medford-group / data_analytics_ChE

Code Issues Pull requests Actions Projects Security Insights

master

data_analytics_ChE / 1-numerical_methods / Topic1-Python_Basics.ipynb

Sihoon Choi exercises updated History

0 contributors

Raw Blame

1404 lines (1404 sloc) 229 KB

Table of Contents

Python Basics

This course will utilize the Python programming language. Some of you may be familiar with Python, but others may have only worked with MATLAB. Python is similar to MATLAB, but there are a few key differences and transitioning can take some effort. There is a nice guide available for transitioning from MATLAB to Python (https://www.enthought.com/wp-content/uploads/Enthought-MATLAB-to-Python-White-Paper.pdf) which is strongly recommended for students who have not worked with Python before. The Introduction to Python Programming (https://nbviewer.jupyter.org/github/jrjohansson/scientific-python-lectures/blob/master/Lecture-1-Introduction-to-Python-Programming.ipynb) from Johansson is also a great place to start. We will not spend a lot of lecture time specifically discussing Python, but this lecture will introduce some important ideas and provide examples for using Python in data analytics.

If you are new to Python, the learning curve may feel very steep. However, we will hold additional help sessions, and encourage you to search internet resources like Stack Overflow (https://stackoverflow.com/) and the official Python tutorial (https://docs.python.org/3.6/tutorial/index.html) or work with more experienced students to learn.

Why Python?

- Python is commonly used for data analysis, and is growing rapidly
- Python is a full programming language

Python vs. MATLAB

Advantages:

- Flexibility and portability
- Free and open source
- Huge and supportive community

Disadvantages

- Speed and efficiency
- Hard to install and manage versions
- No integrated IDE (see Spyder, etc.)

Python vs. Compiled (C++, Fortran):

Advantages:

- Fast development time
- Easy to read/write/maintain

Disadvantages:

- Speed/efficiency

Python "packages"

Unlike Matlab, Python is organized into "packages" that contain different types of functionality. There is a "standard library" (https://docs.python.org/3/library/) that is available in any Python installation. This includes some basic features like math, but the functions (or libraries) still have to be imported:

In [1]:
```
import math

print(math.exp(1))
```
```
2.718281828459045
```

Functions from packages can also be directly imported:

In [2]:
```
from math import exp
import math as m

print(m.exp(1))
print(exp(1))
print(e(1))
```
```
2.718281828459045
2.718281828459045
---------------------------------------------------------------------------
----
NameError                                 Traceback (most recent call l
ast)
```

```
<ipython-input-2-70bb7621c96c> in <module>
      4 print(m.exp(1))
      5 print(exp(1))
----> 6 print(e(1))

NameError: name 'e' is not defined
```

There are also some non-standard packages that have been created by users as unofficial add-ons. One of the most widely used is the "numpy" (https://numpy.org/) "numerical python" package, which has a number of numerical and linear algebra functions built in. We will use it extensively in this course, and it will almost always be imported using an "alias" of "np":

```
In [3]: import numpy as np

        A = np.array([[1,2],[3,4]])
        A

Out[3]: array([[1, 2],
               [3, 4]])
```

In numpy, matrices are called "arrays". One major difference between numpy and Matlab is that numpy defaults to **elementwise operations** while Matlab defaults to matrix/vector operations.

```
In [4]: B = np.array([[2,2],[2,2]])

        A*B

Out[4]: array([[2, 4],
               [6, 8]])
```

Matrix/vector operations can be done using the `dot` function, or the `@` symbol:

```
In [5]: C = np.dot(A,B)
        C

Out[5]: array([[ 6,  6],
               [14, 14]])

In [6]: D = A@B
        D

Out[6]: array([[ 6,  6],
               [14, 14]])
```

Indexing arrays is similar to MATLAB with 3 major exceptions:

- square brackets ('[]') are used instead of parentheses
- indexing starts from 0 instead of 1
- negative indexes are allowed (-1 is `end`)

```
In [7]: print(A[0,0])
        X = np.array([[1.,2.,3.],[4, 5]])
        print(X[-2])
```

```
1
[1.0, 2.0, 3.0]
```

The `:` operator can be used to take "slices" of matrices, similar to Matlab with the syntax of:

`start:end`

or

`start:end:step`

In [8]:
```
a_col = A[:,0]
a_col_2 = a_col.reshape(-1, 1)
a_col_2.shape
A_2 = A.reshape(4,1)
print("A matrix: {}".format(A_2))
a_col
```

```
A matrix: [[1]
 [2]
 [3]
 [4]]
```

Out[8]:
```
array([1, 3])
```

There is a very useful [Matlab/Numpy cheat sheet (https://docs.scipy.org/doc/numpy/user/numpy-for-matlab-users.html)](https://docs.scipy.org/doc/numpy/user/numpy-for-matlab-users.html) that can help you get up to speed quickly.

Defining Functions

Function definitions in Python are similar to MATLAB, but there are some syntax differences. Python is much more flexible about how arguments are passed. Functions take 2 types of arguments: positional arguments and keyword arguments. Positional arguments are always required and must be passed in order, while keyword arguments are optional and do not have to be passed in order. We can look at a quick example:

In [9]:
```
def gaussian(x, mean=0, stdev=1.0):
    g = exp(-(x-mean)**2/(2*stdev**2))
    return g

print(gaussian(0.5))
print(gaussian(0.5, 0))
print(gaussian(0.5, stdev=1, mean=0))
print(gaussian(0.5, 0, 1))
```

```
0.8824969025845955
0.8824969025845955
0.8824969025845955
0.8824969025845955
```

Functions are ended by "returning" a variable. If nothing is explicitly returned, the function will automatically return `None`.

```
In [10]: def null_function():
             x = 5 #none of
             y = x**2 #this code
             z = exp(-y) #matters
             #if nothing is returned
             print(x)
             #return x

         new_var = null_function()
         print(new_var)

         5
         None
```

You will not have to write too many functions from scratch in this class, but will often need to modify existing functions so you should be familiar with how they work. If you do learn to write functions it can save you a lot of coding!

Classes, attributes, and methods

Another significant difference between Matlab and Python is that Python is (very) "object oriented". This means that variables in Python are "objects" defined by an associated "class". We will not discuss how to create classes or exploit this feature of Python, but you will need to interact with many different kinds of classes that have been created by others, so it is important to be familiar with the concept.

In Python, each "object" or variable has functions and variables associated with it. A function associated with an object is called a "method", and a variable associated with an object is called an "attribute". You can see the attributes and methods of an object using the `dir` function:

```
In [11]: dir(A)

Out[11]: ['T',
          '__abs__',
          '__add__',
          '__and__',
          '__array__',
          '__array_finalize__',
          '__array_function__',
          '__array_interface__',
          '__array_prepare__',
          '__array_priority__',
          '__array_struct__',
          '__array_ufunc__',
          '__array_wrap__',
          '__bool__',
          '__class__',
          '__complex__',
          '__contains__',
          '__copy__',
          '__deepcopy__',
          '__delattr__',
          '__delitem__',
          '__dir__',
          '__divmod__',
          '__doc__',
          '__eq__',
          '__float__',
          '__floordiv__',
          '__format__',
          '__ge__',
          '__getattribute__',
          '__getitem__',
          '__gt__',
          '__hash__',
          '__iadd__',
          '__iand__',
          '__ifloordiv__',
          '__ilshift__',
          '__imatmul__',
          '__imod__',
          '__imul__',
          '__index__',
          '__init__',
          '__init_subclass__',
          '__int__',
          '__invert__',
          '__ior__',
          '__ipow__',
          '__irshift__',
          '__isub__',
          '__iter__',
          '__itruediv__',
          '__ixor__',
          '__le__',
          '__len__',
          '__lshift__',
          '__lt__',
          '__matmul__',
          '__mod__',
          '__mul__',
          '__ne__',
          '__neg__',
          '__new__',
          '__or__',
          '__pos__',
          '__pow__',
          '__radd__',
          '__rand__',
          '__rdivmod__',
          '__reduce__',
          '__reduce_ex__',
          '__repr__',
          '__rfloordiv__',
          '__rlshift__',
          '__rmatmul__',
          '__rmod__',
          '__rmul__',
          '__ror__',
          '__rpow__',
          '__rrshift__',
          '__rshift__',
          '__rsub__',
          '__rtruediv__',
          '__rxor__',
          '__setattr__',
          '__setitem__',
          '__setstate__',
          '__sizeof__',
          '__str__',
          '__sub__',
          '__subclasshook__',
          '__truediv__',
          '__xor__',
          'all',
          'any',
          'argmax',
          'argmin',
          'argpartition',
          'argsort',
          'astype',
          'base',
          'byteswap',
          'choose',
          'clip',
          'compress',
          'conj',
          'conjugate',
          'copy',
          'ctypes',
          'cumprod',
          'cumsum',
          'data',
          'diagonal',
          'dot',
          'dtype',
          'dump',
          'dumps',
          'fill',
          'flags',
          'flat',
          'flatten',
          'getfield',
          'imag',
          'item',
          'itemset',
          'itemsize',
          'max',
          'mean',
          'min',
          'nbytes',
          'ndim',
          'newbyteorder',
          'nonzero',
          'partition',
          'prod',
          'ptp',
          'put',
          'ravel',
          'real',
          'repeat',
          'reshape',
          'resize',
          'round',
          'searchsorted',
          'setfield',
          'setflags',
          'shape',
          'size',
          'sort',
          'squeeze',
          'std',
          'strides',
          'sum',
          'swapaxes',
          'take',
          'tobytes',
          'tofile',
          'tolist',
          'tostring',
          'trace',
          'transpose',
          'var',
          'view']
```

The methods/attributes beginning/ending with __ are "private", and you generally should not mess with them unless you know Python well. The others are often very useful ways to analyze or work with the variable. The `shape` attribute of a `numpy` array is very useful, since it tells you the dimensions of the matrix:

In [12]:
```
A.shape
```

Out[12]:
```
(2, 2)
```

The `mean` and `std` methods are useful ways to quickly find the mean and standard deviation of a matrix:

In [13]:
```
print(A.mean())
print(A.std())

np.mean(A)
type(A)
```

```
2.5
1.118033988749895
```

Out[13]:
```
numpy.ndarray
```

In [14]:
```
L = [[0, 1,],[2, 3]]
#L@L
```

Objects, functions, and variables often have associated documentation built right in. You can see this using the `help` function, which is another great way to try to make sense of what is happening and find out more about how to use a method:

```
In [15]: help(A.mean)
```

```
Help on built-in function mean:

mean(...) method of numpy.ndarray instance
    a.mean(axis=None, dtype=None, out=None, keepdims=False)

    Returns the average of the array elements along given axis.

    Refer to `numpy.mean` for full documentation.

    See Also
    --------
    numpy.mean : equivalent function
```

```
In [16]: A.mean(axis=1)
```

```
Out[16]: array([1.5, 3.5])
```

Reading in data

This is a data analysis course, so it is common that you will have to read in different types of data. The `pandas` package is particularly useful for this. It has many helpful features that we will discuss throughout the course, but for now we will just show how to read in a .csv file and convert it into a numpy array.

```
In [17]: import pandas as pd

         df = pd.read_csv("data/ethanol_IR.csv")
         df.head(10)
```

Out[17]:

