PHYS134
Homework/Project 1
2022-02-01

Python (`numpy`, `scipy`, `matplotlib`), Plotting, Probability, and Curve Fitting

This homework/project is all about exploring probability, statistics, and curve fitting in python.

Install `python3` and `jupyter`. Anaconda is a good place to start if you don't want to install it piece by piece: `https://www.anaconda.com/products/individual`.

For some of you, most of the time will be spent learning `numpy` concepts like how to manipulate arrays and `matplotlib` concepts like creating plotting many things on top of each other. To "think numpy," you'll need to wrap your mind around doing everything with arrays. You'll avoid loops, if statements, and recursion because they are very slow and inefficient for numerical/scientific computing. If you want to add a million pairs of numbers together, you create arrays of a million numbers each, and then you add the arrays together in one command `c=a+b`.

Start the top of your notebook with either `%pylab inline` or `%pylab notebook`. The `%pylab` command imports a huge number of functions and makes them immediately available. This isn't great if you're writing a long program where you should only explicitly import what you need, it but makes quick exploration extremely easy. The `inline` version does not allow you to zoom into plots once you've made them. The `notebook` version does allow you to zoom, but requires you to either explicitly write `figure()` before each plotting command or close the figure—otherwise it will draw the new plot on top of the old plot. Try both, but most people will probably settle on `inline` for this assignment.

Give all of your plots titles! The way we'll grade this assignment is just to scroll through looking at the plots and their titles and seeing if you did reasonable things. We probably won't read your code or comments in detail.

I expect you to hand in a PDF, probably by printing your notebook from your browser. Keep your jupyter notebook `.ipynb` notebook file.

1. Numpy and Plotting Basics; Drawing from Distributions; PDF and CDF (6 pts)

- Show me that you can create 1D numpy arrays with various useful starting data. Commands include `array`, `zeros`, `ones`, `arange`, and `linspace`. Look at the help for these (either through a web search or by evaluating something like `zeros?`) and explore their options.

- When add, multiply, exp, sqrt, sin, cos, or arctan arrays, it does it element-by-element. To do proper matrix math, you need to do something else, which we'll explore later.

- Plot some of your favorite functions on the same plot. Look at the help for `plot` and play with some of the options. Use the `label` option of the `plot` function and then call `legend()` after you've made the plots. Be sure to call `title` with a meaningful title for all plots! Use `xlim` and `ylim` to provide limits that zoom the plot in or out. Normally I'd be a stickler about calling `xlabel` and `ylabel` with meaningful labels and units, but not for this assignment. All of these are functions of the `matplotlib` library, so if you search the web, it's good to include `matplotlib` in your search. (Matlab and Octave have similar functions that do very similar things.) The basic philosophy of matplotlib is that the parameters of `plot` are like color and style that apply to one individual line or set of dots that could get a label, whereas things that apply to the plot as a whole are called as functions afterward.

- Use `rand` and `randn` to create random data. Use `hist` to plot histograms. You probably want at least a million samples. You'll want to increase the default number of `bins` and possibly change the `range` parameter when you call `hist`. The `hist` command returns the counts, the bin edges, and the geometry. If you don't want return values like this displayed, end the command with a semicolon.

- Explore other distributions by plotting their histograms. Learn a bit about them by getting help on them and through tomes like Wikipedia. Ones relevant for optics and data analysis include `normal`, `exponential`, `poisson`, and `chisquare`.

- Plot the probability density function (PDF)[a] and cumulative distribution function (CDF)[b] of a few distributions. These need to be imported: `from scipy.stats import norm, expon, chi2`. After you've created an array of `x` values, you can say things like `y = norm.pdf(x)`. The chi2 distribution needs a "degrees of freedom" parameter, which is an integer. Try 1 through 9 see if you can match the Wikipedia plots for the Chi-square distribution[c].

- The CDF of some distributions will be used later. Given a number, you can look up the probability that randomly drawing a number from that distribution would give you something *less*. (For example, `norm.cdf(0)==0.5` because half of the time a standard normal distribution gives you a number less than 0.) If you draw a million numbers from a normal distribution, a histogram of those numbers will be Gaussian. If you then look up the CDF of *each* of those random numbers, you'll get a probability between 0 and 1, and the histogram of those probabilities will be *flat*. Show an example of this. Think about why this should be, based on the definition the CDF. This is one way to tell if a collection of numbers is drawn from a particular distribution.

