ECE 157A - Homework 1 - Getting Started

ECE 157 TAs

Fall 2021

This setup guide will walk through the basics of loading and processing machine learning data in *modularized* and *repeatable* ways.

Contents

1 Brief advice

- Your number one tool will be web search (e.g. Google, DuckDuckGo, Bing, etc.) Web search error messages, web search things you want to do with your code, web search functions you don't recognize from any of my examples, anything. In writing these assignments, we were using web searches all the time. You can't be expected to know every facet of every tool. This is true of all programming, not just this class.
- You're here to learn about the ML tools and workflows of building ML applications. If you're getting hung up on Python's syntax, come to us in labs or office hours.
- Start early and come to lab. It's way harder to work on Numpy matrix reshaping at 2300 on the due date.
- If you can't finish in lab, do *something* each day the assignment is assigned. Even a 10 minute block can refresh your memory of both your own code and the topic you're working with, improving your productivity when you finally do have time.
- Whenever we give you a code block, like

  ```
  print("Hello, world!")
  ```

  it's a good idea to type the line yourself instead of copying it. The `PDF` file format uses unicode characters for "" and ' ' that are not understood by Python.

2 Installation

2.1 Checking your Python version

WARNING: DO NOT INSTALL PYTHON 3.10. As of the writing of this guide, Matplotlib and Sklearn do not install correctly in Python 3.10.

2.1.1 MacOS/Linux

First, open a terminal. Run the command

```
python3 --version
```

If your shell reports the command is not found or spits out an error, follow these installation instructions. If your Python3 version is less than 3.6, it is recommended (but not necessary) to follow these instructions to install a newer version.

If you have an up-to-date version of python, move on to section 2.3.

2.1.2 Windows

Open PowerShell from the start menu. Run the command

```
python --version
```

If it spits out a new enough version, you're good to go! If it doesn't output anything, it's likely that Python is not installed on your computer. Follow the instructions in the next subsection to get Python3 installed on your computer.

If your Python3 version is less than 3.7, it is recommended (but not necessary) to follow these instructions to install a newer version.

If you have an up-to-date version of python, move on to section 2.3.

2.2 Getting Python3

WARNING: DO NOT INSTALL PYTHON 3.10. As of the writing of this guide, Matplotlib and Sklearn do not install correctly in Python 3.10.

Go to `https://www.python.org/downloads/` and download (*almost*) any version of Python released in 2021. (Links to the download pages are in the second table on the webpage. The files are on the bottom of the download pages.)

WARNING: DO NOT INSTALL PYTHON 3.10. As of the writing of this guide, Matplotlib and Sklearn do not install correctly in Python 3.10.

- MacOS: You want the *MacOS 64-bit installer* for your selected version
- Windows: You likely want the *Windows x86-64 executable installer*
- Linux: I assume you know what you're doing.

The default options should be sufficient for this class. If the installer asks about the system "PATH" variable, select the option that adds python to it. If it doesn't ask, don't worry about it!

After installing, try section 2.1 of this guide again. If python is still showing as missing or outdated, reboot your computer and try one last time. Then reach out to your TAs for help!

2.3 Getting the tools

2.3.1 MacOS/Linux

To install tools used in this assignment and guide, you should only need to run:

```
python3 -m pip install numpy sklearn matplotlib
```

If you get error messages from this, give the error a web search and try again. Often your exact problem will have already been covered on StackOverflow or another quesiton/answer website!

2.3.2 Windows

To install tools used in this assignment and guide, you should only need to run:

```
python -m pip install numpy sklearn matplotlib
```

If you get error messages from this, give the error a web search with your OS and try again. Often your exact problem will have already been covered on StackOverflow or another quesiton/answer website!

3 First steps

3.1 Hello World

Here's the entire file for a "Hello world" program in Python:

```
print('Hello, world!')
```

That's it! Type that line into a file, say *helloworld.py*. On Windows, you can then execute the python program with

```
python helloworld.py
```

On MacOS/Linux, you will need instead

```
python3 helloworld.py
```

This should do exactly what you expect. If it doesn't, remember to *type* code we give you, rather than copy-pasting. PDFs use unicode characters for "" and ' ' that are not understood by Python.

