \log v + k$$

$$\text{and } \log H_0 = -0.2 k - \log(10^{-5})$$

Trying this with data

$$\mu = 5 \log v + k \text{ and } \log H_0 = -0.2 k - \log(10^{-5} \text{ Mpc})$$

- ❖ We'll explore fitting using data from the Constitution supernova sample (from Hicken et al. 2009). Download the data from Canvas if you haven't already...

```
path = './' # CHANGE THIS IF NEEDED!
data=pd.read_fwf(path+'sndata.txt')
cz=data['cz']
mu=data['mu']
sigma_mu=data['sigma_mu']
d = 10**((mu/5)*1E-5)
plt.errorbar(cz,mu,yerr=sigma_mu,fmt='b.',alpha=0.2)
logv=np.log10(cz)
```

- ❖ Now plot mu as a function of logv, with error bars.

Fitting for the Hubble Constant

- ❖ 1) Let's fit for μ vs. (as a function of) $\log v$. **Why this direction instead of fitting for $\log v$ as a function of μ ?**

```
# scipy.stats.linregress: ordinary least-squares regression
```

```
slope, intercept, r, p, s = stats.linregress(logv, mu)
```

```
plt.plot(logv, mu, 'b.')
```

```
# overplot the fit
```

```
plt.plot(logv, logv*slope+intercept, 'r-')
```

```
print(f'slope: {slope:.3f} +/- {s:.3f}')
```

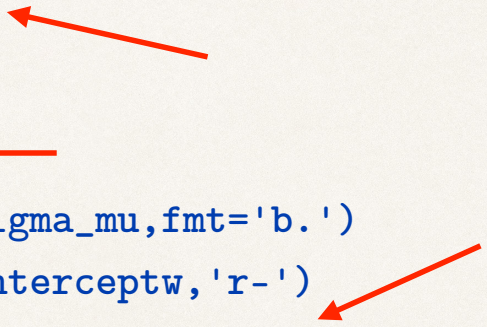
```
print(f'intercept: {intercept:.3f}')
```

```
print(f'H0: {(10**(-0.2*intercept-np.log10(1E-5))):.2f} km/sec/Mpc')
```


Weighting with errors

2) Let's try weighted least squares. For the Gaussian assumptions behind least-squares, the optimal weights are $1/\sigma^2$, but in Python to get that the input weights need to be $1/\sigma$ (I have NO idea why). The simplest routine we can use is `numpy.polyfit` with degree 1. Do things get better?

```
weight = 1./sigma_mu
coeffsw, covarw = np.polyfit(logv,mu,1,w=weight,cov=True)
slopew = coeffsw[0]
interceptw = coeffsw[1]
s=np.sqrt(covarw[0,0])
plt.errorbar(logv,mu,yerr=sigma_mu,fmt='b. ')
plt.plot(logv,logv*slopew+interceptw,'r-')
h0errw = (10**(-0.2*(interceptw-np.sqrt(covarw[1,1]))-np.log10(1E-5))-
          10**(-0.2*(interceptw+np.sqrt(covarw[1,1]))-np.log10(1E-5)))/2.
print(f'slope: {slopew:.3f} +/- {sw:.3f}')
print(f'intercept: {interceptw:.3f}')
print(f'H0: {(10**(-0.2*interceptw-np.log10(1E-5))):.2f} km/sec/Mpc +/- {h0errw:.2f}')
```



Examining the results

- ❖ Recall $\mu = 5 \log v + k$ and $\log H_0 = -0.2 k - \log(10^{-5} \text{ Mpc})$
- ❖ Are the results consistent with our expectations for each method? Consider both the slope and H_0 value (which should be 65 km/sec/Mpc by construction for this dataset).
- ❖ Let's examine the scatter about the line, too. Examine which method gives smallest scatter about the fit line, and explain why.

```
print(f'observed scatter without and with weighting: {np.std(mu - mu_fit):.7f},  
{np.std(mu-mu_fit_err):.7f}')
```

```
print(f'expected scatter from RMS(sigma_mu): {np.sqrt(np.sum(sigma_mu**2)/  
len(sigma_mu)):.4f}')
```


```
print(f'expected scatter from mean(sigma_mu): {np.mean(sigma_mu):.4f}')
```

```
print(f'expected scatter from median(sigma_mu): {np.median(sigma_mu):.4f}')
```