[a] `https://en.wikipedia.org/wiki/Probability_density_function`
[b] `https://en.wikipedia.org/wiki/Cumulative_distribution_function`
[c] `https://en.wikipedia.org/wiki/Chi-square_distribution`

2. Central Limit Theorem (4 pts)
Convince yourself and me that the central limit theorem works (and that you understand it enough to turn it into code):

- Histogram a million random numbers uniformly distributed from 0 to 1. Use `rand`.
- Verify that the mean of a million such uniformly-distributed numbers is close to 1/2 and the standard deviation is close to $1/\sqrt{12}$. (You could calculate this analytically by using the definition and integrating x^2. This $\sqrt{12}$ comes up in advanced discussions treating rounding error or analog-to-digital conversion quantization error as random noise.)
- Histogram the average of 2 arrays of such random numbers.
- Histogram the average of 5 arrays of such random numbers. Try something like `mean(rand(1000000,5),axis=1)`. Look up or ask about each piece of this if you don't know what it means or how it works.
- Note: We'll use arrays that are 1D, 2D, 3D, etc. If you have an array called `a`, you can say `a.shape` to get the size of the array along each of its dimensions. Look at `rand(3,2)` and then get its shape. A big part of "thinking numpy" is keeping in mind how many dimensions of what size each variable is, along with what the rows and columns mean. Nobody ever got fired for examining a matrix mid-calculation.
- What is the mean and standard deviation of these averaged distributions? Are they what the central limit theorem predicts? (The mean should converge on the mean. The variance should narrow by $1/N$, meaning that the standard deviation should narrow by $1/\sqrt{N}$.)
- Try it for the `exponential` and `poisson` distributions, along with any other that catch your fancy. Since `poisson` gives integers, you'll need to average many more together or the histogram will be very bumpy.
- In the next problem you'll probably need the `outer` command, which does an outer product. We'll just use it to duplicate columns of a matrix some number of times. Here's an example: `outer(arange(10),ones(3))`.

■

3. Biased and Unbiased Estimators (5 pts)
Demonstrate an example with plots showing the following:

- You take a few samples from a distribution and find their average. This average is an estimator for the true mean of the underlying distribution. We'll see below how to estimate the variance. To quote Wikipedia[a], "the *bias* of an estimator is the difference between this estimator's expected value and the true value of the parameter being estimated." The average of a few samples is an unbiased estimator—if you take 5 samples and average them together and you do this process a million times, these averages will be distributed symmetrically around the true distribution mean.

- An **unbiased estimate of the mean** of a distribution is the average of samples drawn. Try drawing something like 2, 5, and 100 samples from the uniform, normal, and exponential distributions for each "experiment". Simulate a million experiments, histogram the averages, and show that the mean of the averages is quite close to the true mean. (This is basically identical to what you did above.)

- If you know the true mean μ (rare in practice, but sometimes you can know that it's some special value like zero from symmetry), an unbiased estimate of the variance is just the average of the squares of the deviation from the mean:

$$s^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2$$

- If you don't know the true mean μ and are just estimating it using the average of the data $\overline{x}$, the **unbiased estimate of the variance** needs $N-1$ instead:

$$s^2 = \frac{1}{N-1}\sum_{i=1}^{N}(x_i - \overline{x})^2$$

  If you stick with N instead of $N-1$, you get a biased estimate of the variance.

- Show examples where $N = 2$, 5, and 100 for three cases: where the true mean gives an unbiased estimate of the variance, where N and the sample mean leads to a biased estimate, and where $N-1$ and the sample mean leads to an unbiased estimate. You'll need to draw a million-by-N samples from a distribution with a mean other than zero (like a normal distribution with a mean of 1), subtract either the true mean or the sample mean (this is where `outer` is helpful), square, sum together 2, 5, or 100 numbers a million times, plot the distribution of the s^2 values that you calculate, and finally find the average s^2 value. The distributions haven't converged to gaussians for small N, but the average s^2 should always be around the true variance for the unbiased estimators. Conceptually, why is this? Why should we divide by a slightly smaller number here? See `https://en.wikipedia.org/wiki/Estimator_bias` or `https://en.wikipedia.org/wiki/Bessel%27s_correction`.