3.2 Numpy and Matplotlib

Numpy is short for "Numeric Python". It provides facilities to work with arrays of data. To show off how this works, we're going to generate some data to play with!

First, to use numpy, we have to *import* it. We do this with the line

```
import numpy as np
```

This tells Python to load the code associated with the Python package `numpy` into your program. It puts functions and classes in Numpy under the name `np` so you can access them with dot-notation, e.g. `np.float64` for Numpy's 64-bit floating point datatype. If you left off the `as np`, you could still get to float64 with `numpy.float64`

Next, we'll do the same thing for another tool:

```
import matplotlib.pyplot as plt
```

Matplotlib provides tools for graphing data in "iterable" things – things that contain values, e.x. lists and Numpy arrays.

With all of this loaded, we'll now make our first numpy array!

```
xs = np.arange(0.0,10.0,0.5,dtype=np.float64)
```

The function `arange` takes a start value, a stop value, and a step size. It gives back a numpy array starting at the starting value and increasing by the step size until it makes a number equal to or bigger than the ending value, which is not included.

We assign that array into the variable `xs` for future use.

> *Practice Question*:
>
> If we note down the starting value, then add the step size to get the next value in the array, then end the array after (that is, excluding the first value greater than or equal to) the stop value, how many values should be in the array `xs` given by the above code?
>
> Answer: remember that we start at 0.0, so the next is 0.5, then 1.0, etc. Thus, we have every half and every whole number between 0.0 and 10.0. ten half numbers (0.5 counts!) plus nine whole numbers. 0.0 is also included, so nine plus ten plus one makes twenty values.

Next, we'll do something with these values. Numpy includes copies of common math functions, one of which is `sin`. Lets convert our `xs` into `ys` with the equation, $y = \sin(x)$. That looks like:

```
ys = np.sin(xs)
```

Yep! That easy. This tells Numpy to perform the `sin` operation on every element of the "array-like" `xs`. "array-like"s are anything Python can iterate over, but *also* single values. That is, `np.sin(5.0)` gives exactly the result you'd expect.

Numpy stores the result of operation in a new array, which it assigns to `ys` according to our assignment. Each y value in `ys` corresponds exactly to the element it came from in `xs`, that is, the two arrays are in the same order.

This automatic and very fast operation on "array-like"s (including lists and scalars) makes Numpy really powerful!

> *Practice question*:
>
> The first value in `xs`, the array created by `arange`, was 0.0. The last value in `xs` was 9.5.
>
> What is the first value of `ys`? The last value?
>
> Answer: Recall that `ys` is in the same order as `xs`, but each element has had sin applied to it. Then the first element of `ys` is sin(0.0), and the last is sin(9.5).

Finally, just understanding that the numbers are in the computer somewhere doesn't help us. Can we see the values somehow? The `print` command comes back to help! Any Python object that supports the `str` method can be printed, and Numpy arrays are one such object! Then, to see our Numpy arrays in the program's output:

```
print(xs)
print(ys)
```

You can run the python script now to see the output if you'd like. Unfortunately, it's difficult to clearly visualize what's happening with the results of the computation. Though Numpy wraps the array for us to keep it visible in our terminal, it doesn't help the human brain that much to stare at a bunch of numbers.

This is where our last import, *Matplotlib*, comes into play. With it, we can plot our results visually:

```
plt.plot(xs,ys)
plt.show()
```

That's it! (Note that `plt.show()` is necessary. Otherwise the program doesn't wait for you to *see* the figure before ending!)

Now, when run, the program will pop up a graph of the result of the sin function from 0.0 to 9.5, at increments of 0.5. (See subsection 3.2) You may notice that the graph has sharp corners, rather than the smooth curve you may know from math class. This is because we *discretized* the input. This is necessary, because digital computers cannot describe continuous equations like sin. To improve the graph's accuracy, try reducing the step size you gave to `arange` when making `xs`!