	wavenumber [cm^-1]	absorbance
0	461.563000	0.015
1	466.250502	0.014
2	470.938003	0.014
3	475.625505	0.014
4	480.313007	0.013
5	485.000509	0.012
6	489.688010	0.012
7	494.375512	0.012
8	499.063014	0.012
9	503.750516	0.013

```
In [18]: X = df.values
         X.shape
```

```
Out[18]: (714, 2)
```

```
In [19]: X[:,1]
```

```
Out[19]: array([ 0.015,  0.014,  0.014,  0.014,  0.013,  0.012,  0.012,  0.012,
                 0.012,  0.013,  0.013,  0.013,  0.014,  0.014,  0.014,  0.014,
                 0.013,  0.013,  0.013,  0.013,  0.012,  0.011,  0.008,  0.008,
                 0.007,  0.005,  0.005,  0.004,  0.003,  0.003,  0.003,  0.002,
                 0.001,  0.001,  0.   ,  0.   , -0.001, -0.002, -0.002, -0.003,
                -0.003, -0.003, -0.003, -0.003, -0.004, -0.003,  0.002, -0.002,
                -0.003, -0.003, -0.003, -0.005, -0.006, -0.006, -0.006, -0.006,
                -0.009, -0.006, -0.005, -0.003, -0.003, -0.003, -0.003, -0.002,
                -0.002, -0.002, -0.001,  0.   ,  0.002,  0.003,  0.005,  0.005,
                 0.006,  0.007,  0.007,  0.007,  0.007,  0.006,  0.006,  0.007,
                 0.008,  0.008,  0.011,  0.019,  0.032,  0.055,  0.077,  0.097,
                 0.099,  0.09 ,  0.128,  0.099,  0.131,  0.114,  0.103,  0.087,
                 0.054,  0.034,  0.021,  0.013,  0.008,  0.007,  0.005,  0.003,
                 0.003,  0.003,  0.004,  0.005,  0.007,  0.011,  0.016,  0.021,
                 0.032,  0.047,  0.066,  0.095,  0.138,  0.176,  0.238,  0.271,
                 0.292,  0.458,  0.426,  0.562,  0.648,  0.712,  0.738,  0.759,
                 0.686,  0.848,  0.686,  0.726,  0.66 ,  0.553,  0.469,  0.308,
                 0.277,  0.248,  0.212,  0.151,  0.114,  0.076,  0.058,  0.045,
                 0.036,  0.032,  0.025,  0.021,  0.019,  0.018,  0.019,  0.02 ,
                 0.023,  0.029,  0.034,  0.036,  0.042,  0.048,  0.053,  0.067,
                 0.08 ,  0.11 ,  0.151,  0.174,  0.188,  0.177,  0.151,  0.19 ,
                 0.208,  0.197,  0.186,  0.154,  0.128,  0.097,  0.079,  0.065,
                 0.057,  0.052,  0.048,  0.046,  0.046,  0.046,  0.047,  0.048,
                 0.048,  0.05 ,  0.051,  0.056,  0.062,  0.07 ,  0.084,  0.108,
                 0.143,  0.174,  0.234,  0.279,  0.315,  0.313,  0.289,  0.347,
                 0.231,  0.275,  0.298,  0.253,  0.194,  0.139,  0.108,  0.094,
                 0.09 ,  0.088,  0.087,  0.107,  0.104,  0.069,  0.064,  0.06 ,
                 0.059,  0.058,  0.051,  0.045,  0.04 ,  0.034,  0.029,  0.026,
                 0.02 ,  0.017,  0.014,  0.011,  0.008,  0.007,  0.006,  0.005,
                 0.003,  0.003,  0.005,  0.003,  0.003,  0.003,  0.003,  0.004,
                 0.003,  0.004,  0.005,  0.004,  0.005,  0.005,  0.005,  0.005,
                 0.006,  0.005,  0.005,  0.005,  0.005,  0.005,  0.007,  0.007,
                 0.005,  0.005,  0.005,  0.005,  0.007,  0.007,  0.005,  0.005,
                 0.006,  0.006,  0.006,  0.007,  0.008,  0.007,  0.007,  0.007,
                 0.007,  0.007,  0.007,  0.007,  0.007,  0.007,  0.008,  0.008,
                 0.007,  0.005,  0.005,  0.005,  0.005,  0.005,  0.003,  0.003,
                 0.003,  0.003,  0.002,  0.001,  0.002,  0.002,  0.002,  0.002,
                 0.001,  0.001,  0.001,  0.002,  0.003,  0.003,  0.003,  0.004,
                 0.005,  0.005,  0.007,  0.007,  0.008,  0.01 ,  0.011,  0.012,
                 0.013,  0.013,  0.013,  0.012,  0.011,  0.009,  0.008,  0.006,
                 0.005,  0.004,  0.003,  0.003,  0.002,  0.001,  0.   ,  0.   ,
                 0.   ,  0.   ,  0.   ,  0.   ,  0.   ,  0.   ,  0.   ,  0.   ,
                 0.   ,  0.   , -0.001,  0.   ,  0.   ,  0.   ,  0.   ,  0.   ,
                 0.   ,  0.001,  0.001,  0.002,  0.003,  0.003,  0.004,  0.005,
                 0.005,  0.005,  0.007,  0.007,  0.007,  0.007,  0.006,  0.005,
                 0.004,  0.003,  0.003,  0.003,  0.003,  0.003,  0.003,  0.003,
                 0.003,  0.003,  0.002,  0.002,  0.001,  0.   ,  0.   ,  0.   ,
                 0.   ,  0.   ,  0.   ,  0.001,  0.001,  0.001,  0.002,  0.003,
                 0.004,  0.004,  0.005,  0.005,  0.004,  0.003,  0.003,  0.002,
                 0.001,  0.   ,  0.   ,  0.   ,  0.   ,  0.   ,  0.   ,  0.   ,
                 0.001,  0.002,  0.003,  0.002,  0.   ,  0.   ,  0.   ,  0.   ,
                 0.   ,  0.   ,  0.001,  0.001,  0.001,  0.001,  0.001,  0.002,
                 0.003,  0.003,  0.003,  0.003,  0.003,  0.003,  0.003,  0.002,
                 0.002,  0.002,  0.001,  0.001,  0.001,  0.001,  0.001,  0.001,
                 0.001,  0.001,  0.   ,  0.   ,  0.   ,  0.   ,  0.   ,  0.   ,
                 0.   ,  0.   ,  0.   ,  0.   ,  0.   ,  0.   ,  0.   ,  0.   ,
                 0.   ,  0.   ,  0.   ,  0.   ,  0.   ,  0.   ,  0.001,  0.001,
                 0.001,  0.001,  0.002,  0.003,  0.004,  0.004,  0.005,  0.005,
                 0.005,  0.006,  0.006,  0.007,  0.007,  0.007,  0.008,  0.008,
                 0.008,  0.009,  0.01 ,  0.01 ,  0.011,  0.011,  0.012,  0.013,
                 0.015,  0.017,  0.018,  0.02 ,  0.021,  0.024,  0.027,  0.029,
                 0.03 ,  0.031,  0.033,  0.034,  0.036,  0.037,  0.039,  0.04 ,
                 0.043,  0.047,  0.049,  0.052,  0.055,  0.058,  0.062,  0.066,
                 0.071,  0.077,  0.085,  0.097,  0.111,  0.141,  0.169,  0.183,
                 0.235,  0.265,  0.324,  0.35 ,  0.396,  0.421,  0.45 ,  0.467,
                 0.486,  0.514,  0.51 ,  0.464,  0.444,  0.437,  0.432,  0.432,
                 0.437,  0.442,  0.45 ,  0.475,  0.501,  0.541,  0.553,  0.594,
                 0.611,  0.611,  0.607,  0.612,  0.607,  0.521,  0.471,  0.424,
                 0.331,  0.264,  0.216,  0.161,  0.114,  0.094,  0.054,  0.034,
                 0.021,  0.014,  0.008,  0.007,  0.005,  0.004,  0.003,  0.003,
                 0.002,  0.002,  0.002,  0.001,  0.001,  0.001,  0.001,  0.   ,
                 0.   ,  0.   ,  0.   ,  0.   ,  0.   ,  0.   ,  0.   ,  0.   ,
                 0.   ,  0.   ,  0.   ,  0.   ,  0.   ,  0.001,  0.001,  0.001,
                 0.002,  0.003,  0.003,  0.003,  0.004,  0.004,  0.003,  0.003,
                 0.002,  0.001,  0.001,  0.   ,  0.   , -0.001, -0.001, -0.002,
                -0.002, -0.002, -0.002, -0.002, -0.003, -0.003, -0.003, -0.003,
                -0.003, -0.003, -0.003, -0.003, -0.003, -0.003, -0.003, -0.003,
                -0.003, -0.002, -0.002, -0.002, -0.002, -0.002, -0.002, -0.002,
                -0.001, -0.001,  0.   ,  0.   ,  0.   ,  0.   , -0.001, -0.001,
                -0.002, -0.002, -0.002, -0.001,  0.   ,  0.   ,  0.   ,  0.   ,
                 0.001,  0.001,  0.   ,  0.   ,  0.   ,  0.   ,  0.   , -0.001,
                -0.001, -0.001, -0.001, -0.002, -0.002, -0.002, -0.001, -0.001,
                 0.   ,  0.   ,  0.   ,  0.   ,  0.001,  0.002,  0.004,  0.004,
                 0.004,  0.004,  0.005,  0.005,  0.007,  0.008,  0.01 ,  0.012,
                 0.017,  0.023,  0.032,  0.042,  0.056,  0.072,  0.091,  0.117,
                 0.138,  0.14 ,  0.14 ,  0.158,  0.131,  0.124,  0.125,  0.114,
                 0.102,  0.061,  0.034,  0.021,  0.013,  0.007,  0.004,  0.001,
                 0.   ,  0.   ,  0.   ,  0.   ,  0.   ,  0.   ,  0.   ,  0.   ,
                 0.   ,  0.   ,  0.   ,  0.001,  0.001,  0.001,  0.001,  0.   ,
                 0.   ,  0.   ])
```

Plotting data

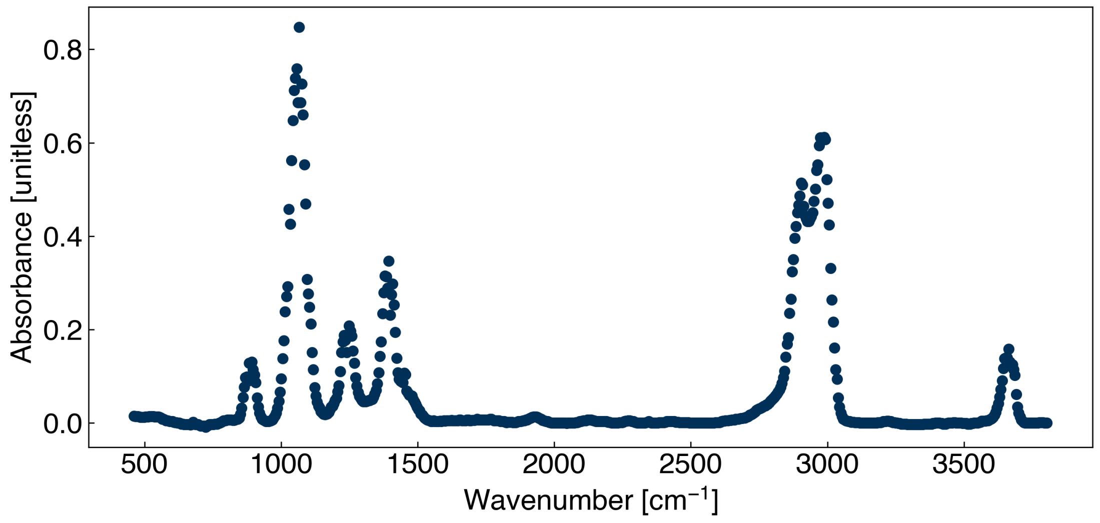
Plotting data in Python is also similar to Matlab, and the main library, `matplotlib` is designed to work similarly. It is rather powerful, but can also be clunky. In particular, plots cannot be easily edited after they are generated, so all modifications must be done using code. There is a gallery of Matplotlib examples (https://matplotlib.org/gallery.html), along with associated source code, that can be very useful if you want to create fancier plots.

Typically we will make very simple plots in this course. An example of the `ethanol_IR.csv` dataset is provided below:

In [20]:
```
%matplotlib inline
#from matplotlib import pylab as plt # This is the long way to do it
import pylab as plt #a much shorter but equivalent import

plt.style.use('../settings/plot_style.mplstyle')
```

```
In [21]: x = X[:,0] #take the first column as independent variable
         y = X[:,1] #take the second column as dependent variable
```

```
In [22]: fig, ax = plt.subplots(figsize=(10, 5)) #create a new figure object (fi
         g) with an associated axis object (ax)
         ax.plot(x,y,marker='o', ls='none')
         ax.set_xlabel('Wavenumber [cm$^{-1}$]')
         ax.set_ylabel('Absorbance [unitless]');
```



Integrated Development Environments (IDE's)

In this course we will use Jupyter Notebooks (like this one) to display code and submit homeworks. However, these can be a difficult way to learn Python if you aren't familiar. It can be much easier to use an IDE like "Spyder" to learn, since it will make Python feel more like Matlab. You can use Spyder (or another IDE) to write code and copy/paste it into Jupyter notebooks to turn it in.

Another useful tip is that Jupyter notebooks can be exported as `.py` files (`File -> Download as -> .py`), which you can then open in Spyder.

Python "gotchas"

Python is great once you get used to it, and is generally very similar to Matlab and other scripting languages. However, there are a number of very tricky things that can cause issues and lots of frustration. I will collect the most insidious "gotchas" that come up in the course here in this notebook throughout the semester. If you have one to add, please let an instructor or teaching assistant know and we can include it.

Tabs and spaces

Python forces you to write structured code by using spaces to determine variable scope. You probably found this out pretty quickly. However, one very frustrating issue that often comes up is that Python *does* distinguish between tabs and spaces when determining spacing. For this reason you should always be consistent and use tabs *or* spaces, but not both.

Jupyter automatically "expands" tabs into spaces, so it is generally not an issue inside of notebooks, but it can cause problems if you are copy/pasting code, or writing in a text editor that isn't configured for Python coding.

See below for an example:

```
In [23]: def hello_world():
             print("Hello World")
             print('...with space indent works.')
         #       print('…with tab indent doesn’t') ## This line was copied in fr
         om a text editor.

         hello_world()
```

```
Hello World
...with space indent works.
```

Numpy array copying

The underlying routines in Numpy are fast because they are written in C++, and Python is just used to access them efficiently. However, this leads to some quirky behavior. In particular, Numpy arrays work on "pointers" by default. This means that *all instances of an array operate on the same space in memory*. This can save a lot of coding and time if you know how it works, but it can also create some pretty insidious bugs through "non-local" modification of numpy arrays.

For example:

```
In [24]: import numpy as np
         A = np.array([[1., 2., 3.],[4., 5., 6.],[7., 8., 9.]])
         print(A)
         b = A[0,:]
         print(b)
         b[0] = 10
         print(b)
         print(A)
```

```
[[1. 2. 3.]
 [4. 5. 6.]
 [7. 8. 9.]]
[1. 2. 3.]
[10.  2.  3.]
[[10.  2.  3.]
 [ 4.  5.  6.]
 [ 7.  8.  9.]]
```

You can avoid this behavior by "copying" arrays when assigning sub-arrays as variables:

```
In [25]: c = A[1,:].copy()
         c[1] = 15
         print(c)
         print(A)
```

```
[ 4. 15.  6.]
[[10.  2.  3.]
```

```
 [ 4.  5.  6.]
 [ 7.  8.  9.]]
```

Integer type casting

Another tricky thing in Python is the "integer" type. By default, any number without a decimal is an integer. In older versions of Python this caused problems with division since the rule was that the result of any operation between two integers must also be an integer, but that has been fixed in Python 3:

In [26]:
```
x = 1
y = 2
print(x/y)
```

```
0.5
```

However, Numpy also uses specific types for arrays, and it follows the rule that assigning a variable to an array cannot change its type:

In [27]:
```
A = np.array([1,2])
print(A)
```

medford-group / data_analytics_ChE

Code Issues Pull requests Actions Projects Security Insights

master

data_analytics_ChE / 1-numerical_methods / Topic2-Linear_Algebra.ipynb

Sihoon Choi exercises updated History

0 contributors

Raw Blame

969 lines (969 sloc) 396 KB

Table of Contents

Linear Algebra

Linear algebra is required for all engineers, but the conceptual aspects are often not taught or have been forgotten, so it is useful to have a refresher on some key concepts.

Matrix-vector multiplication

First, some definitions:

- Dot product or "inner product":

$$\vec{a} \cdot \vec{b} = \sum_i a_i b_i$$

- Matrix/vector multiplication:

$$\bar{\bar{A}}\vec{x} = \sum_j A_{ij} x_j = b_i$$

- Matrix/matrix multiplication:

$$\bar{\bar{A}}\bar{\bar{B}} = \sum_j A_{ij} B_{jk}$$

Exercise: Write a function that uses for loops to multiply a matrix and a vector.

Input:

```
A  = np.array([[0, 1], [2, 3]])
B = np.array([0, 1])

for i in range(A.shape[0]):
    sum = 0
    for j in range(A.shape[1]):
        sum += A[i][j] * B[j]
    print(sum)
```

Output:

```
1
3
```

We will explore matrix-vector multiplication conceptually by constructing a dataset from the "Vandermonde" matrix of polynomials and a weight vector, $\vec{w}$ to construct a dataset of the form:

$y_i = w_0 + w_1 x_i + w_2 x_i^2$

it may be useful to write this with summation notation and compare it to the matrix-vector multiplication definition:

$y_i = \sum_{j=0}^{2} w_j {x_i}^j$

First, we can use numpy to create a vector x_i:

```
In [1]: %matplotlib inline
        import numpy as np
        import pylab as plt
        plt.style.use('../settings/plot_style.mplstyle')

        xi = np.linspace(0,10,11)
        xi
```

```
Out[1]: array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.])
```

Now, we can create a new vector $z_i = x_i^2$:

```
In [2]: zi = xi**2
        zi
```

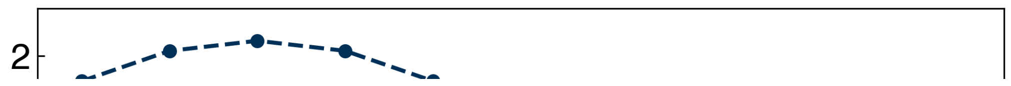
```
Out[2]: array([  0.,   1.,   4.,   9.,  16.,  25.,  36.,  49.,  64.,  81., 10
        0.])
```

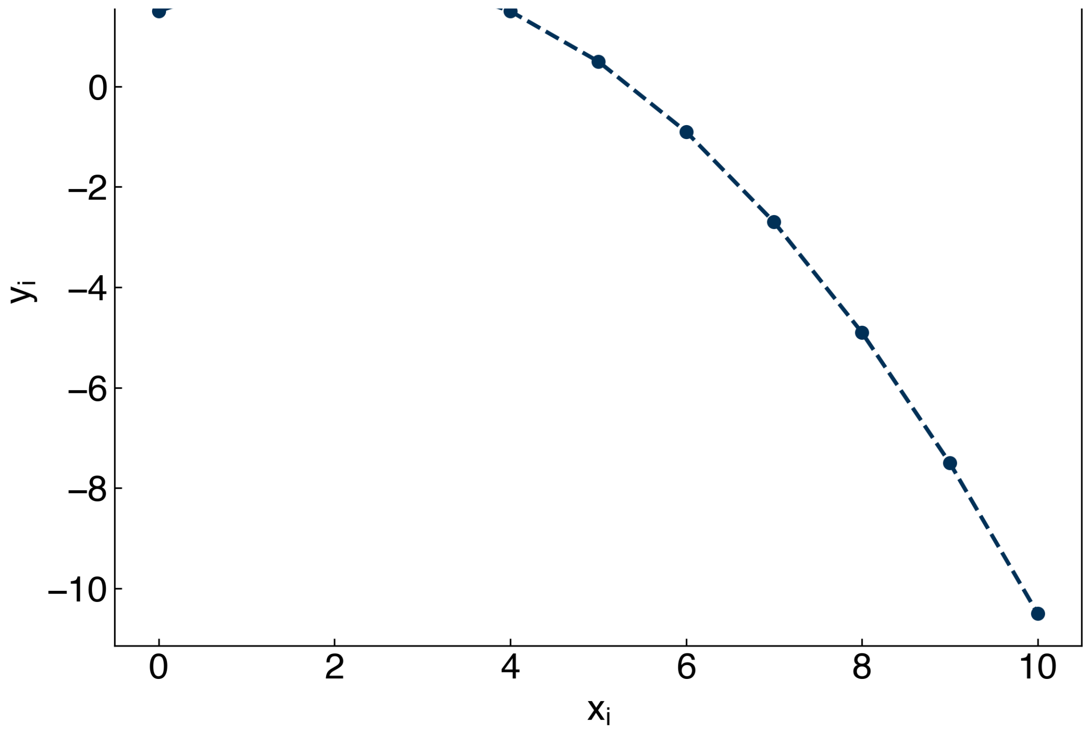
Next, let's create a vector that contains each of the weight parameters, w_j:

```
In [3]: wj = [1.5, 0.8, -0.2]
```

We can now construct y_i manually:

```
In [4]: yi = wj[0] + wj[1]*xi + wj[2]*zi

        fig, ax = plt.subplots()

        ax.plot(xi, yi, '--o')
        ax.set_xlabel('$x_i$')
        ax.set_ylabel('$y_i$');
```





This works, but we can create the same dataset using a Vandermonde matrix, which is a matrix of polynomials defined as:

$X_{ij} = {x_i}^j$

In other words, each column of the matrix consists of a different polynomial.