A key tool: checking fit residuals

- ❖ An important question is whether the residuals look symmetric about the fit. This will be easiest to check if we average the data in bins.

```
plt.plot(logv, mu-mu_fit,'b.')  
logv_bin,edges,bin_numbers = stats.binned_statistic(logv,logv,statistic='mean', bins=15)  
  
residual_bin,edges,bin_numbers = stats.binned_statistic(logv, mu-mu_fit,statistic='mean',bins=15)  
  
plt.plot(logv_bin,residual_bin,'r-')  
plt.axhline(0,color='g',ls='--')
```



Quadratic fits

- ❖ We can fit a quadratic function to μ vs. $\log v$ by changing the order of the polynomial:

```
coeffs_quad, covar_quad = np.polyfit(logv,mu,2,w=weight,cov=True)
```

- ❖ Two easy options for evaluating polynomials:

1) `numpy.polyval`: takes as input coefficients of the polynomial **and** the x values to be evaluated

```
quad_fit = np.polyval(coeffs_quad,logv)
```

2) `numpy.poly1d`: takes as input coefficients of the polynomial; creates a **function** that can evaluate the polynomial

```
poly_quad = np.poly1d(coeffs_quad)
```

```
quad_fit2 = poly_quad(logv)
```


Quadratic fit

- ❖ Overplot both the quadratic fit (`quad_fit`) and the linear fit (`mu_fit`) on the data (i.e., enable the plot!).
 - ❖ How do they compare?
 - ❖ How does the linear coefficient (slope) of the quadratic fit compare to what you expect?
- ❖ To evaluate whether the quadratic fit is a better fit to the data:
 - ❖ Compute chi-squared from the linear fit (using `mu`, `mu_fit_err` & `sigma_mu`) and from the quadratic fit (using `mu`, `quad_fit`, & `sigma_mu`), and see which has smaller chi-squared.

$$\chi^2 = \sum (y_i - f(x_i; \alpha, \beta, \gamma \dots))^2 / \sigma_i^2$$

When should we stop fitting?

- ❖ We could fit a quartic to the data:

```
coeffs_quart, covar_quart = np.polyfit(logv,mu,4,w=weight,cov=True)
poly_quart = np.poly1d(coeffs_quart)
quart_fit = poly_quart(logv)
```

- ❖ Compare the quartic fit with the linear and quadratic fits (enable the plot).
- ❖ Also compare the chi-squared value for this fit to those obtained for the linear and quadratic cases:

```
print(f'chi squared - quartic: {np.sum((mu-quart_fit)**2/sigma_mu**2):.3f}')
```


When do we stop fitting?

- ❖ Compare the trends in the residuals.
- ❖ This code will make a plot; also **write code to do a quantitative comparison of how much the binned residuals scatter about zero for the quartic vs. linear case**
- ❖ **Note:** you should look at the mean of $(\text{residual})^2$ or its square root, **not** the standard deviation, since the latter subtracts off the mean residual...

```
residual_bin_quart,edges,bin_numbers = stats.binned_statistic(logv, mu-quart_fit,  
statistic='mean', bins=15)  
  
plt.plot(logv_bin,residual_bin,'r-',label='linear fit')  
plt.plot(logv_bin,residual_bin_quart,'b--',label='quartic fit')  
plt.axhline(0,color='g',ls='--')
```


When do we stop fitting?