Our final program:

firstarrays.py

```
import numpy as np

import matplotlib.pyplot as plt
import numpy.random

xs = np.arange(0.0, 10.0, 0.5)

ys = np.sin(xs)

print(xs)
print(ys)

plt.plot(xs, ys)
plt.show()
```

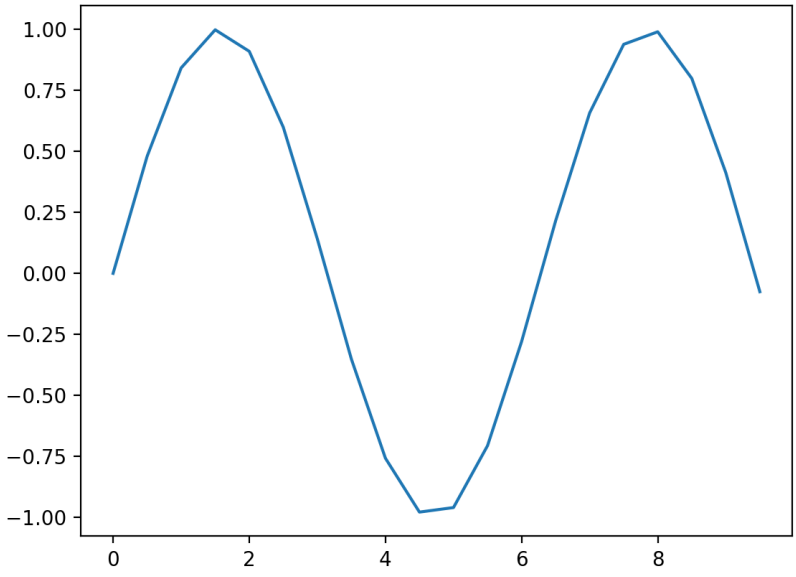

Figure 1: The expected figure from firstarrays.py

3.3 Indexing and Slices

3.3.1 Indices

Numpy arrays wouldn't be all that useful if the values could only be drawn or printed out after you make them. Luckily, Numpy lets you access the individual values inside the array via something known as *indexing*. You can also access or swap continuous sections of the subarray through something known as *slices*.

Indexing for a particular value is easy. Starting with our *firstarrays.py* above:

```
print(ys[2])
```

The statement `ys[2]` is asking Numpy for the element at index `2` in the array `ys`. This syntax may be familiar from C pointer arithmetic, or Matlab matrices – it's more in line with the Matlab history with one major difference:

Indexes start with 0

That is, the first element in the array is at index 0. The second is at index 1, and so forth. This is related to how C stores arrays, and is the more common indexing schema of most programming languages.

Practice question:

Looking back on *firstarrays.py*, what would `ys[2]` evaluate to? Recall that the first few elements of `ys` are [sin(0.0), sin(0.5), sin(1.0), sin(1.5), ...etc].

Answer: The first element is at index 0. The second is at index 1. That puts index 2, the one we were asking for, pointing at the third element of the array: sin(1.0).

3.3.2 Slices

That shows us how to retrieve individual values, but what if I want the subarray of sin values between 0.0 and 5.0? For this, we use *slices*. A slice defines an index that we start cutting at, and an index we stop *before including* Here are some identical slices of the `xs` and `ys` arrays:

```
print(ys[0:10])
print(ys[:10])
print(np.sin(xs[0:10]))
print(np.sin(xs[:10]))
```

Note how each slice is defined by the `:` (colon). That only generates ranges of indices specifically! To make ranges of numbers elsewhere in Python, you must use the integer `range` or the `np.arange` we learned about before.

The first slice is hopefully clear: We cut out every element starting at index 0 and ending just before index 10. Index 10 itself is not included. This also tells us what the third slice is doing; `np.sin` is reading the slice over the same elements of `xs` into a new array, exactly like when we made `ys` in the first place. We've just reduced the number of elements on which `np.sin` is operating.

What's up with the second and fourth slices, though? When an index is left off of the slice definition, Python assumes it's the furthest index in that direction. Thus, `0:10` and `:10` describe the same slice, because the index `0` comes first! Thus, both the second and fourth slices are identical to the other two.