We can construct this matrix using `numpy`. First, we need to turn x_i into a column vector:

In [5]:
```
print("x_i vector shape: {}".format(xi.shape))
#xi_col = xi.reshape((xi.shape[0], 1)) #<- here we "reshape" the matrix
into a column
xi_col = xi.reshape(-1, 1) #<- this is equivalent, but less clear. It i
s a common shortcut.
print("x_i column shape: {}".format(xi_col.shape))
```

```
x_i vector shape: (11,)
x_i column shape: (11, 1)
```

Now, we can "stack" these vectors together to create the Vandermonde matrix:

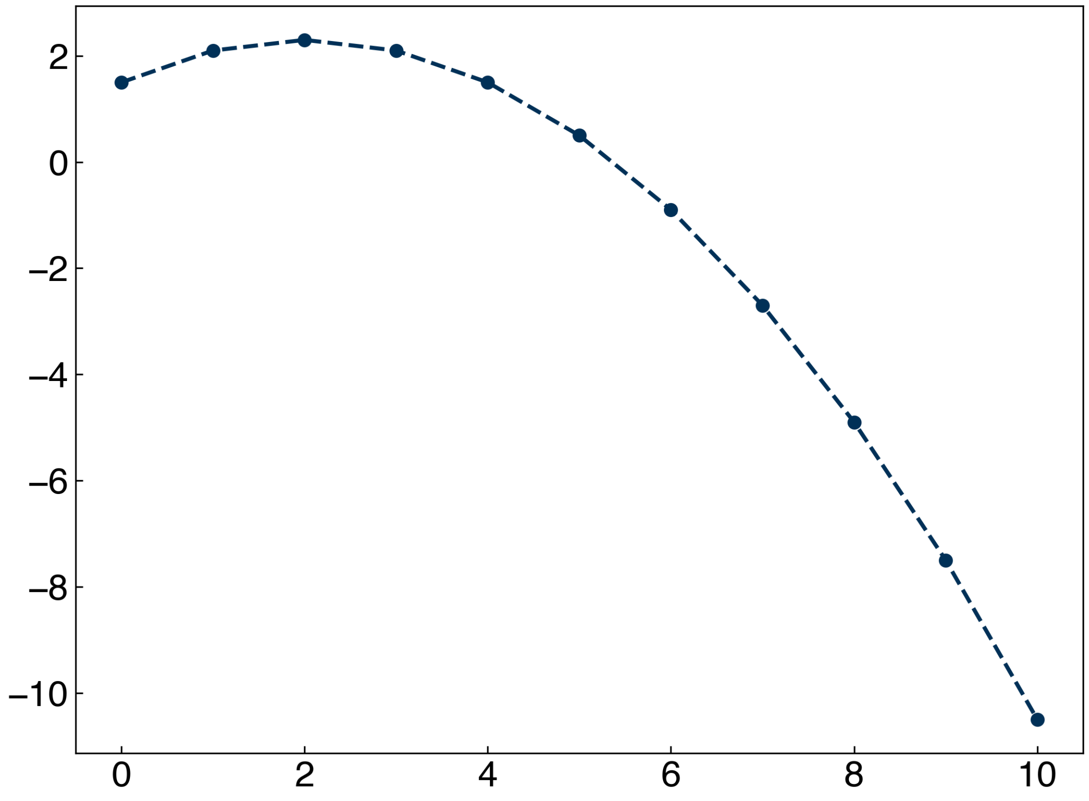
In [6]:
```
xi = xi_col
X_vdm = np.hstack((xi**0, xi**1, xi**2))
X_vdm
```

```
Out[6]: array([[  1.,   0.,   0.],
               [  1.,   1.,   1.],
               [  1.,   2.,   4.],
               [  1.,   3.,   9.],
               [  1.,   4.,  16.],
               [  1.,   5.,  25.],
               [  1.,   6.,  36.],
               [  1.,   7.,  49.],
               [  1.,   8.,  64.],
               [  1.,   9.,  81.],
               [  1.,  10., 100.]])
```

Next we can directly create y_i using matrix-vector multiplication based on the definition of matrix-vector multiplication:

$$\bar{\bar{X}}\vec{w} = \sum_j X_{ij}w_j = \sum_j {x_i}^j w_j = w_0 x_i^0 + w_1 x_i^1 + w_2 x_i^2 + ... w_n x_i^n = w_0 + w_1 x_i + x_2 x_i^2 + ... w_n x_i^n$$

```
In [7]: yi_vdm = X_vdm@wj

        fig, ax = plt.subplots()

        ax.plot(xi, yi_vdm, '--o');
```



We can verify that they are equal:

```
In [8]: yi == yi_vdm

Out[8]: array([ True,  True,  True,  True,  True,  True,  True,  True,  True,
                True,  True])
```

Sometimes it is also useful to use `isclose` instead of `==` since numerical methods are prone to very small errors:

```
In [9]: np.isclose(yi, yi_vdm)

Out[9]: array([ True,  True,  True,  True,  True,  True,  True,  True,  True,
                True,  True])
```

The `.all()` method is a good way to confirm that all values are equal:

```
In [10]: M = np.isclose(yi, yi_vdm)
         M.all()

Out[10]: True
```

Exercise: Create your own Vandermonde matrix

Let's make a 2nd order Vandermonde matrix.

Input:

```
x = np.linspace(0, 5, 6)
vdm = np.hstack((x**0, x, x**2))
print(vdm)
```

Output:

```
np.array([[1, 0, 0],
                [1, 1, 1],
                [1, 2, 4],
                [1, 3, 9],
                [1, 4, 16],
                [1, 5, 25]])
```

Norms, inner products, and orthogonality

Vectors can be described by various "norms" that capture their size and distance. The most common is the L_2 norm, also called the "Euclidean distance" which is defined as:

$$||\vec{x}||_2 = \sqrt{\sum_i x_i^2}$$

This can also be computed by taking the square root of the vector-vector "inner product" of a vector with itself:

$$||\vec{x}||_2 = \sqrt{\vec{x}^T \vec{x}}$$

A vector is called "normal" if it's norm is 1. We can always "normalize" a vector by dividing it by its norm:

```
In [11]: col_0 = X_vdm[:,0]
         norm_col_0 = np.linalg.norm(col_0,2)
         col_0_normed = col_0/norm_col_0
         print('Column norm: {}'.format(norm_col_0))
         print('Normed column norm: {}'.format(np.linalg.norm(col_0_normed,2)))

         Column norm: 3.3166247903554
         Normed column norm: 1.0
```

We will also occasionally use the L_1 norm, defined as:

$\|\vec{x}\|_1 = \sum_i |x_i|$

These norms, and others are discussed in detail in the online notes of Machine Learning Refined (https://github.com/jermwatt/machine_learning_refined/blob/gh-pages/notes/16_Linear_algebra/16_5_Norms.ipynb) and Lecture 3 of Trefethen & Bau.

It is also useful to remember that the inner product between two different vectors is equal to the product of their magnitudes and the cosine of the angle between them:

$\vec{x}^T\vec{y} = \|\vec{x}\|_2\|\vec{y}\|_2 cos(\theta)$

This is described in detail in the online Machine Learning Refined (https://github.com/jermwatt/machine_learning_refined/blob/gh-pages/notes/16_Linear_algebra/16_2_Vectors.ipynb) notes.

We can use this to compute the angle between two normed vectors:

```
In [12]: col_1 = X_vdm[:,1]
         norm_col_1 = np.linalg.norm(col_1,2)
         col_1_normed = col_1/norm_col_1

         cos_theta = np.dot(col_1_normed, col_0_normed)
         theta = np.degrees(np.arccos(cos_theta))
         theta
```

```
Out[12]: 32.31153323742384
```

Vectors are defined as "orthogonal" if their inner product is zero. We can check that this is consistent with our typical definition of "orthogonal":

```
In [13]: theta = np.degrees(np.arccos(0))
         theta
```

```
Out[13]: 90.0
```

In other words, orthogonal vectors are at right angles to each other, or have no projection onto each other.

One key concept that comes in handy is the ability to find the orthogonal components of an arbitrary set of vectors. We can do this by subtracting off the projection of one vector onto another:

```
In [14]: col_1_ortho = col_1_normed - np.dot(col_0_normed, col_1_normed)*col_0_n
         ormed

         np.dot(col_1_ortho, col_0_normed)
```

```
Out[14]: -6.938893903907228e-17
```

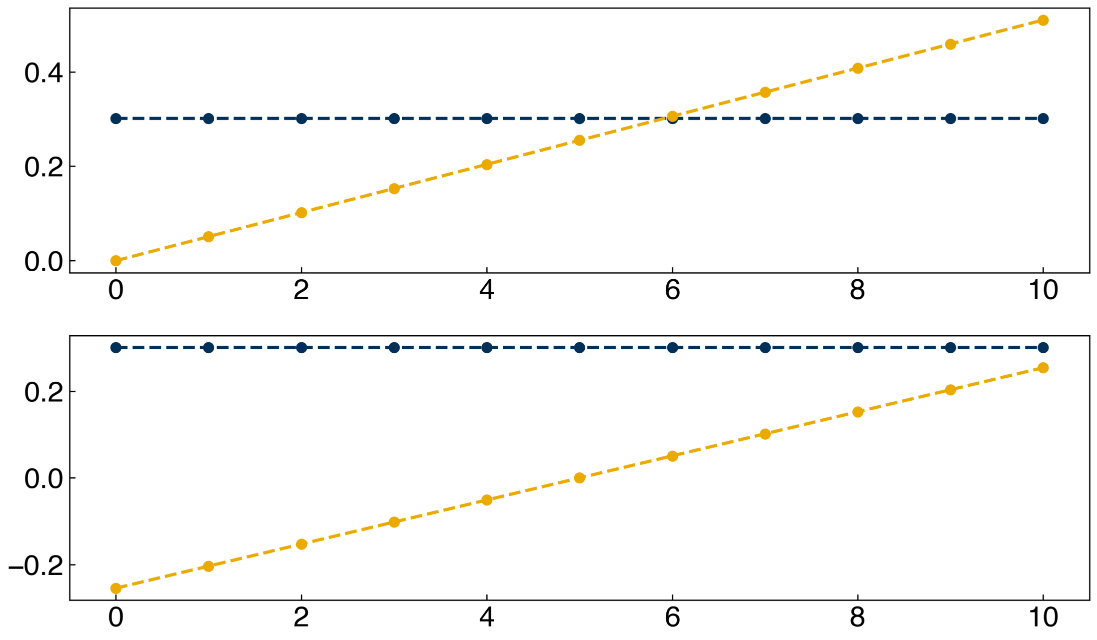
While this isn't technically zero, it is very close:

```
In [15]: np.isclose(np.dot(col_1_ortho, col_0_normed), 0)

Out[15]: True
```

There is a discussion and proof of why this works for orthnormal vectors in Lecture 2 of Trefethen & Bau.

Let's take a look at the original vectors as compared to the orthonormal ones:

```
In [16]: fig, axes = plt.subplots(2,1, figsize=(10,6))

         axes[0].plot(xi, col_0_normed, '--o')
         axes[0].plot(xi, col_1_normed, '--o')

         axes[1].plot(xi, col_0_normed, '--o')
         axes[1].plot(xi, col_1_ortho, '--o');
```



Note that these vectors don't appear to be at "right angles" to each other. Remember that these are not 2-dimensional vectors, but rather 11-dimensional vectors. Our intuition only goes so far when working in high-dimensional spaces!

Exercise: Create an orthonormal version of the Vandermonde matrix

Gram-Schmidt process for the first two columns

Input:

```
x = np.linspace(0, 5, 6)
vdm = np.hstack((x**0, x**1, x**2))

# Orthonormalizing the 1st column
ortho_1 = vdm[:, 0] / np.linalg.norm(vdm[:, 0], order = 2)
```

```
    # Orthonormalizing the 2nd column
    ortho_2 = vdm[:, 1] - np.dot(vdm[:, 0], vdm[:, 1]) / np.dot(vdm[:, 1], vdm
    [:, 1]) * vdm[:, 1]
```

Rank, Inverses, and Linear Systems

The concept of the "rank" of a matrix is important. The formal definition of rank is the "number of linearly independent columns/rows". For an $m \times n$ matrix, the rank is always less than or equal to the minimum of m and n:

rank $\leq min(m, n)$

It is sometimes convenient to think of rank in terms of a linear system of equations defined by:

$\bar{\bar{A}}\vec{x} = \vec{b}$

where $\bar{\bar{A}}$ is an $m \times n$ matrix that defines equations in terms of unknown variables defined by $\vec{x}$. If $m = n$ then $\bar{\bar{A}}$ is square and the number of equations is equal to the number of unknowns. As long as there are no redundant equations, then the rank of $\bar{\bar{A}}$ is equal to m and n. However, if there are redundant equations then the rank is equal to the number of non-redundant equations and the system is **underconstrained**. On the other hand, if $m>n$ then there are more equations than unknowns, and the system is **overconstrained** (assuming there are no redundant equations).

A matrix is **invertible** if and only if it is a square, full-rank matrix. This is equivalent to saying that a system of equations can only be solved ($\vec{x} = \bar{\bar{A}}^{-1}\vec{b}$) if the number of equations is equal to the number of unknowns (square matrix) and no equations are redundant (full-rank).

Let's look at an example based on the Vandermonde matrix and our dataset from before. We will use 3 points to extract the weights, w_j:

In [17]:
```
A = X_vdm[1:4, :]
b = yi[1:4]

print('Shape of A: {}'.format(A.shape))
print('Rank of A: {}'.format(np.linalg.matrix_rank(A)))

A_inv = np.linalg.inv(A)
w = A_inv@b

print('Weights: {}'.format(w))
```

```
Shape of A: (3, 3)
Rank of A: 3
Weights: [ 1.5  0.8 -0.2]
```

Exercise: See how the rank changes if redundant equations are selected.

```
A = np.array([[0, 1], [2, 3]])
print(np.linalg.matrix_rank(A))

# Let's add a redundant row here
B = np.array([[0, 1], [2, 3], [2, 4]])
print(np.linalg.matrix_rank(B))
```

Output:

```
2
2
```

In practice, it is very inefficient to solve systems of equations with matrix inverses. You should be familiar with Gaussian Elimination (http://mathworld.wolfram.com/GaussianElimination.html) from your linear algebra course. There are many other ways to solve linear systems, such as the QR factorization or using eigenvalues. Unfortunately we don't have time to cover these methods in this course, and will simply solve systems using the `solve` function from `numpy`:

In [18]:
```
w_solve = np.linalg.solve(A,b)
np.isclose(w_solve, w).all()
```

Out[18]:
```
True
```

In this course, solving systems of equations is easy as long as (1) you can write the system in the form $\bar{\bar{A}}\vec{x} = \vec{b}$, and (2) the matrix $\bar{\bar{A}}$ is invertible (i.e. the system can be solved).

Eigen and Singular Value Decompositions

The eigenvalue problem for a matrix $\bar{\bar{A}}$:

$$\bar{\bar{A}} v_n = \lambda_n v_n$$

where v_n is the nth eigenvector and λ_n is the nth eigenvalue.