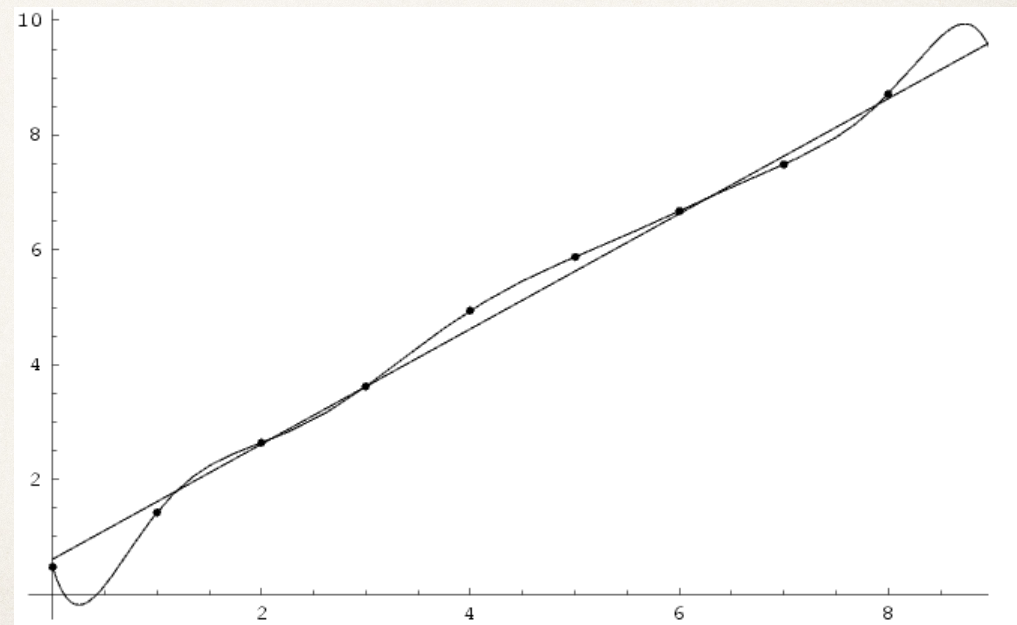
- ❖ Let's test how the fits extrapolate:

```
x=np.linspace(0,10,1000)
plt.plot(logv,mu,'b.',label='data')
plt.xlim((1,7))
plt.ylim( (30,50))
plt.xlabel('log v')
plt.ylabel(r'$\mu$')
plt.plot(x,poly_quart(x),'r-',label='quartic')
plt.plot(x,poly_quad(x),'b--',label='quadratic')
plt.plot(x,poly_lin(x),'c--',label='linear')

plt.legend()
```


When do we stop fitting?

- ❖ Consider Occam's razor: "entities [=parameters] should not be multiplied beyond necessity". Simpler models of the Universe are more likely to be correct.
- ❖ We have a better fit with a quartic than a linear function, but also a more complicated model.
- ❖ We should **always** get smaller (or at least no larger) residuals when we add parameters that extend a model. Is the improvement when we add extra parameters significant, or are we overfitting?



How do we test fits?

- ❖ For fits of probability distribution functions, we might use Pearson's chi-squared test or the K-S test to evaluate the null hypothesis that our data fits the model we have produced.
- ❖ If we have Gaussian errors of known amplitude, we can use the observed chi-squared ($\chi^2 = \sum (y_i - f(x_i; \alpha, \beta, \gamma \dots))^2 / \sigma_i^2$) value to test whether our model is a good fit, by comparing to the CDF for chi-squared with (# data - # parameters) degrees of freedom.
- ❖ If we have multiple models that are both reasonable fits, we need to think more...

When do we stop fitting?

```
print(f'chi-squared for linear: {chisq_lin:.3f}')  
print(f'chi-squared for quartic: {chisq_q:.3f}')  
print(f'difference: {chisq_lin-chisq_q:.3f}')
```

- ❖ We have 164 data points, so there are 162 DOF for the linear model, 159 for quartic. We can compute the significance=0.05 rejection region or the p -value, as before:

```
print(f'alpha=0.05 chi-squared cutoff for linear: {stats.chi2.ppf(.95,162):.3f}')
```

```
print(f'p-value for linear: {1-stats.chi2.cdf(chisq_lin,162):.3g}')
```
- ❖ **Assess: are our fits acceptable?** (i.e., is the chi-squared value obtained one that would have a reasonable probability of occurring by chance if this was the correct fit? or: do we reject the null hypothesis that our fit is consistent with the data?)
- ❖ When going from linear to quartic, we added 3 parameters, and reduced chi-squared by 0.39 . Is that significant?