3.3.3 Negative indices

This raises a question: what about negative indexes? Python handles negative indices by counting backward from the end of the list! With this, to get the last five elements of an array (no matter the array's size!) is simply:

```
print(xs[-5:])
```

There's one caveat to this. Negative indices are replaced with their positive counterparts. Thus, a slice cannot cross the end of a list and circle back to the beginning. Try it:

```
print(xs[-5:5])
print(xs[-5:16])
```

The first will give you `array([])`. The second will give you `array([7.5])`. This is because -5 is replaced with len(xs)-5, which with the default *firstarrays.py* is 15, the index of 7.5. Thus, the two above slices are equivalent to:

```
print(xs[15:5])
print(xs[15:16])
```

This demonstrates another important point: slices do not cut backward (by default.) If given the same index twice or indices in the wrong order, it will instead return an empty array, because no values (even the beginning index) can be included without including a value greater than the stop.

> *Practice question*:
>
> For the array `xs = np.arange(0.0,10.0,0.5)`, what are the following slices?
>
> - xs[:]
> - xs[-1:-5]
> - xs[-5:-1]
>
> - If we leave off an index, the slice assumes we mean to include all elements in that direction. Thus, the slice with no indices includes all elements in the array, meaning the slice is equivalent to `xs` itself.
> - `xs` is 20 elements long. Thus, the slice is equivalent to `xs[19:15]`. Because 19 > 15, there are no elements included in the slice, so we get `array([])`.
> - `xs` is 20 elements long. Thus, the slice is equivalent to `xs[15:19]`. This gives us the last five elements of xs, except for the last one, aka: `array([7.5,8.0,8.5,9.0])`

3.3.4 Slice step size

One might wonder, though: Say I want the elements in an array in reverse order, or I want only every other element of the array. Are those slices possible? Yes.

Remember that we noted before that slices cannot cut backward *by default*. The default *step size* of a slice is 1, meaning each index is the previous index plus 1. We can change that step size for a given slicing by extending the slice syntax:

```
print(ys[10:5:-1])
print(np.sin(xs[10:5:-1]))
```

Both of these give the same result: sin(5.0), sin(4.5), sin(4.0), sin(3.5), sin(3.0). Notice that this is counting backward, just as we'd hoped! Better yet, Numpy has no problem passing this backward slice through `np.sin`, as once the slicing is complete, we're left with a regular array-like!

We can set the forward and backward step sizes to any integer:

```
print(xs[5:10:2])
print(xs[10:5:2])
print(xs[10:5:-2])
print(xs[5:10:-2])
```

The first retrieves 2.5, 3.5, 4.5. The second retrieves nothing, because $10 > 5$ and we're stepping in the positive direction, so we can't get down to 5. The third retrieves 5.0, 4.0, 3.0. The fourth retrieves nothing, because $5 < 10$ and we're stepping in the negative direction, so we can't get up to 10.

Non-integer step sizes and step sizes of zero will throw errors.

3.4 Multidimensional Arrays

All of these arrays are fine and dandy, but it can be annoying to keep track of dozens of them. If we know we'll have two lists of numbers of the same length, can we keep them in one object? Yep!

Make a new file, *secondarrays.py*, and copy over all code from *firstarrays.py* above. After the line creating `ys`, add the lines:

```
print(xs.shape)
print(ys.shape)
ys = np.stack((ys,np.cos(xs)))
```

When you run *secondarrays.py* You'll get... an error message! Uh oh! Take a look at what the error says.

In Python, errors give you two parts: A **traceback** and the **error message** itself. The traceback lists the function calls you made along the way. At the top of the traceback (the oldest call) we see that it's choking when we call `plt.plot(xs, ys)`. The error message says "x and y must have the same first dimension, but have shapes $(20,)$ and $(2, 20)$.

Why did this happen? If we look higher in the code, we ask for the shapes of `xs` and `ys`. Both of them were $(20,)$, right? This is known as a "1-dimensional" array, where we have twenty elements along `axis=0` and no other axes. However, we *reassign* the name `ys` to instead be the result of `np.stack`ing `ys` with another array: `np.cos(xs)`. When we stacked them, that made a 2-dimensional array. `axis=0` now had two arrays, each of which has 20 elements.