To calculate eigenvalues of a matrix, use the `eigvals` function, and for calculating both eigenvalues and eigenvectors, use the function `eig`:

In [19]:
```
from numpy.linalg import eigvals, eig

print('Eigenvalues of A: {}'.format(eigvals(A)))

vals, vecs = eig(A)
print('Eigenvectors of A: {}'.format(vecs))

np.isclose(vals, eigvals(A))
```

```
Eigenvalues of A: [10.60311024  1.24543789  0.15145187]
Eigenvectors of A: [[-0.13772903 -0.81480675  0.65820453]
 [-0.43070617 -0.49754289 -0.7324674 ]
 [-0.89192091  0.29755844  0.17394918]]
Out[19]: array([ True,  True,  True])
```

The eigenvectors of a **symmetric** matrix will always be orthonormal:

```
In [20]: A_sym = (A.T + A)/2. #make A symmetric
         vals, vecs = eig(A_sym)

         vec0 = vecs[:,0]
         vec1 = vecs[:,1]
         vec2 = vecs[:,2]

         np.isclose(vec0@vec1,0)
Out[20]: True
```

Eigendecomposition is only possible for a square matrix. However, there is a similar concept called a "singular value decomposition", or SVD, that will work for any matrix:

$$A = \hat{U}\hat{\Sigma}V^T$$

```
In [21]: from numpy.linalg import svd

         vecsL, vals, vecsR = svd(A_sym)
         np.isclose(vecsR[2,:], vecs[:,2])
Out[21]: array([ True,  True,  True])
```

For a square matrix, the SVD is equivalent to the eigendecomposition, although the order of the vectors will not always be the same. In general, eigenvalues and singular values are ordered from largest to smallest, but this is not guaranteed by the algorithms that compute them.

The advantage of the SVD is that it will also work for non-square matrices:

```
In [22]: vecsL, vals, vecsR = svd(X_vdm)

         print('Original matrix shape: {}'.format(X_vdm.shape))
         print('Left singular vectors shape: {}'.format(vecsL.shape))
         print('Right singular vectors shape: {}'.format(vecsR.shape))
         print('Singular Values: {}'.format(vals))

         Original matrix shape: (11, 3)
         Left singular vectors shape: (11, 11)
         Right singular vectors shape: (3, 3)
         Singular Values: [160.3135455    5.20337928   1.22146387]
```

medford-group / data_analytics_ChE

Code Issues Pull requests Actions Projects Security Insights

master

data_analytics_ChE / 1-numerical_methods / Topic3-Linear_Regression.ipynb

Sihoon Choi exercises updated History

0 contributors

Download

2.19 MB

Table of Contents

Linear Regression

Simple linear regression

Linear regression is a great starting point for understanding how linear algebra and optimization are used together for data analytics. We will start with simple linear regression, which you should be familiar with.

The form of a simple linear regression model is given as:

$y = mx + b + \epsilon$

where the y is the independent value, x is the dependent value, m is the slope of the line, b is the intercept, and ϵ is the "error" between the model and the actual data. This can also be written with indices on the data:

$y_i = mx_i + b + \epsilon_i$

where i refers to the index of the data point (e.g. the first, second, third, ... data point). We can also think of these quantities as vectors:

$\vec{y} = m\vec{x} + b + \vec{\epsilon}$

To make things consistent with prior lectures, we can re-write this as:

$y_i = w_0 x_i^0 + w_1 x_i^1 + \epsilon_i$

where $w_0 = b$ and $w_1 = m$. Now we can re-write this as a matrix-vector product:

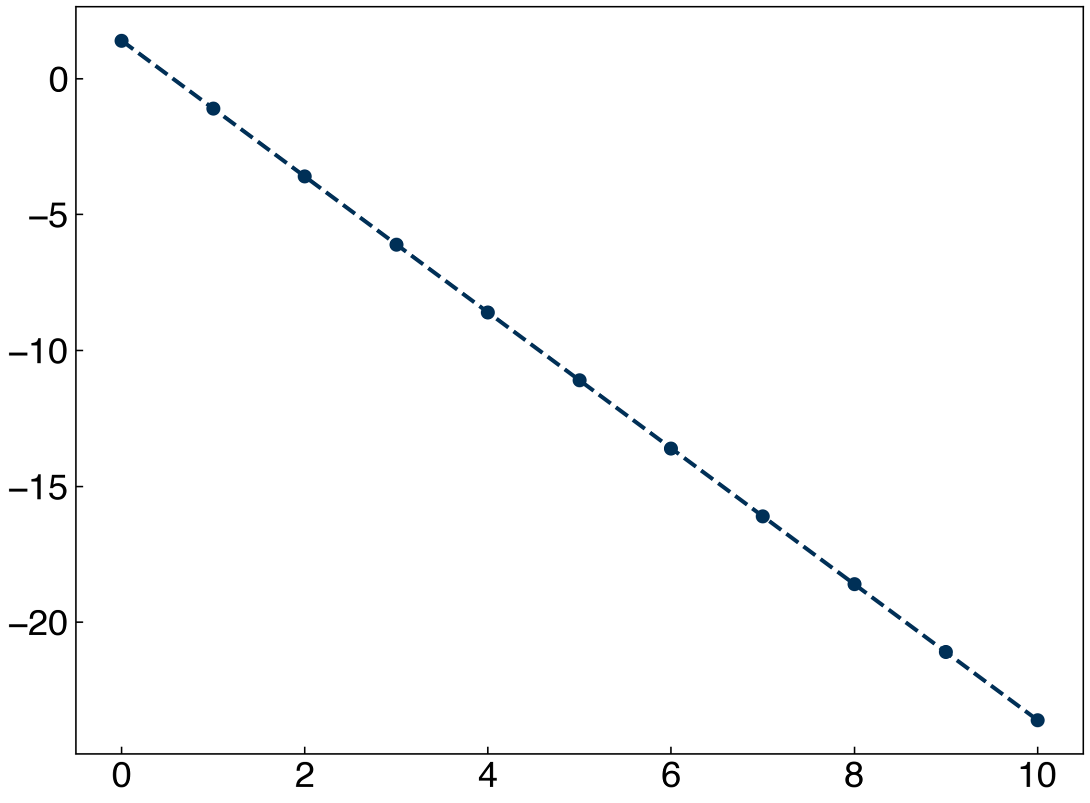
$y_i = \sum_{j=0}^{1} w_j {x_i}^j + \epsilon_i$

If you recall the Vandermonde matrix, this can be written as:

$\vec{y} = \bar{\bar{X}}\vec{w} + \vec{\epsilon}$

where $\bar{\bar{X}}$ is the first-order Vandermonde matrix. We can create a dataset that satisfies this model using `numpy`:

In [1]:
```
%matplotlib inline
import numpy as np
import pylab as plt
plt.style.use('../settings/plot_style.mplstyle')

x = np.linspace(0,10,11)
x = x.reshape(-1, 1) #make x into a column vector
w = [1.4, -2.5]

X = np.hstack((x**0, x))
y = X@w

fig, ax = plt.subplots()
ax.plot(x, y, '--o');
```

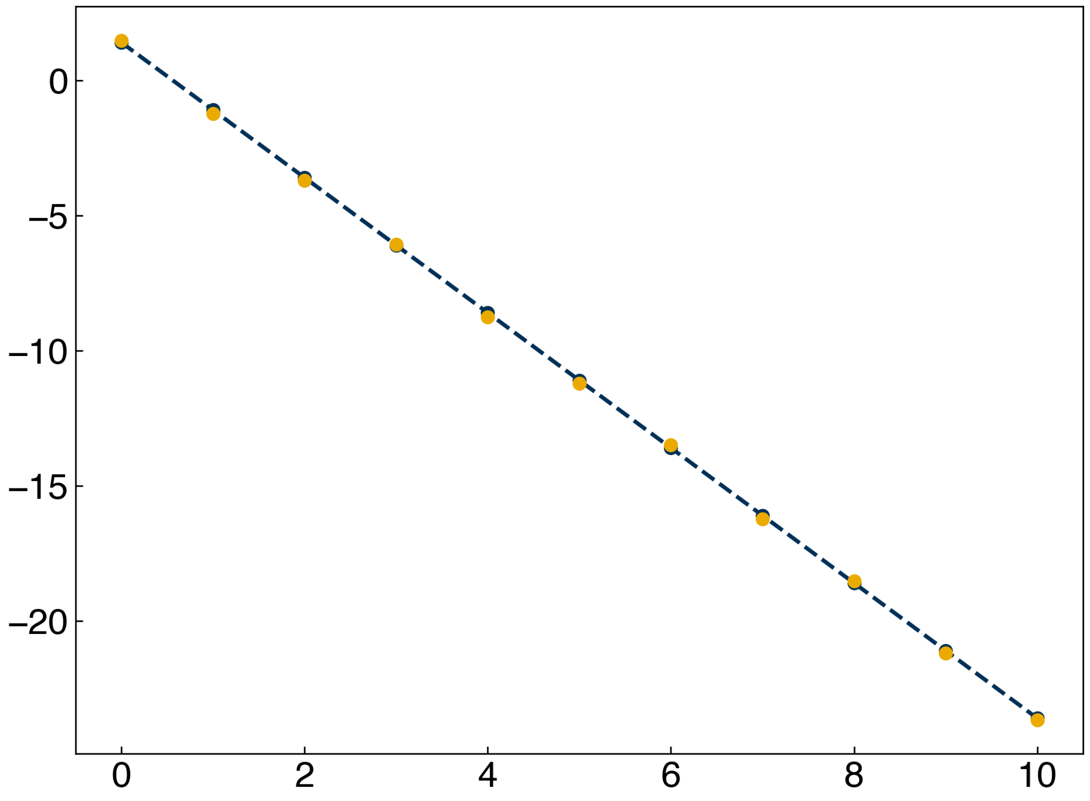


We are still missing the ϵ term. This is the error, and in linear regression we assume that the error follows a normal distribution. Hopefully you remember normal distributions from your stats class. We can generate a vector of normally-distributed noise using `numpy` and add it to $\vec{y}$:

In [2]:
```
from numpy.random import normal
#help(normal)
epsilon = normal(0, 0.1, len(x))
```

Discussion: Why does the error term have a mean ("loc") of 0?

```
In [3]: fig, ax = plt.subplots()
        ax.plot(x, y, '--o')
        y = y + epsilon
        ax.plot(x, y, 'o');
```



The goal of linear regression is to use the data, y_i to recover the "best fit" line. In this case, we know the answer since we generated the data. However, we can also try to recover the line based only on the noisy data.

There are multiple ways to derive linear regression, but here we will derive it by minimizing the sum of squared errors. This is the origin of the name "least squares": we want to find the line that gives the lowest squared errors. First, we will set up a "cost function" that quantifies the squared errors:

$g = \sum_j \epsilon_j^2$

Next, we can recall the definition of an inner product to see that $\sum_j \epsilon_j^2 = \vec{\epsilon}^T \vec{\epsilon}$, so:

$g = \vec{\epsilon}^T \vec{\epsilon}$.

Next, we can re-arrange our expression for the model to solve for $\vec{\epsilon}$:

$\vec{y} = \bar{\bar{X}}\vec{w} + \vec{\epsilon} \Rightarrow \vec{\epsilon} = \vec{y} - \bar{\bar{X}}\vec{w}$

Next we substitute this into the loss function

$g = \vec{\epsilon}^T \cdot \vec{\epsilon} = (\vec{y} - \bar{\bar{X}}\vec{w})^T(\vec{y} - \bar{\bar{X}}\vec{w})$

Recalling matrix transpose rules:

$$(\vec{y} - \bar{\bar{X}}\vec{w})^T(\vec{y} - \bar{\bar{X}}\vec{w}) = (\vec{y}^T - \vec{w}^T\bar{\bar{X}}^T)(\vec{y} - \bar{\bar{X}}\vec{w})$$

and multiplying:

$$\vec{w}^T\bar{\bar{X}}^T\bar{\bar{X}}\vec{w} - \vec{y}^T\bar{\bar{X}}\vec{w} - \vec{w}^T\bar{\bar{X}}^T\vec{y} + \vec{y}^T\vec{y}$$

the middle two terms are both dot products of $\vec{y}^T(\bar{\bar{X}}\vec{w})$ or $(\bar{\bar{X}}\vec{w})^T\vec{y}$, which are transposes of each other, and scalar quantities. The transpose of a scalar is equal to the same scalar, so these terms are equal and can be combined giving:

$$g = \vec{\epsilon}^T \cdot \vec{\epsilon} = \vec{w}^T\bar{\bar{X}}^T\bar{\bar{X}}\vec{w} - 2\vec{y}^T\bar{\bar{X}}\vec{w} + \vec{y}^T\vec{y}$$

Discussion: What is g a function of? Are $\bar{\bar{X}}$ and $\vec{y}$ variables?

We now have the sum of squared errors quantified as a function of the weights, w:

$$g(\vec{w}) = \vec{w}^T\bar{\bar{X}}^T\bar{\bar{X}}\vec{w} - 2\vec{y}^T\bar{\bar{X}}\vec{w} + \vec{y}^T\vec{y}$$

Now we can recall the definition of minimia from calculus: the derivative of a function at a minimum (or maximum) is 0. This implies that we need to take the derivative of the loss function with respect to the parameters ($\vec{w}$) and set it equal to zero:

$$\frac{\partial g}{\partial \vec{w}} = 0$$

Taking derivatives with respect to vectors can be tricky, but the following two identities are useful (and will be provided for exams):

$$\frac{\partial(\bar{\bar{A}}\vec{x})}{\partial \vec{x}} = \bar{\bar{A}}^T$$

$$\frac{\partial(\vec{x}^T\bar{\bar{A}}\vec{x})}{\partial \vec{x}} = (\bar{\bar{A}}^T + \bar{\bar{A}})\vec{x}$$

Using these identities you should be able to show that:

$$\frac{\partial g}{\partial \vec{w}} = 2\bar{\bar{X}}^T\bar{\bar{X}}\vec{w} - 2\bar{\bar{X}}^T\vec{y}$$

Setting equal to zero and re-arranging gives:

$$\bar{\bar{X}}^T\bar{\bar{X}}\vec{w} = \bar{\bar{X}}^T\vec{y}$$

Now we can notice that $\bar{\bar{X}}^T\bar{\bar{X}}$ is a matrix, which we can call $\bar{\bar{A}}$, and $\bar{\bar{X}}^T\vec{y}$ is a vector, which we can call $\vec{b}$. If we let $\vec{w} = \vec{x}$ then we can see that this a system of linear equations:

$$\bar{\bar{A}}\vec{x} = \vec{b}$$

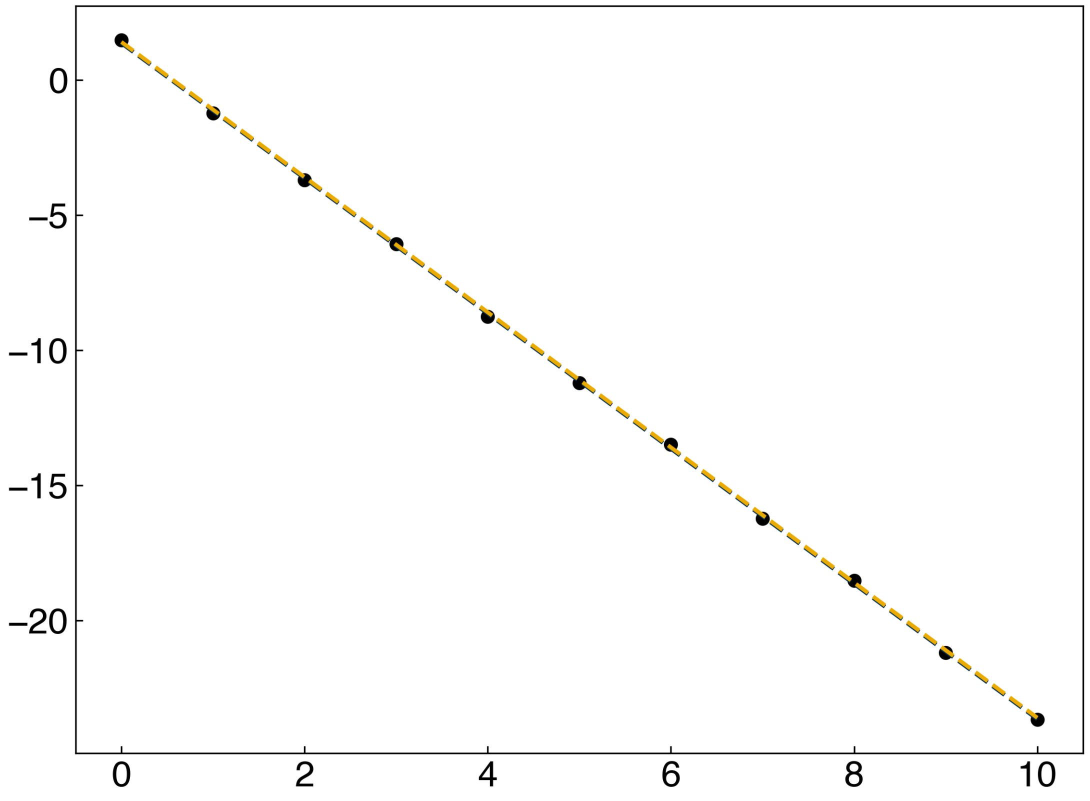
Let's set this up in Python for our toy problem:

```
In [4]: A = X.T@X
        b = X.T@y
        w_lsr = np.linalg.solve(A,b)
        w_lsr = np.linalg.solve(A,b)
        print('Weights from least-squares regression: {}'.format(w_lsr))
        print('Original weights to generate data: {}'.format(w))
```

```
Weights from least-squares regression: [ 1.35659694 -2.50059412]
Original weights to generate data: [1.4, -2.5]
```

We see that the results are not identical, but are close. We can also check the quality of the best-fit line visually:

```
In [5]: yhat = X@w_lsr

        fig, ax = plt.subplots()
        ax.plot(x, y, 'ok')
        ax.plot(x, yhat, '--')
        ax.plot(x, X@w, '--');
```



We can see that the best fit line actually "fits" the data better than the original weights! We will explore more strategies for quantifying the fit in the "regression" lecture.