- Either way, s itself, once you take the square root, is unfortunately a biased estimate of the standard deviation. Show examples of this. Mathematically, the variance is almost always easier to work with. Unfortunately, the standard deviation is usually nicer to report because it has the same units as the thing you are measuring—the variance has those units squared.

- The standard error of the mean (SEM) or simply "standard error" measures how far the sample mean (average) of N data points is likely to be from the true population mean. This is $s/\sqrt{N}$. Yes, you divide by another factor of $\sqrt{N}$ because your estimate of the mean gets better and better the more data you take, even though you are sampling from a distribution with the same underlying standard deviation. No, this additional factor is not $\sqrt{N-1}$.

4. Linear Least-Squares Fit Part 1 (5 pts)
Say you've collected the following data by making measurements at 7 different x locations. You did each measurement many times and found the mean and standard error at each point, as described above. The data (means) with standard errors turned out to be:

```
xdata = array([    0,    10,    20,    30,    40,    50,    60])
ydata = array([ 43.5,  79.4, 156.0, 326.1, 196.5, 368.1, 412.7])
yerr  = array([ 30.7,  13.6,  79.9,  65.2,  84.2,  16.1,  27.2])
```

- Make a plot of the data as points with error bars using the `errorbar` plotting function.
- Fit the data using the code below. (Type these in yourself because copy and paste don't work well with the LaTeX Python formatting for some reason.)

```
from scipy.optimize import curve_fit

# Define the function to fit.
# The first parameter is the independent variable.
# The remaining parameters are the fit parameters.
def line(x,m,b):
    return m*x + b

# Do the fit. Read the help on curve_fit!
(popt,pcov) = curve_fit(line, xdata, ydata, sigma=yerr, absolute_sigma=True)
# popt now holds the optimized parameters m in popt[0] and b in popt[1]
# pcov is the covariance matrix, which gives errors and correlations.
# To extract just the errors on the fit parameters as sigmas:
perr = sqrt(diag(pcov))
# now perr holds the +- 1sigma error for m and b.
```

- Plot the fit on top of the data with error bars:

```
# First plot the data with error bars
errorbar(xdata, ydata, yerr, fmt="o", label="data")

# There are many ways to plot the line given m and b, but here's one:
yfit = line(xdata, *popt) # * passes a list as remaining function parameters
plot(xdata, yfit, label="fit")

legend()
title("ALL OF YOUR PLOTS MEANINGFUL TITLES")
savefig("hw1_meaningful_file_name.pdf") # PDFs can be used as LaTeX figures
```

- Calculate the residuals and plot those with error bars: `residuals = ydata - yfit`. It sometimes helps to draw a horizontal line at $y = 0$: `hlines(0, min(xdata), max(xdata))`. If everything is working, the line should be within 1 sigma of the data point 68% of time. In other words, the line should cross through an error bar around 2/3 of the time.
- Calculate and plot the normalized residuals with normalized error bars (normalized error bars are all ± 1). `normalized_residuals = residuals/yerr`. Again, if everything is working, the line should be within 1 sigma 68% of time. Again, draw a horizontal line through zero. These plots help you see if you model is qualitatively the right one.
- Some students think that making the error bars bigger when making measurements is a conservative and safe thing to do. This is incorrect. If the errors are statistical in nature and calculated properly, and the model is correct (any systematic errors are added as fit parameters), around 68% of the points should be within 1σ. If your error bars are too big, the data points will be too close to the same best-fit line and the fit will be "too good," which we'll explore next.

5. Linear Least-Squares Fit Part 2 (5 pts)

- Calculate χ^2, the sum of all of the normzlied residuals squared: `chi_squared = sum(normalized_residuals**2)`

$$\chi^2 = \sum_{m=1}^{M} \frac{(y_m^{\text{data}} - y_m^{\text{fit}})^2}{\sigma_m^2}$$

  The χ^2 is the total number of sigmas that the data points are away from the fit. You might think that a great fit would have a really low χ^2 because the line goes right through the center of all of the error bars. This actually means that you overestimated your error bars and were drawing from a narrower distrubtion than you reported. The χ^2 should be distributed according to the appropriately-named Chi-square distribution.