MatPlotLib is telling us that, when we hand it `xs` and `ys`, they must both have the same first dimension. We have two ways to fix this:

- add `ys = np.transpose(ys)` to transpose the array
- change the stacking to `ys = np.stack((ys,np.cos(xs)),axis=1)`

The first rotates the array ys so that the first and second axes are swapped, exactly like you know from transposition of matrices in Linear Algebra. The second will tell `np.stack` to create the new dimension as `axis=1`, the second axis, so we don't need to fix the rotation later!

The latter is more efficient, so let's implement that. Replace the line

```
ys = np.stack((ys,np.cos(xs)))
```

with the line

```
ys = np.stack((ys,np.cos(xs)),axis=1)
```

Now when you run the program, you should see a cosinusoid dancing around our original sinusoid. Great!

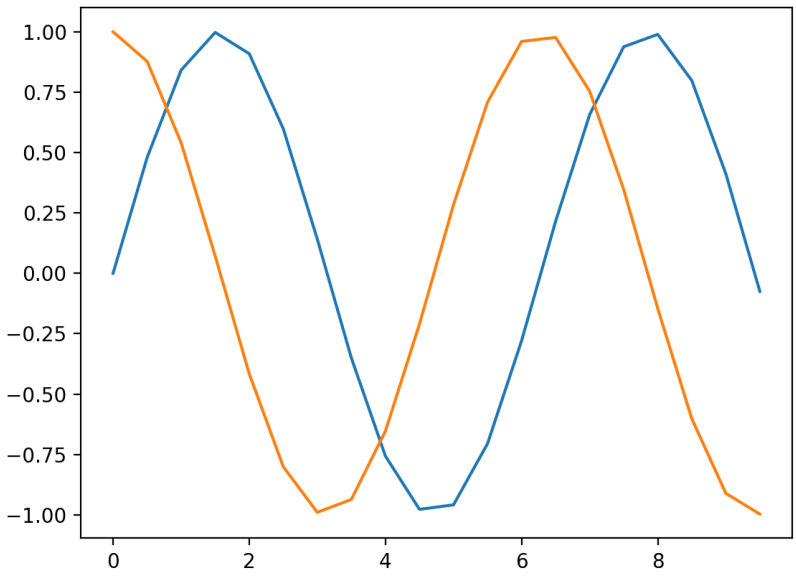

Figure 2: The expected figure from secondarrays.py

Next, we'll walk through loading the assignment data and setting up the required **DecisionTree** classifier, as an example for the other classifier options.

4 Building your Classifier experiment

4.1 Loading data

We recommend that you have a function to load your data files, so that you can call it from elsewhere. By keeping it in a separate file from the rest of your work, you can keep access to it modularized, to more easily reuse the code later!

Create a file called *preprocessing.py*. We'll fill this file with utility functions that load your data and cut it up, so that every place in your experimenting does it the same way.

Also create a file called *test.py*. The first line of *test.py* should be `import preprocessing`. Now, any function you write in *preprocessing.py* will be available in *test.py* by just typing `preprocessing.funcname`, just like with Numpy!

In *preprocessing.py*, we'll want to read the CSV file we're given. To do that, we need the Python **CSV** module, which is automates a lot of the boring stuff in that process. As before, getting the CSV file is just:

```
import csv
```

No need to rename it, because the name is already so short! We'll also want to have Numpy, as its datatypes are faster than Python's:

```
import numpy as np
```

Next, defining a function in Python looks like this:

```
def load_data(filename='diabetes.csv'):
```

This is a function that takes a variable called `filename` which, by default if not otherwise provided, is set to the string "diabetes.csv". Note the colon at the end of the line! This indicates the beginning of a "block" of instructions inside the function. The code to be run inside of this function is all code indented *four spaces* from the indentation of the function definition. So, if we were to write:

```
def joke_load_data(filename='diabetes.csv'):
    print("Nah, I won't load ",filename," today.")
print("I print out only once!")
```

Then, anywhere in the program, we can write `joke_load_data('joke.csv')` to get it to print in reply, `"Nah, I won't load joke.csv today."` We can also call `joke_load_data()` (without its argument) to print the default, `"Nah, I won't load diabetes.csv today."` However, the line `"I print out only once!"` is printed only once, when the python interpreter gets to its line.