Polynomial Regression

If you have seen simple linear regression before, the derivation above probably seemed far more complex than what you have seen in the past. However, the advantage is that it is also much more general, as we will see when moving to polynomial regression.

In polynomial regression, we expand the model to be of the form:

$y_i = w_0 + w_1 x_i + w_2 x_i^2 + w_3 x_i^3 ... + \epsilon_i$

As before, we can write this in summation notation:

$$y_i = \sum_{j=0}^{m} w_j {x_i}^j + \epsilon_i$$

where m is the order of the highest polynomial. Recalling the definition of the Vandermonde matrix, we see that this can also be written as:

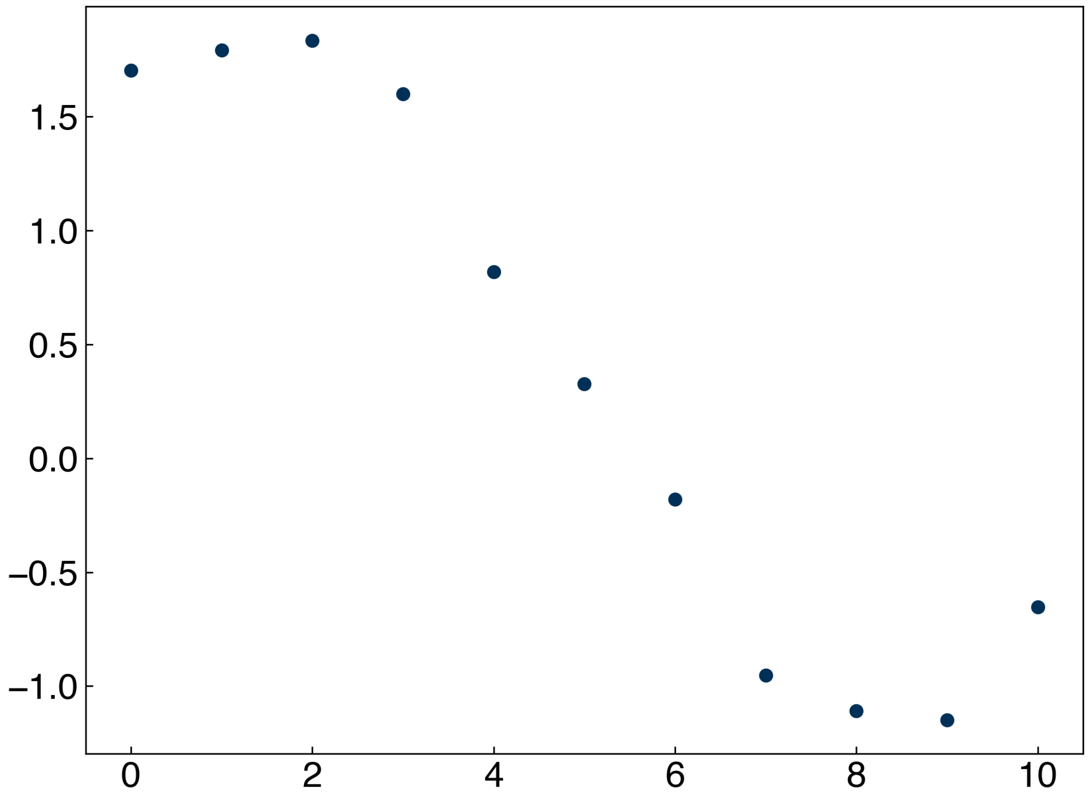
$$\vec{y} = \bar{\bar{X}}\vec{w} + \vec{\epsilon}$$

which is identical to the form that we used for linear regression. The only difference is that now the matrix X has $m+1$ columns instead of 2 columns. This means that we can use the same solution from linear regression for polynomial regression!

$$\bar{\bar{X}}^T\bar{\bar{X}}\vec{w} = \bar{\bar{X}}^T\vec{y}$$

Let's see an example. First, we can generate some data:

```
In [6]: x = np.linspace(0,10,11)
        x = x.reshape(-1, 1) #make x into a column vector
        w = [1.4, 0.8, -0.3, 0.02]

        X = np.hstack((x**0, x, x**2, x**3))
        epsilon = normal(0, 0.2, len(x))
        y = X@w + epsilon

        fig, ax = plt.subplots()
        ax.plot(x,y,'o');
```

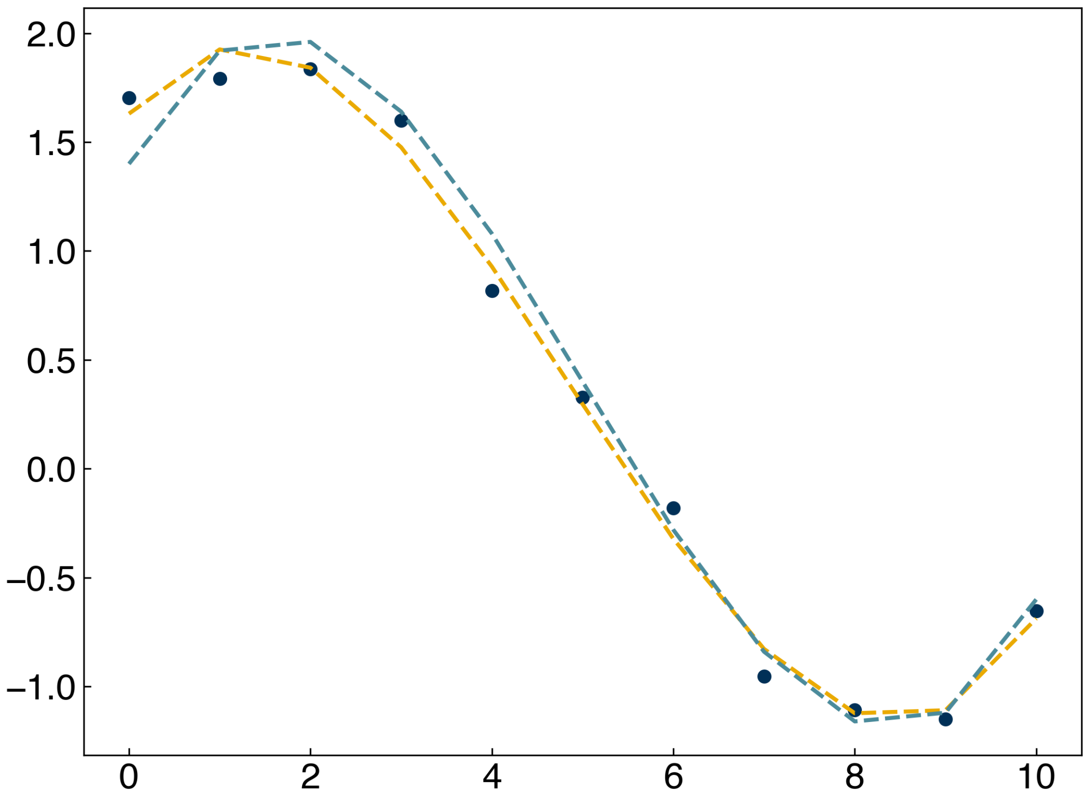


Now we can try to recover the weights, w using the same math as before:

```
In [7]: A = X.T@X
        b = X.T@y
        w_lsr = np.linalg.solve(A,b)
        print('Weights from least-squares regression: {}'.format(w_lsr))
        print('Original weights to generate data: {}'.format(w))
```

```
Weights from least-squares regression: [ 1.63048437  0.5175107  -0.2387
8312  0.01638495]
Original weights to generate data: [1.4, 0.8, -0.3, 0.02]
```

```
In [8]: yhat = X@w_lsr

        fig, ax = plt.subplots()
        ax.plot(x, y, 'o')
        ax.plot(x, yhat, '--')
        ax.plot(x, X@w, '--');
```



In this case we cheated a bit, since we knew that the data was generated from a third-order polynomial. We can also find fits based on lower (or higher) orders of polynomials by modifying the order of the Vandermonde matrix used in the least-squares equation. First, we can make a function that creates a Vandermonde matrix of any order:

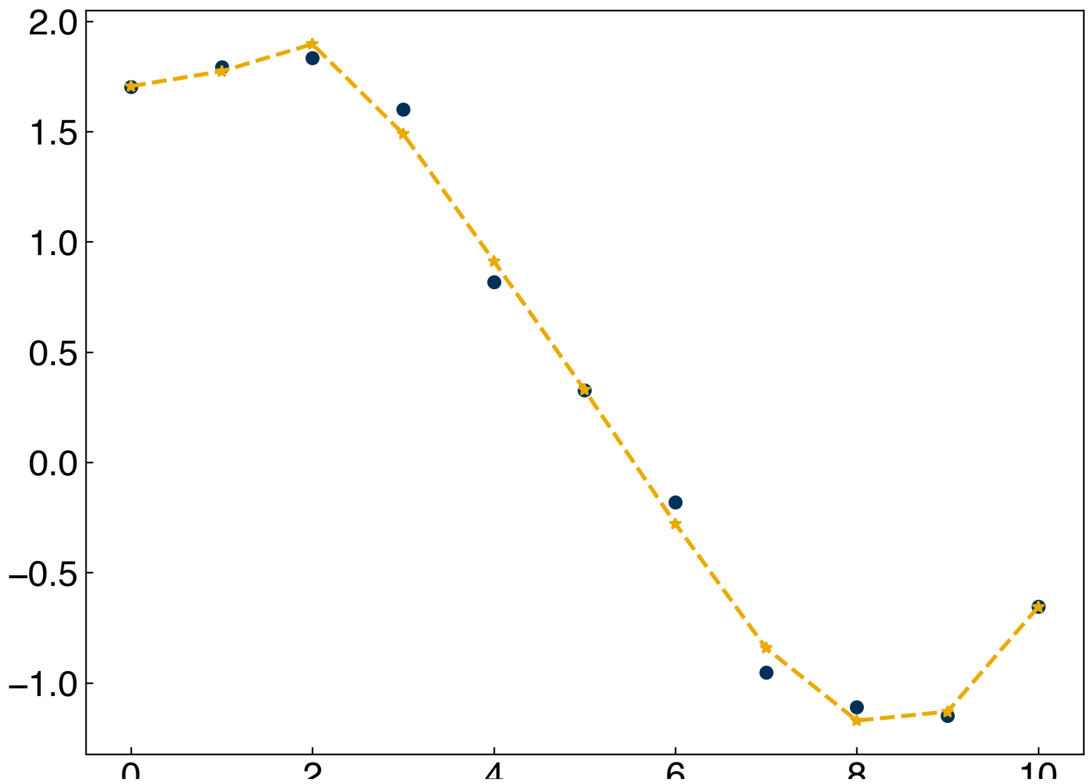
```
In [9]: def vandermonde(x, order):
            cols = []
            for i in range(order):
                cols.append(x**i)
            return np.hstack(cols)

        X_vdm = vandermonde(x,4)
        X_vdm
```

```
Out[9]: array([[   1.,    0.,    0.,    0.],
       [   1.,    1.,    1.,    1.],
       [   1.,    2.,    4.,    8.],
       [   1.,    3.,    9.,   27.],
       [   1.,    4.,   16.,   64.],
       [   1.,    5.,   25.,  125.],
       [   1.,    6.,   36.,  216.],
       [   1.,    7.,   49.,  343.],
       [   1.,    8.,   64.,  512.],
       [   1.,    9.,   81.,  729.],
       [   1.,   10.,  100., 1000.]])
```

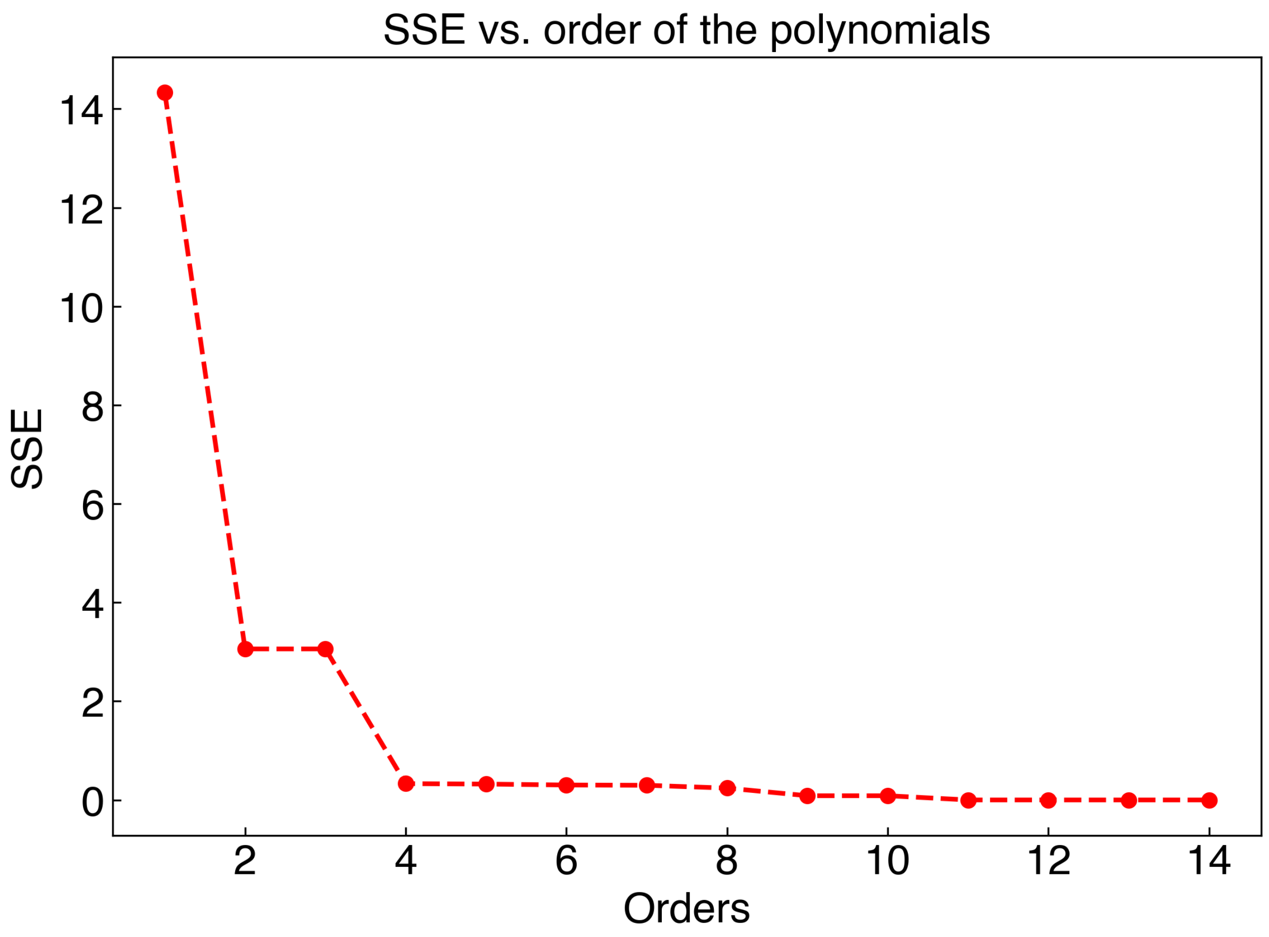
Now we can repeat the fitting procedure with different orders:

In [10]:
```
order = 9
X_vdm = vandermonde(x, order)
A = X_vdm.T@X_vdm
b = X_vdm.T@y
w_lsr = np.linalg.solve(A,b)

print('Weights: {}'.format(w_lsr))

yhat = X_vdm@w_lsr
SSE = sum((y - yhat)**2)
print('Sum of Squared Errors (g): {}'.format(SSE))

fig, ax = plt.subplots()
ax.plot(x, y, 'o')
ax.plot(x, yhat, '--*');
```

```
Weights: [ 1.70601223e+00 -9.94434937e-01  2.12268179e+00 -1.45836510e+
00
  4.75202408e-01 -8.61230848e-02  8.86556647e-03 -4.84176272e-04
  1.09016031e-05]
Sum of Squared Errors (g): 0.05129994706901623
```



Exercise: Compute the sum of squared errors (g) as a function of the order of the polynomial used to fit the data.



