CS 111: Homework 6: Due by 11:59 pm Sunday, November 7, 2021

Submit your paper as one PDF file, and tell GradeScope which page(s) each problem is on. If you worked with a partner, you must each separately write up and turn in your own homework paper, and report the name of your partner. No groups of more than two.

1. This problem is about using the least squares method for fitting a function of a given form to data from measurements. The two main goals here are (1) to learn to use the numpy routine `npla.lstsq()`, and (2) to learn to make well-designed plots of the data and the least-squares fits to the data. We'll talk about the algorithms behind `npla.lstsq()` in class later this week, and you'll experiment with them on next week's homework.

1.1 The data we'll use is NASA's annual measurements of the so-called "temperature anomaly", which is how much the mean global temperature for a given year differs from the average for the reference years 1951-1980. You can read about this data at `https://data.giss.nasa.gov/gistemp/`. For our purposes, I've downloaded the data as a JSON file called `annual_temps.json` that's with the homework on our GitHub site.

Use the routine `cs111.get_gistemp()` to read the temperature data. It returns a numpy array of years (from 1880 to 2016), and an array of Celsius temperatures. Make a graph that plots the data as dots, one per year, with the years on the horizontal axis extending as far as 2040. Label the axes with a description of what they represent (year and temperature anomaly); put a good descriptive title on the plot; and include a legend (using `plt.legend()`) indicating that the dots are the measured data. Turn in your plot and the python code you used to produce it.

1.2 Here we'll fit a straight line to the data, of the form

$$f(y) = x_0 + x_1 y,$$

where y is the year. We want to choose x_0 and x_1 so that, if the measured temperature in year y_i was t_i, we have $f(y_i) = t_i$. However, the 137 points (y_i, t_i) don't all lie on a straight line, so we can't satisfy this exactly for all i. We will find the values of x_0 and x_1 that are best in the *least squares* sense, meaning that they make the value of the squared error norm

$$\sum_i (f(y_i) - t_i)^2$$

as small as possible.

Let's say that in terms of matrices and vectors. Let t be the column vector $(t_0, \ldots, t_{136})^T$ of measured temperatures, one per year. Let A be a matrix with two columns, with one row for each year. In row i of A, the first entry is a 1 and the second entry is the year y_i. Then if $x = (x_0, x_1)^T$ is the 2-vector of unknowns,

the product Ax is the vector of the 137 values $f(y_i)$, which we want to be as close to t as possible. Thus the problem we need to solve is to find the vector x that minimizes the 2-norm of the residual,

$$\min_x ||Ax - t||_2.$$

This is exactly the problem that `npla.lstsq(A, t, rcond = None)` solves. (The `rcond = None` just suppresses a warning message about different versions of the routine.)

Use `npla.lstsq()` to find the minimizing x for the NASA data from 1880 to 2016. Make another plot, which has the same data points as your plot from (1.1), and also has the straight line representing $t = f(y)$. Plot the straight line all the way from $y = 1880$ to $y = 2040$. Update the legend to describe what the line is in addition to the dots. Turn in your plot and the python code you used to produce it. Report the value of x. Finally, report what the temperature anomaly would be in 2040 if it were on the least-squares fit line.

1.3. You might notice that linear extrapolation along the least squares line badly undershoots the actual temperature anomalies in recent years, so the predicted 2040 value you computed above is not very credible. Make a third plot that adds two more straight lines to your plot from (1.2), one of which fits only the data since 1970 and one of which fits only the data since 2010. Plot both of the new lines all the way out to 2040, and add descriptions of them to the legend. (This plot should still show all the data points from 1880 on, as well as the original line you computed in (1.2).) Turn in your plot and the python code you used to produce it. Report the values of x for each new line, and report the predicted 2040 temperature anomaly according to each line.

1.4. Although these latest predictions look somewhat better than the first one, it really doesn't seem like the data follows a linear trend. Let's try fitting a quadratic (a parabola) to the data, of the form

$$f(y) = x_0 + x_1 y + x_2 y^2.$$

At first glance this doesn't look like a "linear" least squares problem, but it is, because the unknowns x appear linearly. Thus we can solve it in a matrix formulation as before, by finding the minimizer

$$\min_x ||Ax - t||.$$

This time x is a 3-vector $(x_0, x_1, x_2)^T$, and A is a matrix with three columns. What's in the third column of A? The squares y_i^2, which are the coefficients of x_2 in $f(x) = Ax$.

Use `npla.lstsq()` to find the 3-vector x that describes the quadratic least-squares fit to the data, using all the years from 1880 to 2016. Make one more plot that shows the data (as dots), the 1880-2016 linear fit (as a line), and the 1880-2016 quadratic fit (as a parabola). (To make the graph less noisy, you can omit the other two linear fits you found in (1.3).)