Information criteria

- ❖ Different arguments can lead to different criteria for whether an improvement in the fit is 'worth' the extra parameters - i.e. larger than you'd expect by chance just from increasing the freedom of the model to fit the data. Common ones are (see [Liddle 2007, astro-ph/0701113](#)):
- ❖ Bayesian (or Schwartz) Information Criterion, BIC: $BIC = -2 \ln L_{max} + k \ln n$, where we have n datapoints, k parameters, and L_{max} is the maximum probability in the model. The smaller the BIC, the better the model.
- ❖ Akaike Information Criterion, AIC: derived from maximum-entropy type arguments, $AIC = -2 \ln L_{max} + 2k + 2k(k+1)/(n-k-1)$
- ❖ Deviance Information Criterion, DIC: uses averages over the full likelihood distribution (obtained with MCMC techniques) to count only parameters effectively constrained by the data; most similar to AIC

Information criteria

- ❖ Since $-2 \ln L = \chi^2$ when we are dealing with Gaussian errors, for our case these reduce to:
 - ❖ $\text{BIC} = \chi^2 + k \ln n = 140$ for quartic model, 130 for linear.
 - ❖ $\text{AIC} = \chi^2 + 2k + 2k(k+1)/(n-k-1) = 128$ for quartic model, 124 for linear.
 - ❖ So both criteria agree that the linear fit is preferred!

Bayesian model selection

- ❖ A more general Bayesian method of handling a choice between models (M_1, M_2) given data x is using the *Bayes factor*, the ratio of the likelihood of each model:

$$K = \frac{p(x|M_1)}{p(x|M_2)}.$$

- ❖ K is then the odds in favor of model M_1 over M_2 , absent priors. Note we are marginalizing over the parameters of each model,

$$K = \frac{p(x|M_1)}{p(x|M_2)} = \frac{\int p(\theta_1|M_1)p(x|\theta_1, M_1)d\theta_1}{\int p(\theta_2|M_2)p(x|\theta_2, M_2)d\theta_2}.$$

- ❖ Compare this to the likelihood ratio test, where we compared the **maximum** probability of observing the data under each hypothesis, $\Lambda = \max(L(x|\theta_1, M_1) / \max(L(x|\theta_2, M_2))$
- ❖ One way to think of the Bayes factor K : if the prior odds on model 1 over model 2 is P , the posterior odds given our data will be KP .

Where do regression error estimates come from?

- ❖ We saw before that, for large n (or Gaussian errors), the likelihood for the parameters will be a multivariate Gaussian, with covariance matrix equal to the inverse of the Fisher information matrix, $I_{ij} = \mathbf{E}(\partial \ln L / \partial \vartheta_i \partial \ln L / \partial \vartheta_j) = -\mathbf{E}(\partial^2 \ln L / \partial \vartheta_i \partial \vartheta_j)$
- ❖ We could **approximate** the expectation value just by taking the value at the peak of the likelihood, $-(\partial^2 \ln L / \partial \vartheta_i \partial \vartheta_j)$. This is generally the method regression routines use.
- ❖ Those derivatives will only be correct if your error estimates are correct (and only appropriate at all if your errors are Gaussian or n is large).
- ❖ A trick some routines use is to adjust (or estimate) the errors in the data by **assuming** that the chi-squared value for the fit $(=\sum (y_i - f(x_i; \alpha, \beta, \gamma \dots))^2 / \sigma_i^2)$ **must equal** the # of degrees of freedom (= # of independent data points - # of parameters).
 - ❖ A χ^2 random variable has mean k and variance $2k$ for k DOF, though -- we shouldn't expect $\chi^2 = k$ exactly...

Bootstrap error estimates for regression

- ❖ There are two different ways we could generate bootstrap samples:
 1. Suppose the noise is *homoskedastic*; then $y_i - f(x_i; \alpha, \beta, \gamma \dots) = \varepsilon_i$, where all the ε_i are drawn from the same function.
 - ❖ So after we have a fit, we could use the original set of x_i , and generate a new set of bootstrap y_i^* for each one, where $y_i^* = f(x_i; \alpha, \beta, \gamma \dots) + \varepsilon_j$, where ε_j is the **residual** ($= y_i - f(x_i; \alpha, \beta, \gamma \dots)$) from the (randomly selected) j th data point.
 2. We could randomly select sets of (x_i, y_i) , just like we selected sets of x_i before. A given (x_i, y_i) **pair** could appear 0,1,2, etc. times in each bootstrap sample, just like a given x_i did before.
 - ❖ This does not depend on the errors being homoskedastic, but some regression routines may not be able to handle repeated data well.