We see that as the order of the polynomial increases, the sum of squared errors decreases. However, we are only checking the behavior of our model at the points we use to train it. We can also use the model to interpolate between points or extrapolate to new points by creating a new Vandermonde matrix with more rows. Increasing the resolution adds rows within the original range, and results in interpolation, while increasing the range will result in extrapolation. Let's try both:

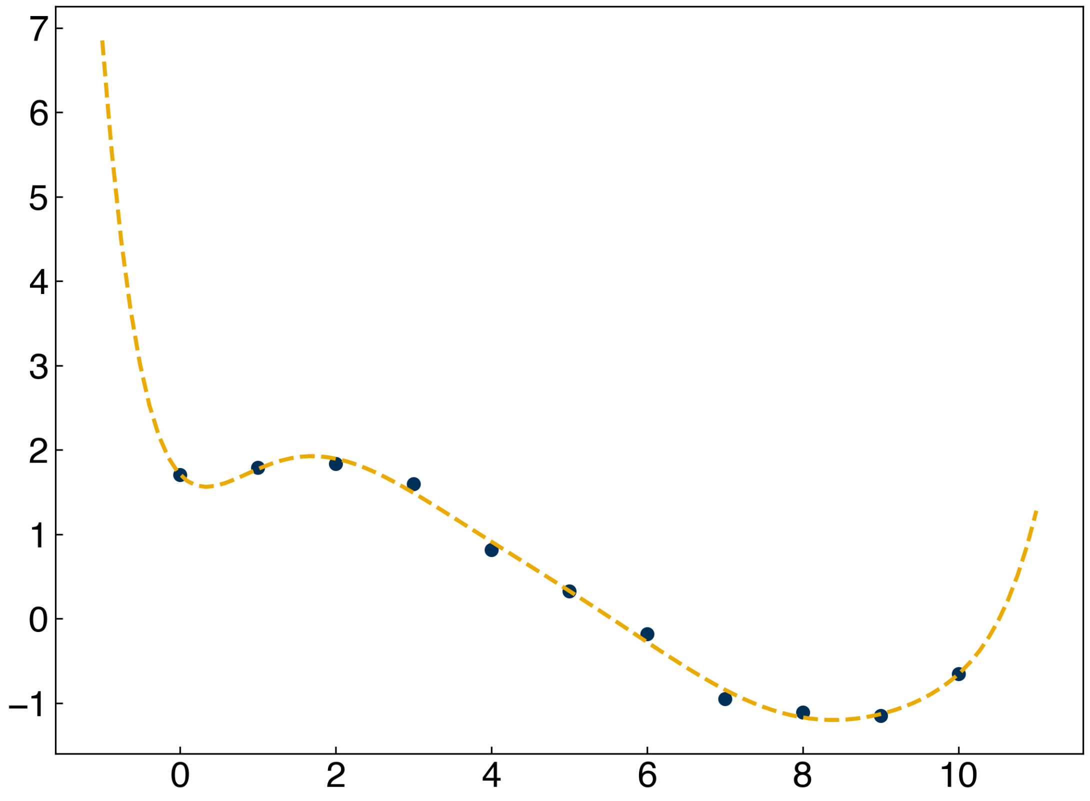
```
In [11]: x_new = np.linspace(-1, 11, 100)
         x_new = x_new.reshape(-1, 1) #create column vector
         x_new.shape

Out[11]: (100, 1)
```

```
In [12]: X_vdm_new = vandermonde(x_new, order)
         X_vdm_new.shape

Out[12]: (100, 9)
```

We can create a new dataset based on this new higher resolution data:

```
In [13]: yhat_new = X_vdm_new@w_lsr
```

```
fig, ax = plt.subplots()
ax.plot(x, y, 'o')
ax.plot(x_new, yhat_new, '--');
```



Discussion: What happens as the order is increased? Does the model always improve?

We will discuss more techniques for validating the model and assessing the best order of the polynomial in future lectures.

General Linear Regression

We just saw that a model of the form:

$\vec{y} = \bar{\bar{X}}\vec{w} + \vec{\epsilon}$

can be used for simple linear regression (if $\bar{\bar{X}}$ is a first-order Vandermonde matrix) or polynomial regression (if $\bar{\bar{X}}$ is a higher-order Vandermonde matrix). In fact, this form can be used for many different types of linear regression and is referred to as a **general linear model**. Note that this is different from a *generalized linear model*, where the error term is assumed to follow a distribution other than normal. This is very confusing, but not terribly relevant in practice.

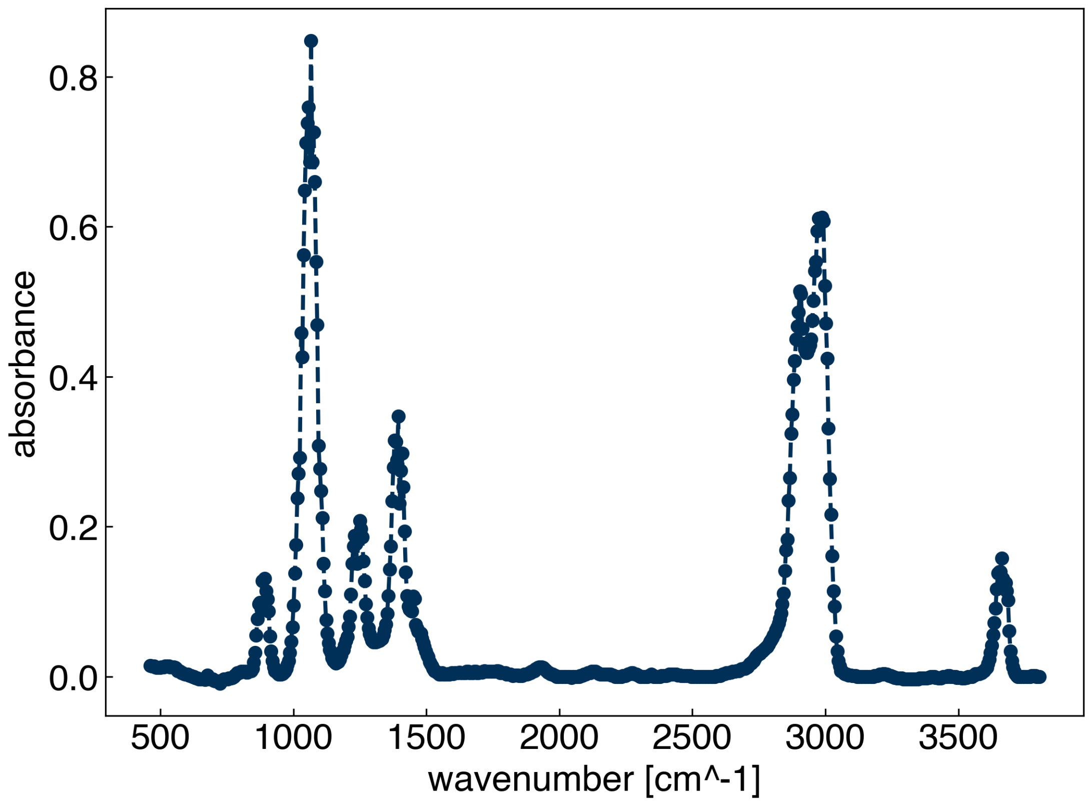
The key concept is that the columns of $\bar{\bar{X}}$ can contain any type(s) of linearly-dependent non-linear functions and the math will remain the same:

$\bar{\bar{X}}^T\bar{\bar{X}}\vec{w}^* = \bar{\bar{X}}^T\vec{y}$

where $\vec{w}^*$ are the optimal least-squares parameters.

We will call the columns of $\bar{\bar{X}}$ the "basis functions" for general linear regression. One common technique is the use of Gaussians as the basis functions. We can demonstrate this with a more realistic dataset. We will load in a dataset from infrared spectroscopy of an ethanol molecule. The data was obtained from NIST (https://webbook.nist.gov/cgi/cbook.cgi?ID=C64175&Units=SI&Type=IR-SPEC&Index=2#IR-SPEC), and we will use a new library called `pandas` to read it in. You will learn more about this in the "Data Management" module, but for now you just need to run the cell below:

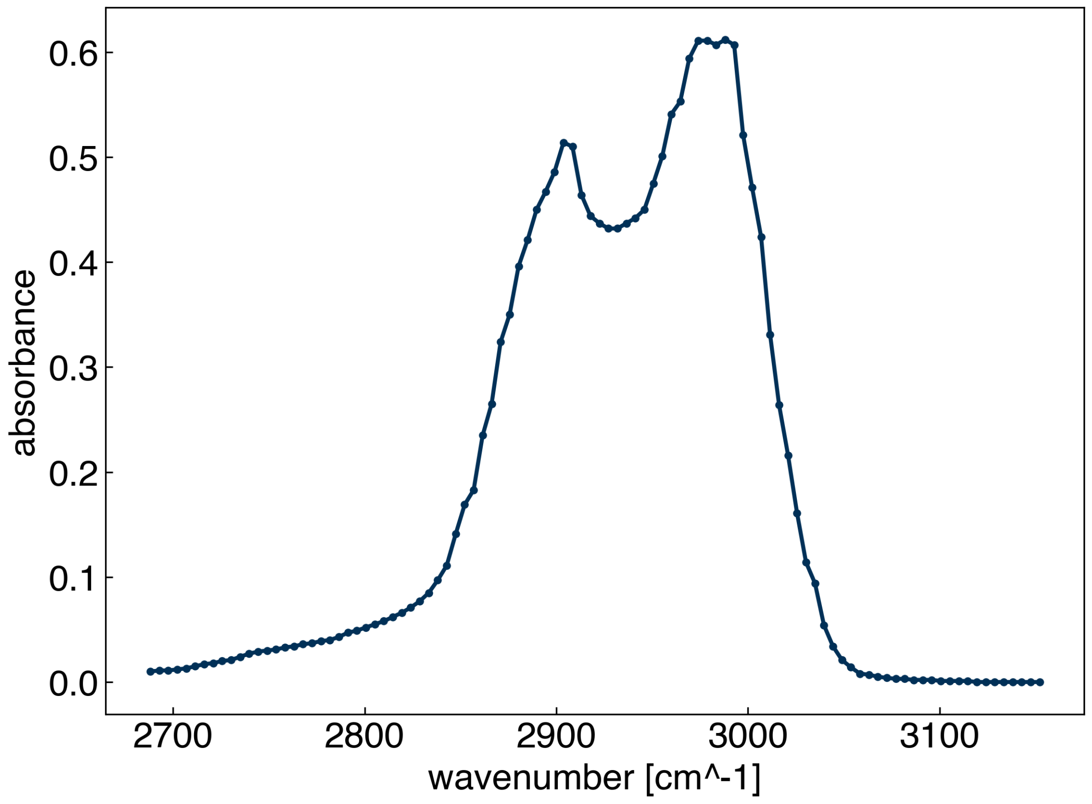
```
In [14]: import pandas as pd
         from matplotlib import pyplot as plt

         df = pd.read_csv('data/ethanol_IR.csv')
         x_all = df['wavenumber [cm^-1]'].values
         y_all = df['absorbance'].values

         fig, ax = plt.subplots()
         ax.plot(x_all,y_all)
         ax.set_xlabel('wavenumber [cm^-1]')
         ax.set_ylabel('absorbance');
```



This data looks a lot more complicated than our toy dataset from before. You don't need to understand the details of chemistry or infrared spectroscopy for this exercise. The key to analyzing these spectra is recognizing that the size and position of the peaks tells us information about the nature of the molecule. Let's make things easier by just selecting one of the peaks:

```
In [15]: x_peak = x_all[475:575]
         y_peak = y_all[475:575]
```

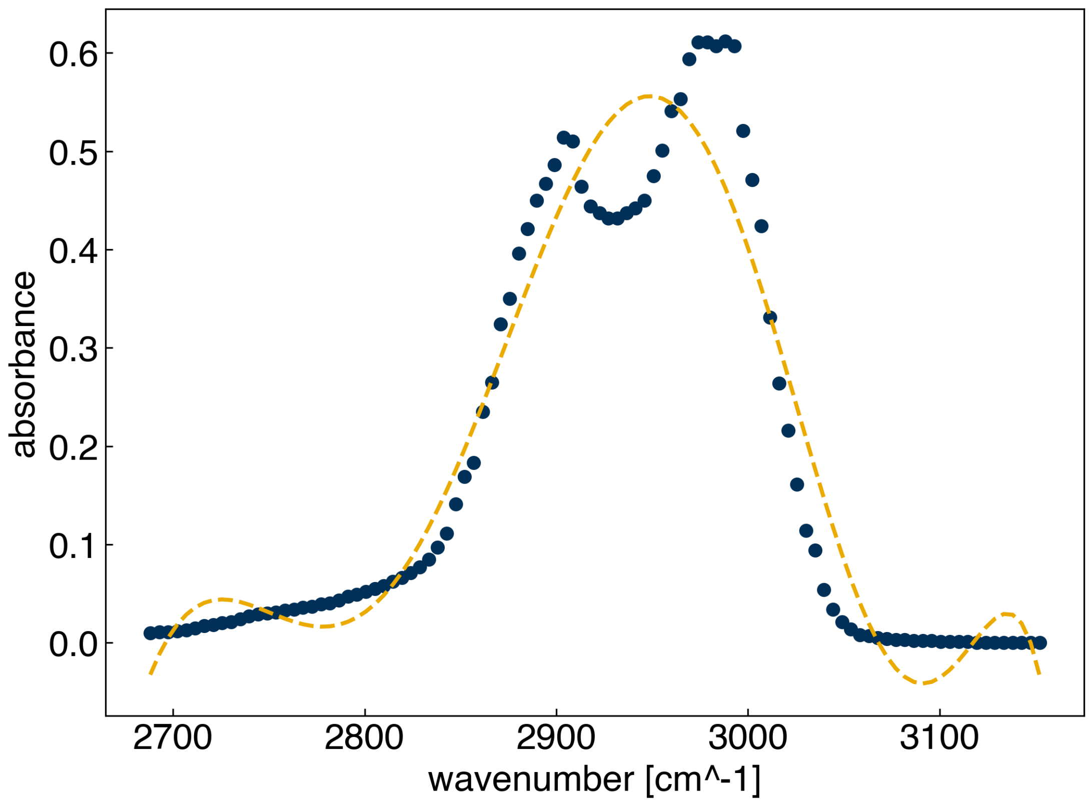
```
fig, ax = plt.subplots()
ax.plot(x_peak,y_peak, '-', marker='.')
ax.set_xlabel('wavenumber [cm^-1]')
ax.set_ylabel('absorbance');
```