Now we'll finish the first few lines of `load_data`:

```
def load_data(filename='diabetes.csv'):
    with open(filename,'r') as file:
        reader = csv.reader(file)
        columnNames = next(reader)
        rows = np.array(list(reader), dtype=float)
```

In Python, the `open` function opens a file. The second argument, `'r'`, tells `open` that we only need to read the file.

The `with` block manages some resource (in this case, the open file `file`) and makes sure it's automatically closed at the end of the block. We need to "close" a file to give it back to the operating system we're running on, as every open file costs a bit of the computer's capabilities. Files are closed automatically when the program crashes or ends, but it's good practice to use `with` blocks to close them as soon as they're no longer needed!

`csv.reader` takes a file and returns a *generator*. A generator knows how to make all values in some sequence. In this case, the sequence is the rows of the CSV file. We then ask for the next item in the generator with `next`. Because we haven't asked for any yet, this gives us the first row of the CSV file. If you open the CSV file, you'll see that this is the header, listing the column names!

Finally, we have the line handing the reader to the `list` function, then to `np.array`. Both are a little fancier than regular functions, but for now all we need to worry about is that `list` takes the generator and returns a list of every item the generator has left to create – that's all the rows of the CSV file after the header! Then `np.array` takes the list, which is array-like, and converts it to a real Numpy array. We also specify that the Numpy array's datatype should be `float`, a default Python floating-point number, which should be the fastest for your platform.

Now we've successfully extracted the column names and rows of our file! To pass the data back to our caller, add one more line to the function:

```
    return columnNames, rows
```

Now over in *test.py*, we can call this function to access the loaded data:

```
import preprocessing

columnNames, rows = preprocessing.load_data('diabetes.csv')
print(columnNames)
print(rows)
```

Now, if you run *test.py*, it'll load the data and print it to the console. Yay!

4.2 Separating samples and labels

Now we have the data extracted from the file, but it's jumbled up. To train a classifier, we need "samples" and "labels" as separate lists.

In *test.py*, try printing out just the column names. One of these, **Outcome**, is the one we're trying to predict! We need a new function to separate that column from the rest.

Reopen *preprocessing.py*. Start the new function:

```
def separate_labels(columnNames, rows):
    labelColumnIndex = columnNames.index('Outcome')
```

Now we know where the *Outcome* column is. To separate it, we need to first extract the labels for later use, then erase the column. That looks like this:

```
    ys = rows[:, labelColumnIndex]
```

Remember back to section 3.4. Our first index here is a slice, selecting *every* row of the input data. The second index is just the index we already have, to the data representing the label we want to extract. Numpy sees this as "Make a new array from the index labelColumnIndex of every row of the data." That's exactly what we want!

Next, to get just the data of each sample, and not the labels, we need to delete the label column from the rows. Numpy provides a function, `np.delete`, to provide a copy of an array without a specified index or slice. Here's how we can use that:

```
    xs = np.delete(rows,labelColumnIndex,axis=1)
```

We're almost done, but we're missing one more spot. The name of the label column is still present in the `columnNames`! Python has a different syntax for erasing from lists:

```
    del columnNames[labelColumnIndex]
```

Done. Now we can return all three segments of the data: the corrected column names, the samples, and the labels.

```
    return columnNames, xs, ys
```

The complete data separation function is:

```
def separate_labels(columnNames, rows):
    labelColumnIndex = columnNames.index('Outcome')
    ys = rows[:, labelColumnIndex]
    xs = np.delete(rows,labelColumnIndex,axis=1)
    del columnNames[labelColumnIndex]
    return columnNames, xs, ys
```

Now, from the files you're working in, you can load the data and separate labels with:

```
import preprocessing
columnNames, data = load_data('diabetes.csv')
columnNames, xs, ys = separate_labels(columnNames, data)
```

For loading data that doesn't include labels, you're done at the first line:

```
import preprocessing
columNames, xs = load_data('unknowns.csv')
```

Now you're ready to apply your `xs` and `ys` to the sklearn tools. Take a look at *Cross-validation* for an immediate next step, then look at the classifiers' help pages for functions like `.fit` and `.predict`. Once you've trained a model and created predictions, *Sklearn Metrics* provide numerous ways to interpret and score your classifiers' capabilities.

Good luck!