Fitting for the Hubble Constant

- ❖ Applying bootstrap errors :

```
nsims=int(1E3)
n=len(mu)
bootidx=np.floor(random.rand(nsims,n)*n)
bootidx=bootidx.astype(int) ←
logv_boot=logv[bootidx]
mu_boot=mu[bootidx]
w_boot=weight[bootidx]
slope=np.zeros(nsims)
intercept=np.copy(slope)
for i in np.arange(nsims):
    coeffs = np.polyfit(logv_boot[i,:],mu_boot[i,:],1,w=w_boot[i,:])
    intercept[i]=coeffs[1]
    slope[i]=coeffs[0]
```

- ❖ We then plot the distributions of slope and intercept values across the bootstraps. **Overplot a point corresponding to the best linear fit for comparison.**

What if there are errors in the x values?

- ❖ Before, we assumed that independent variables were known perfectly, but there are random errors in the y 's:

$$y_i = f(x_i; \alpha, \beta, \gamma \dots) + \varepsilon_i$$

- ❖ Then:

$$L(\text{data} \mid \text{params}) = \prod (2\pi \sigma_i^2)^{-1/2} \exp \left(- (y_i - f(x_i; \alpha, \beta, \gamma \dots))^2 / 2\sigma_i^2 \right)$$

- ❖ Suppose instead there are errors in the x 's as well: $x_{i,obs} = x_{i,true} + \delta_i$; and y is a function of $x_{i,true}$, not $x_{i,obs}$. Then, if $\delta_i = \psi_i N(0,1)$:

$$L(x_{i,obs}, y_i \mid \text{params}) = \prod \int (2\pi \sigma_i^2)^{-1/2} (2\pi \psi_i^2)^{-1/2} \times \\ \exp \left(- (y_i - f(x_{i,true}, \alpha, \beta, \gamma \dots))^2 / 2\sigma_i^2 \right) \exp \left(- (x_{i,obs} - x_{i,true})^2 / 2\psi_i^2 \right) dx_{i,true}$$

What if there are errors in the x values?

- ❖ We have a few options to handle this case:
 1. Do a full likelihood analysis, marginalizing over $x_{i,true}$
 2. Do a fit that ignores any error estimates, and treats errors / deviations in X and Y equivalently
 - ❖ e.g., minimize sum-of-squares in the direction orthogonal to the fit line, rather than the sum-of-squares deviation in y .
 - ❖ Problem: this only works optimally if errors in x and y are comparable

What if there are errors in the x values?

3. Apply an algorithm that takes errors in both variables into account !
 - ❖ See the Python package `scipy.odr`, which does non-linear fitting by minimizing orthogonal distance (optionally, in a chi-squared sense, incorporating x and y errors rather than treating both axes the same)
 - ❖ Note: for lots more detail on regression methods, including these cases, see <https://arxiv.org/abs/1008.4686> by Hogg, Bovy and Lang!

NOTE: If we are interested in modeling the relationship between an *observed* y and *observed* x , ordinary least squares ('OLS') of y vs. x is still favored; these techniques are for trying to determine the 'true' relationship between x and y .

Fitting methods from Isobe et al.