Hint: To draw the parabola $f(y) = x_0 + x_1 y + x_2 y^2$ over the range 1880-2040, you can plot a piecewise linear curve that goes through the values of $f(y)$ for a suitably chosen set of y values. You can use `np.linspace()` (see its docstring) to generate evenly spaced y values as close together as you want.

Turn in your suitably labelled plot and the python code that produced it. What are the values of the quadratic coefficients x? What does the quadratic fit predict as the temperature anomaly in 2040?

2. Recall from lecture (or NCM section 2.9) the definition of the condition number $\kappa(A)$ of a matrix. Let

$$A = \begin{pmatrix} 1 & 0 \\ 0 & 1000 \end{pmatrix} \quad \text{and} \quad B = \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix}.$$

2.1. What IEEE 64-bit floating-point number represents $\kappa(A)$? Give your answer both as an ordinary base-10 number, and also as a 64-bit word (16 hexadecimal digits). You may obtain the latter from `cs111.print_float64()`.

2.2. What IEEE 64-bit floating-point number represents $\kappa(B)$? Give your answer both as an ordinary base-10 number, and also as a 64-bit word (16 hexadecimal digits).

3. These questions are all about IEEE standard 64-bit floating-point arithmetic, which is behind both numpy's `float64` type and C's `double` type. You can use `cs111.print_float64()` to see the actual bits that represent any number. Recall that one hexadecimal digit stands for 4 bits.

3.1. *Machine epsilon*, or just ϵ for short, is defined as the smallest floating-point number x such that $1 + x > 1$ in floating-point arithmetic. The experiment we did in class showed that ϵ is approximately 10^{-16}, which is why we say that IEEE floating-point can be accurate to about 16 decimal digits.

What is the exact value of ϵ? (Give your answer as an exact arithmetic expression involving powers of 2, not a decimal expansion.) What is the 16-digit hex representation of ϵ in the IEEE standard? (Use `cs111.print_float64()`.) Approximately how close is ϵ to 10^{-16} in relative terms? (Answer this by computing $|\epsilon - 10^{-16}|/\epsilon$.)

3.2. What is the 16-digit hex representation of $1/\epsilon$? What is its approximate value in base 10? (Here you should give a decimal expansion.)

3.3. What is the largest non-infinite positive number that can be represented exactly in IEEE floating-point? Give your answer three ways: As an exact arithmetic expression involving powers of 2; as the 16-hex-digit IEEE representation; and as an approximate value in base 10. (Hint: Consider the largest possible value of the 11-bit exponent field in the IEEE standard, but remember that value is reserved to represent "infinity".)

3.4. A common mistake some people make (but not you!) is to think that ϵ is the smallest positive floating-point number. It's not, by a long shot. Consider for example $x = \epsilon^{10}$. What is the approximate value of x in base 10 (as a decimal expansion)? Is x an exact floating-point number? If so, give its IEEE 16-hex-digit representation; if not, give an IEEE 16-hex-digit representation of a floating-point number as close to x as you can.

3.5. How many different floating-point numbers x are there with $1 < x < 2$? How many with $4096 < x < 8192$? How many with $1/64 < x < 1/32$?

4. For this problem, you may want to read the article "Tearing apart Google's TPU 3.0 AI coprocessor" (under Readings on the course GitHub site), especially the section "TPU Chips" that starts on page 9.

The 1985 IEEE floating-point standard specifies a 16-bit version and a 32-bit version as well as the ubiquitous 64-bit version. The 16-bit version was never used much before about 2010, because most scientific modeling requires higher precision. Recently, though, 16-bit floating-point has become popular in machine learning applications, because the weights in a neural network don't need to be determined very precisely.

IEEE standard 16-bit floating-point uses 1 bit for the sign, 5 bits for the exponent, and 10 bits for the mantissa. However, when Google developed its TPU ("Tensor Processing Unit") chip for machine learning, it used a different 16-bit format that it calls "bfloat", with 1 bit for the sign, 8 bits for the exponent, and 7 bits for the mantissa.

4.1. How many different floating-point numbers x are there with $1 < x < 2$ in IEEE 16-bit floating-point? In bfloat?

4.2. What is machine epsilon for IEEE 16-bit floating-point? For bfloat? (Show your answer both as an exact mathematical expression and as an ordinary base-10 decimal approximation. You don't need to show the 16-bit hex representation unless you want to.)

4.2. Assume that both IEEE 16-bit and bfloat treat exponents the same way IEEE 64-bit does, with the largest exponent reserved for infinity and all the exponents shifted to represent about the same number of negative exponents as positive exponents. (I don't actually know whether this is true for bfloat.) What is the largest non-infinite positive number that can be represented exactly in IEEE 16-bit floating-point? In bfloat? Give your answers both as exact arithmetic expressions and as approximate base-10 numbers.

4.3. Name one advantage of bfloat over IEEE-16, and one advantage of IEEE-16 over bfloat.