Let's see what happens if we try to fit this with the polynomials from before:

```
In [16]: m = 20

         x_peak = x_peak.reshape(-1, 1) #create a column vector
         X_vdm = vandermonde(x_peak, m) #generate Vandermonde matrix
         b_m = np.dot(X_vdm.T, y_peak) #generate b vector with new features
         A_m = np.dot(X_vdm.T, X_vdm) #generate A matrix with new features
         w_m = np.linalg.solve(A_m, b_m) #solve Ax=b with new features
         print('Weights: {}'.format(w_m))

         yhat_m = np.dot(X_vdm, w_m) #compute predictions
         SSE_m = np.sum((y_peak - yhat_m)**2) #compute sum of squared errors
         print('Sum of Squared Errors: {}'.format(SSE_m))

         fig, ax = plt.subplots()
         ax.plot(x_peak, y_peak, 'o')
         ax.plot(x_peak, yhat_m, '--')
         ax.set_xlabel('wavenumber [cm^-1]')
         ax.set_ylabel('absorbance');
```

```
Weights: [ 7.03107204e+04 -2.48441196e+00 -4.67031587e-02  4.72309134e-
06
  8.17065392e-09 -5.82965454e-13  2.20483928e-17 -2.18438231e-19
 -2.31557743e-23 -1.38025477e-27 -4.33126419e-30  4.72375168e-33
  9.03665065e-37 -3.92292263e-41 -1.60712746e-43 -4.06640311e-47
  1.62313438e-50 -3.49613454e-55  5.63975509e-58 -1.44629384e-61]
Sum of Squared Errors: 0.29843275865015
Sum of Squared Errors: 0.29843275865015
```



Even if we add a lot of polynomials, the fit does not look very good. We never seem to capture the two peaks. Instead of polynomials we can see that the two peaks in the spectra look like the Gaussian distributions:

$$N(x,\ \mu,\ \sigma) = \frac{1}{2\pi\sigma^2}\exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

We can use Gaussian functions, instead of polynomials, to construct our model:

$$y_i = \sum_j w_j \exp\left(-\frac{(x_i-\mu_j)^2}{2\sigma_j^2}\right)$$

Note the similarity to the polynomial expression:

$$y_i = \sum_j w_j {x_i}^j$$

Instead of transforming x by making polynomials, we instead transform it by making Gaussian distributions. However, there is a catch. We need to set the mean, μ_j, and standard deviation, σ_j for each distribution.

Let's first try to do this manually. If we look at the data, we can see that one peak is around 2900 and the other is around 2980. We can also guess that the standard deviation is around 25 wavenumbers.

Let's write this out:

$$y_i = w_0\exp\left(-\frac{(x_i-2900)^2}{2(25^2)}\right) + w_1\exp\left(-\frac{(x_i-2980)^2}{2(25^2)}\right)$$

Next, we can consider the transformed x vectors as new vectors, $\tilde{x}$:

$y_i = w_0 \tilde{x}_{i,\ 0} + w_1 \tilde{x}_{i,\ 1}$

where

$$\tilde{x}_{i,\ 0} = \exp\left(-\frac{(x_i - 2900)^2}{2(25^2)} \right)$$

and

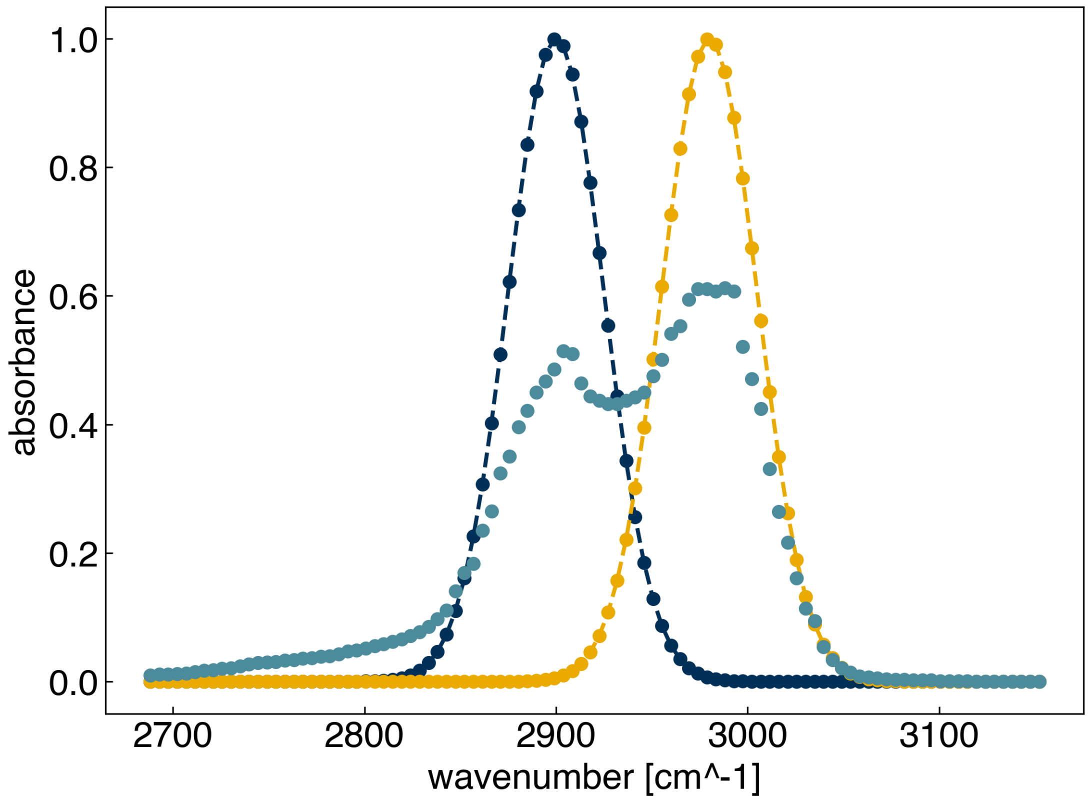
$$\tilde{x}_{i,\ 1} = \exp\left(-\frac{(x_i - 2980)^2}{2(25^2)} \right)$$

Now, we can re-write the model as:

$y_i = \sum_{j\ =\ 0}^{j\ =\ 1} w_j \tilde{X}_{ij}$

We can see that this is identical to the general linear model, but the matrix $\tilde{X}_{ij}$ is constructed in a different way. Let's implement this one in Python:

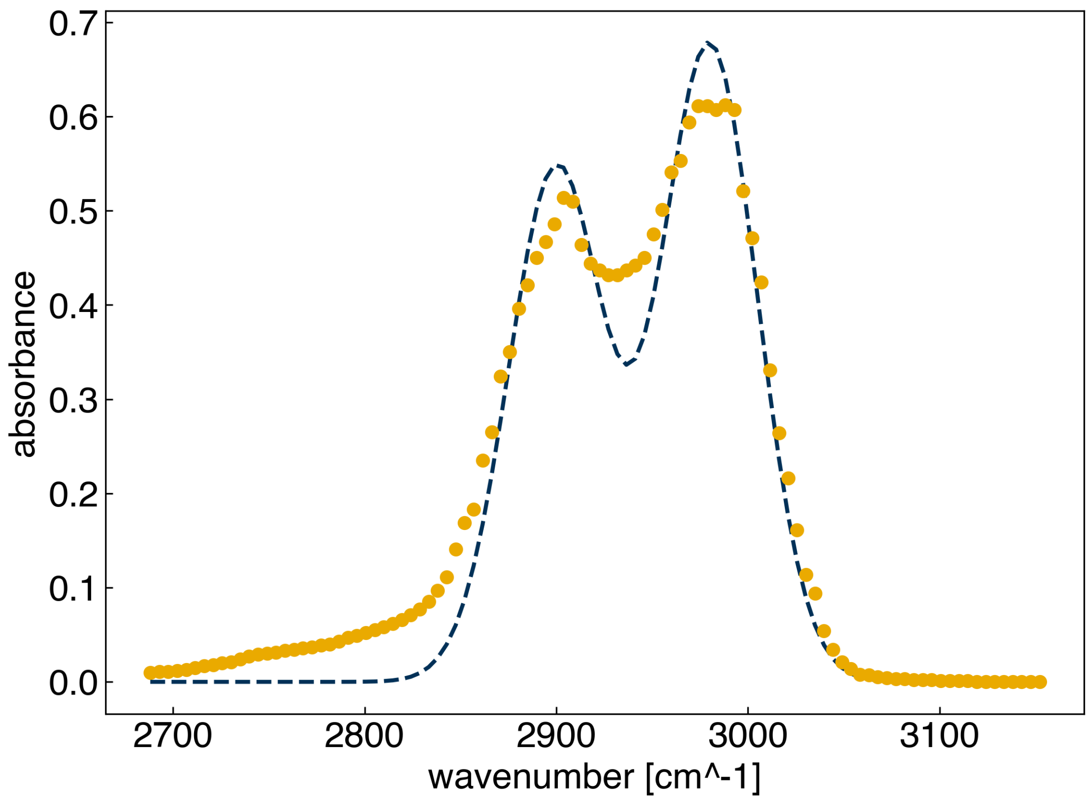
In [17]:
```
x_peak = x_peak.reshape(-1) #convert x_peak back into a vector
X_gauss = np.zeros((len(x_peak), 2))
X_gauss[:,0] = np.exp(-(x_peak - 2900)**2/(2*(25**2)))
X_gauss[:,1] = np.exp(-(x_peak - 2980)**2/(2*(25**2)))

fig, ax = plt.subplots()
ax.plot(x_peak, X_gauss[:,0])
ax.plot(x_peak, X_gauss[:,1])
ax.plot(x_peak, y_peak, 'o')
ax.set_xlabel('wavenumber [cm^-1]')
ax.set_ylabel('absorbance');
```



These are our "basis vectors", and our goal is to find the weights that best match the data. We can do this with the same exact math as before!

In [18]:
```
A = X_gauss.T@X_gauss
b = X_gauss.T@y_peak
w_lsr = np.linalg.solve(A,b)
yhat = X_gauss@w_lsr
print('Weights from least-squares regression: {}'.format(w_lsr))

fig, ax = plt.subplots()
ax.plot(x_peak, yhat, '--')
ax.plot(x_peak, y_peak, 'o')
ax.set_xlabel('wavenumber [cm^-1]')
ax.set_ylabel('absorbance');
```

```
Weights from least-squares regression: [0.54548962 0.67533912]
```



This looks much better than the polynomial fit, and we only needed 2 parameters! We can also gain some insight from these parameters, since they tell us the relative amounts of the different peaks.

However, this is a little unsatisfying since we did have to know the positions and widths of the peaks, and there are still substantial deviations between the model and the data. We will discuss some other strategies for dealing with this in future lectures, but for now let's see what we can do with the "general linear model" framework.

Let's see what happens if we add more Gaussian peaks. Instead of picking them manually, we can just use a set number of Gaussians with a fixed standard deviation, and space them evenly. We can write a function for this:

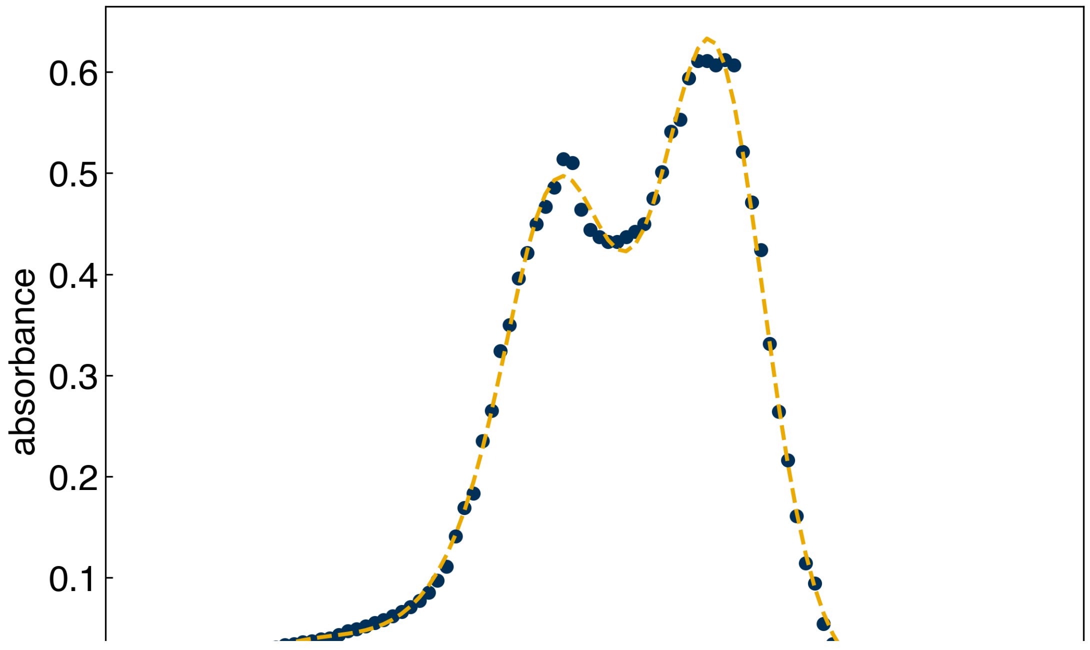
In [19]:
```
def gaussian_features(x, N , sigma = 25):
    # x is a vector
    # sigma is the standard deviation
    x = x.reshape(-1) #ensure that x is a vector
    xk_vec = np.linspace(min(x), max(x), N)
    features = []
    for xk in xk_vec:
        features.append(np.exp(-((x - xk)**2/(2*sigma**2))))
    return np.array(features).T
```

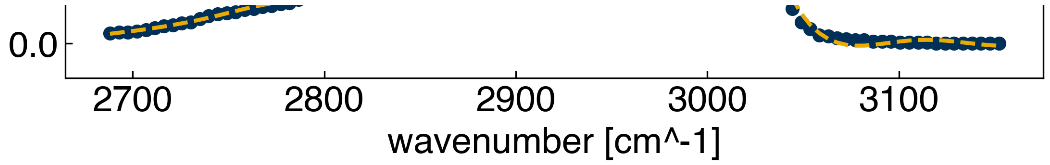
Note that we have used some significantly different strategies to create our matrix here. There are always multiple ways to do things in Python.