- Calculate the reduced χ^2 (also known as χ^2 per degree of freedom). This is χ^2 divided not by M (the number of data points), but M minus the number of fit parameters (2 in this case). The reduced χ^2 should be, on average, around 1.0 because each measurement point should be around 1σ away from the truth on average. Check this.

```
M = len(xdata)  # number of data points
df = M-len(popt) # M-2 degrees of freedom: M data points minus 2 parameters
reduced_chi_squared = chi_squared/df
```

- Look up the probability that you'd randomly have found a χ^2 of this size or *bigger*. This is a job for the χ^2 distribution's Cumulative Distribution Function:

```
from scipy.stats import chi2
PTE = 1-chi2.cdf(chi_squared, df=df) # Probability to Exceed
```

  The probability `PTE` is the "probability to exceed" because it's the probability that a random χ^2 would exceed the χ^2 that you found for your fit. This is meaningful only if *both* the underlying model is correct (linear here) *and* you estimated your errors correctly (which you automatically did by artificially drawing from the correct distribution). There is a one-to-one mapping between reduced χ^2 and PTE. Often only one or the other gets reported.

  – A really *low* PTE like 0.001 means that your χ^2 is too *big* (it's unlikely that a random χ^2 would be this big or bigger). It's very unlikely that you'd get such a high χ^2 by chance. It often means that the error bars that you used are too *small* compared to the actual underlying statistical errors in a sequence of honest measurements.

  – A really *high* PTE like 0.999 means that your χ^2 is too *small* and often means that the error bars that you used are too *big*. This often happens when people think they are being conservative with their error bars and incorrectly adding a little extra: "Surely if my ruler is only marked in millimeters, my error bar shouldn't be smaller than a millimeter." This is not true if you are consistently and correctly estimating fractions of a marking.

  – A good fit has a PTE between 5% and 95%, which should happen 90% of the time.

  – An OK fit has a PTE between 1% and 99%, which should happen 98% of the time.

  – Note that 2% of the time you get unlucky: you did everything right, but your particular measurements happened to group together or spread out more than they "should".

6. Linear Least-Squares Fit Part 3 (5 pts)
Now for the real fun.

- Do the entire above process at least 10000 times (maybe 1000 while you're debugging). Start by playing god and picking some particular physical constants for the true slope `m` (say 7) and true intercept `b` (say 50). Keep the same `xdata` and `yerr`. Initialize any variables that you need to record things as empty lists. Each time through your loop:
  - Draw random `ydata` from a Gaussian centered around the true values (`y = m*xdata+b`) with the given sigma (`yerr`) at each point.
  - Do the fit
  - Save the fit parameters and their errors by appending to the lists.
  - Calculate the residuals and normalized residuals.
  - Calculate and save the χ^2, reduced χ^2, and PTE.
- Show that your set of reduced χ^2 has a distribution peaking around 1. Show that the mean is really close to 1.
- Show that your set of PTEs has a flat distribution from 0 to 1. A good fit has a PTE between 5% and 95%, which should happen 90% of the time. An ok fit has a PTE between 1% and 99%.
- Show that your set of parameter estimates for the slope and intercept each have a Gaussian-like distribution around the true value.
- Show that the standard deviation of these parameter estimates is close to the mean of your estimated errors (in `perr`). Draw horizontal lines that go ± 1 average `perr` on either side of the true value.
- How often was the parameter estimate within the 1σ parameter-error of the true value? It had better be around 68% of the time if the previous two points hold.
- Explore what happens to these histograms when you make the error bars artificially twice as big and half as small. Do this by copying and pasting your code and artificially doubling and then halving the error bars that you send to the fit and to your calculation of the normalized residuals (but not the distribution that you use to draw the data). In effect, you're lying to the fit about the distribution from which the data points were drawn. It will fit the *same line*, but the errors on the parameters, the χ^2, and the PTE will be artificially increased or decreased.
- If you did a fit and found a really low PTE like 0.001, assuming your model is correct, are your error bars too big or too small?
- What if you found a really high PTE like 0.999?

■