Isobe et al. 2009,
Ap. J. Vol. 364,
p. 104

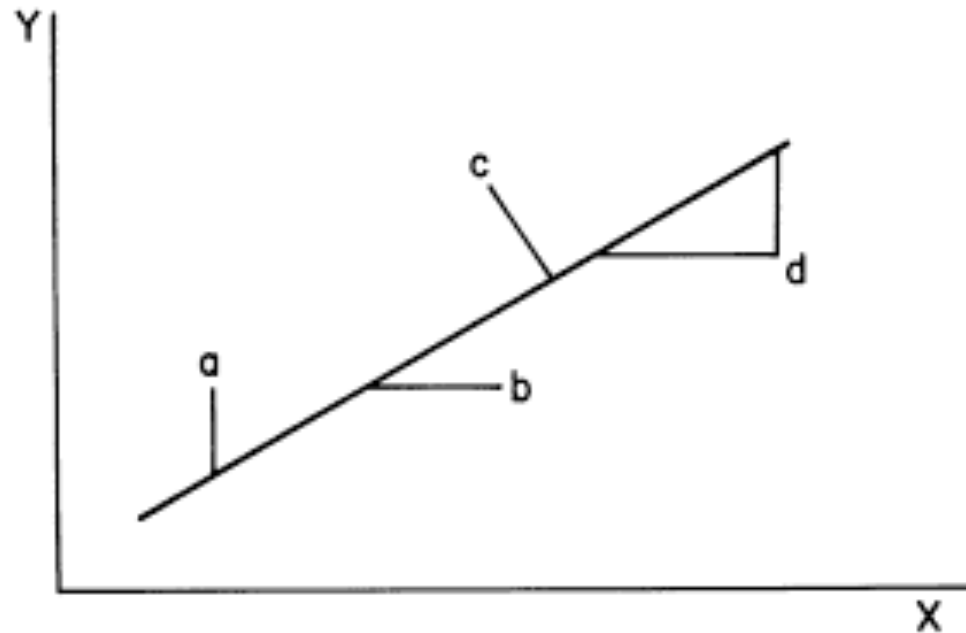


FIG. 1.—Illustration of the different methods for minimizing the distance of the data from a fitted line: (a) OLS($Y|X$), where the distance is measured vertically; (b) OLS($X|Y$), where the distance is taken horizontally; (c) OR, where the distance is measured vertically to the line; and (d) RMA, where the distances are measured both perpendicularly and horizontally. No illustration of the OLS bisector is drawn in this figure.

Alternative to EIV regression

- ❖ Isobe et al. generally recommended using the bisector between the line you get by doing least-squares with y as the dependent variable and with x as the dependent variable.
- ❖ This tends to be more stable than most of the alternatives they considered.
- ❖ However, it is effectively assuming errors are comparable in both variables...

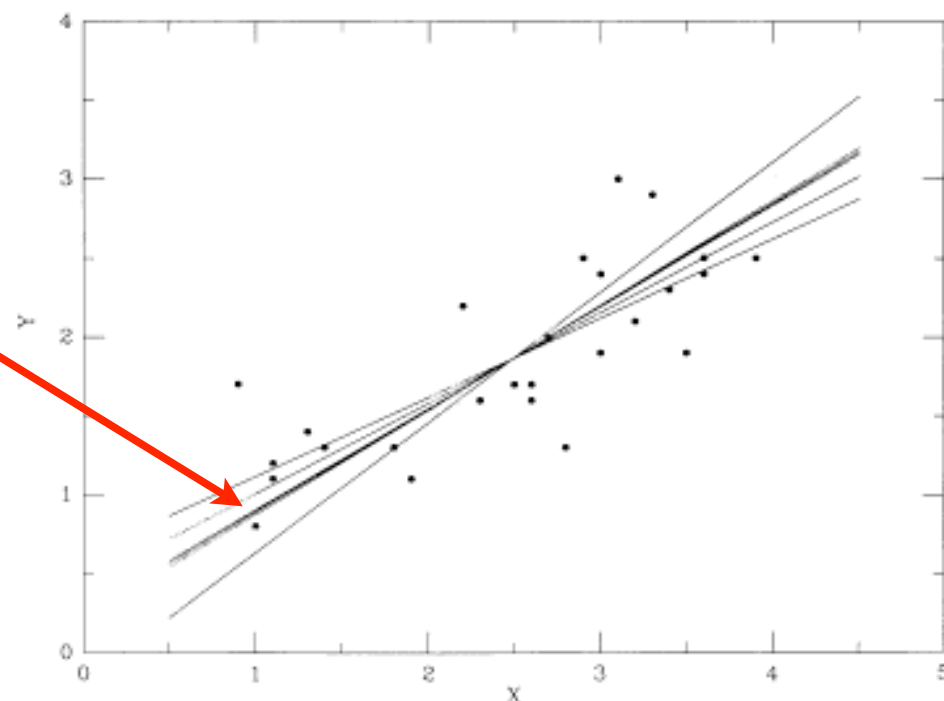


FIG. 1.—[Figs. 1–6 are diagrams of the bivariate regressions discussed in the text, using a hypothetical data set with 26 points.] Unweighted least-squares models (§ 2). Starting with the steepest line, they are OLS($X|Y$), OLS mean, OLS bisector, reduced major axis, orthogonal, and OLS($Y|X$) regressions. The lines are calculated using the program SLOPES.