Now we can use the `gaussian_features` function similarly to the `vandermonde` function to create a matrix $\bar{\bar{X}}$ for our general linear model:

In [20]:
```
m = 12

X_gauss = gaussian_features(x_peak, m) #generate Vandermonde matrix
b_m = np.dot(X_gauss.T, y_peak) #generate b vector with new features
A_m = np.dot(X_gauss.T, X_gauss) #generate A matrix with new features
w_m = np.linalg.solve(A_m, b_m) #solve Ax=b with new features
print('Weights: {}'.format(w_m))

yhat_m = np.dot(X_gauss, w_m) #compute predictions
SSE_m = np.sum((y_peak - yhat_m)**2) #compute sum of squared errors
print('Sum of Squared Errors: {}'.format(SSE_m))

fig, ax = plt.subplots()
ax.plot(x_peak, y_peak, 'o')
ax.plot(x_peak, yhat_m, '--')
ax.set_xlabel('wavenumber [cm^-1]')
ax.set_ylabel('absorbance');
```

```
Weights: [ 0.00618991  0.01283186  0.02895143  0.02964597  0.08421234
0.42542746
  0.18814892  0.574715    0.02757571 -0.00945447  0.00639036 -0.0040002
9]
Sum of Squared Errors: 0.007590570362196462
```





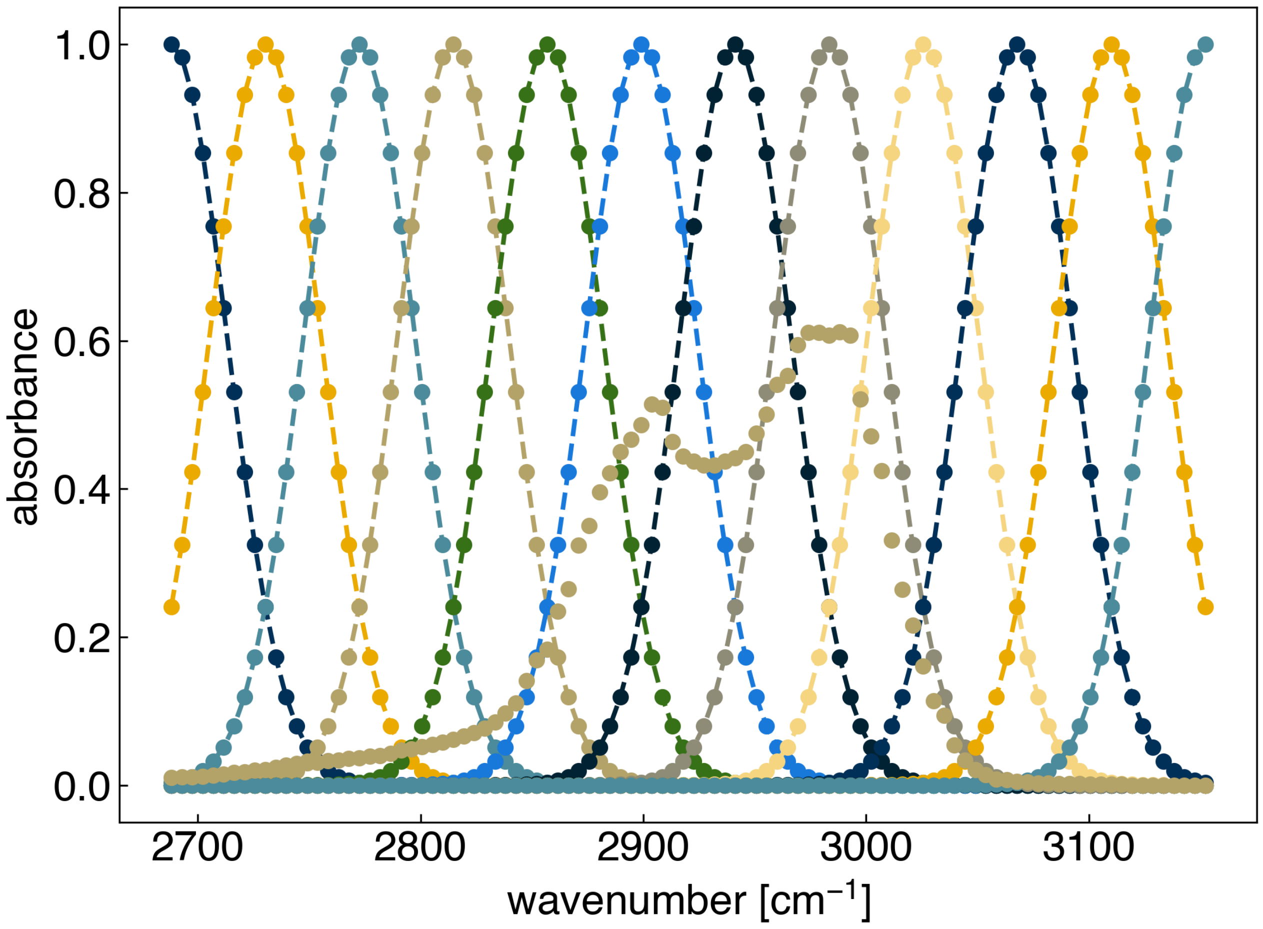
Now we see that the fit gets much better as more possible peaks are added. We can look at the w vector to see which peaks have large coefficients and use this to determine where peaks might be. However, there are a few problems with this approach:

1) The peak width is fixed, so it may take multiple Gaussians to represent a wider peak.

2) Some of the coefficients are negative, and we know that peaks should not have negative absorption.

To address these challenges we will need to move beyond general linear regression and use non-linear techniques. This will be explored in the next section.

Exercise: Plot the Gaussian basis functions used in the previous code block.



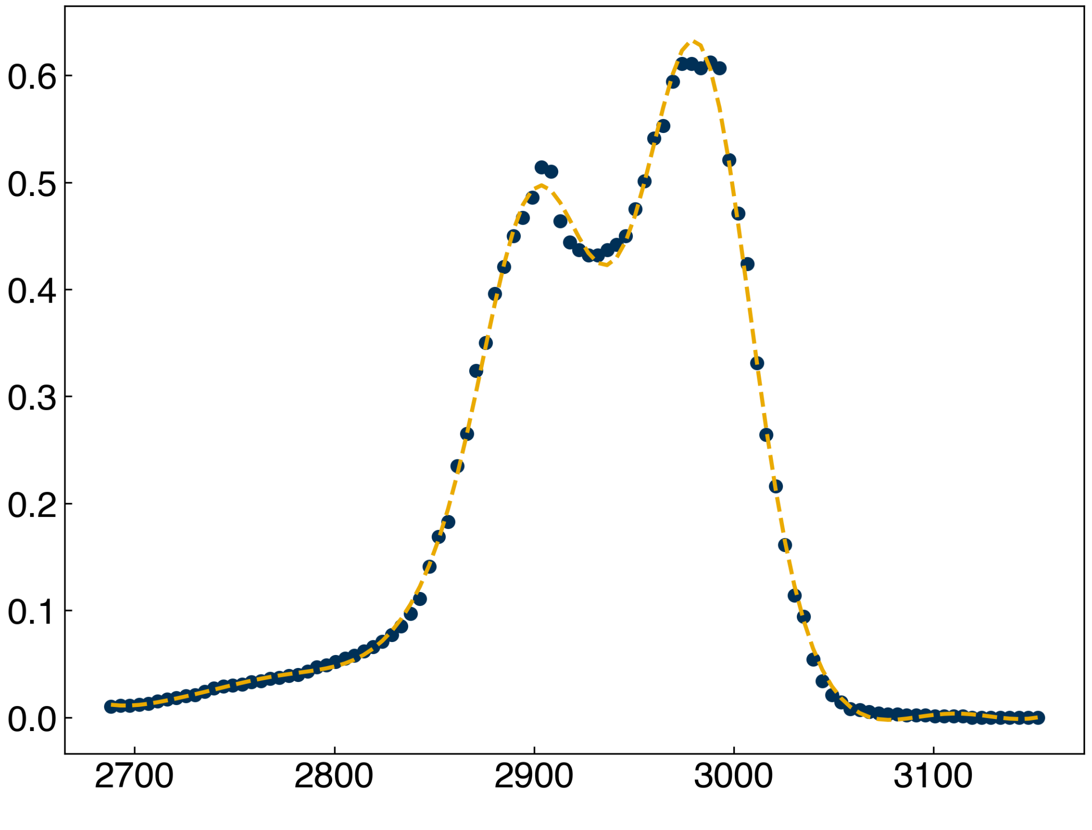
Linear Regression in Scikit-Learn

So far we have solved our general linear regression models directly from linear algebra. This is relatively easy, but it still requires us to set up a linear system and solve it. There is a very useful

Python package called `scikit-learn` that has implementations of many commonly-used algorithms, including general linear models.

We will introduce `scikit-learn` here to show how it can make things simpler. One thing to note is that `scikit-learn` only solves the regression part of the problem, so we still need to set up the feature matrix, X_{ij}. We will keep working with the spectra example, and us the `gaussian_features` function that we wrote earlier:

In [22]:
```
from sklearn.linear_model import LinearRegression

m = 12

X_m = gaussian_features(x_peak,m,sigma=25) #generate features

model = LinearRegression() #create a linear regression model instance

model.fit(X_m, y_peak) #fit the model (equivalent to the linear solve)

yhat = model.predict(X_m) #create the model prediction (equivalent to t
he matrix multiplication)

fig, ax = plt.subplots()
ax.plot(x_peak, y_peak, 'o')
ax.plot(x_peak, yhat, '--');
```



We can see that this requires much less code. However, `scikit-learn` is heavily "object" or "class" based, which can make the syntax confusing if you aren't familiar with Python. The advantage is that it is very easy to change the model and compare performance, as we will see in future lectures.

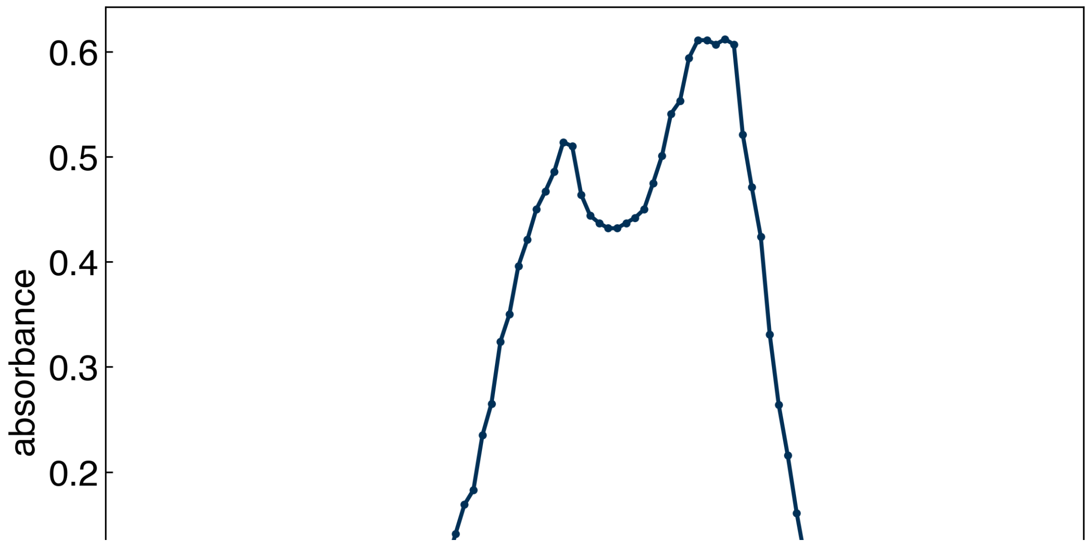
Table of Contents

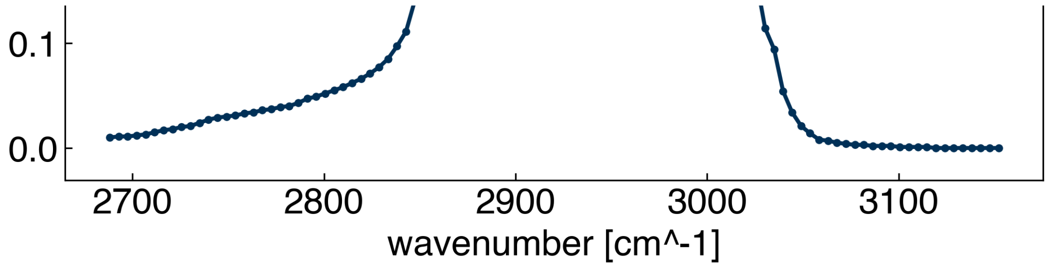
Numerical Optimization

In this lecture we will continue to work with the ethanol peaks dataset and look at numerical optimization from the perspective of non-linear regression.

First, we can re-load the dataset and select the same region we were working on before:

In [1]:
```
%matplotlib inline
import pandas as pd
from matplotlib import pyplot as plt
plt.style.use('../settings/plot_style.mplstyle')

df = pd.read_csv('data/ethanol_IR.csv')
x_all = df['wavenumber [cm^-1]'].values
y_all = df['absorbance'].values

x_peak = x_all[475:575]
y_peak = y_all[475:575]

fig, ax = plt.subplots()
ax.plot(x_peak,y_peak, '-', marker='.')
ax.set_xlabel('wavenumber [cm^-1]')
ax.set_ylabel('absorbance');
```





Non-linear Regression

In the prior lecture we considered "general linear models" that followed the form:

$$y_i = \sum_j w_j X_{ij} + \epsilon_i$$

and all non-linear behavior has been captured by using non-linear transforms of x_i. However, in some cases we may want to optimize models that are not linear. For example, consider the Gaussian peak problem:

$$y_i = w_0 \exp\left(-\frac{(x_i - \mu_0)^2}{2(\sigma_0^2)}\right) + w_1 \exp\left(-\frac{(x_i - \mu_1)^2}{2(\sigma_1^2)}\right) + \epsilon_i$$

Previously we just guessed values for μ_i and σ_i. However, it would be better if we could determine them from the data. Let's go back to the derivation of the linear regression equations. Remember that our goal is to minimize the sum of squared errors:

$$g = \sum_i \epsilon_i^2$$

We can solve for ϵ_i from the model:

$$\epsilon_i = y_i - w_0 \exp\left(-\frac{(x_i - \mu_0)^2}{2(\sigma_0^2)}\right) - w_1 \exp\left(-\frac{(x_i - \mu_1)^2}{2(\sigma_1^2)}\right) = y_i - \sum_j w_j G(x_i, \mu_j, \sigma_j)$$

where $G(x_i, \mu_j, \sigma_j) = \exp\left(-\frac{(x_i - \mu_j)^2}{2(\sigma_j^2)}\right)$.

and substitute:

$$g = \sum_i (y_i - \sum_j w_j G(x_i, \mu_j, \sigma_j))^2$$

Now our loss function depends on all the parameters, w_j, μ_j, and σ_j!

$$g(w_j, \mu_j, \sigma_j) = \sum_i (y_i - \sum_j w_j G(x_i, \mu_j, \sigma_j))^2$$

Let's introduce a new vector, $\vec{\lambda}$, that is a vector containing all the parameters:

$$\vec{\lambda} = [\vec{w}, \vec{\mu}, \vec{\sigma}]$$

We can do this since $\lambda_{i \leq m}$ contains the weights, $\lambda_{m<i \leq 2m}$ contains the means, and $\lambda_{i>2m}$ contains the standard deviations. This is convenient since we can now write:

$$g(\lambda_j) = \sum_{i=0}^{m} (y_i - \sum_j \lambda_j G(x_i, \lambda_{m+j}, \lambda_{2m+j}))^2$$

and we can minimize the loss by setting the derivative equal to zero:

$$\frac{\partial g}{\partial \lambda_j} = 0$$

This may look scary, but we are actually just using multivariate calculus exactly like we did for linear regression. However, we are stuck with two new problems:

(1) Getting the derivative $\frac{\partial g}{\partial \lambda_j}$ will be very complicated.

(2) We need a way to find the point where $\frac{\partial g}{\partial \lambda_j} = 0$.

In the case of linear regression, we derived this with matrix algebra then solved the resulting equations but that will be much more difficult in this case, and will not be general to other non-linear forms. Instead, we will use numerical methods this time around.

First, we need to implement our loss function, g, which we will call `gaussian_loss` since it results from a sum of m Gaussians:

In [2]:
```
import numpy as np

def gaussian_loss(lamda, x, y, m=2):
    yhat = np.zeros(len(y))
    for i in range(m):
        w_i = lamda[i]
        mu_i = lamda[m+i]
        sigma_i = lamda[2*m+i]
        yhat = yhat + w_i*np.exp(-(x - mu_i)**2/(2*sigma_i**2))
    squared_error = (y - yhat)**2
    return np.sum(squared_error)/len(y)
```

Let's do a sanity check by generating some data and testing the loss function:

In [3]:
```
x = np.linspace(-1,1,20)
y = 0.3*np.exp((-(x-0.2)**2)/(2*(0.5**2))) #create a Gaussian with w=0.
3, mu=0.2, sigma=0.5
y = y + 0.7*np.exp(-(x-0.5)**2/(2*0.1**2)) #add a Gaussian with w=0.7,
 mu=0.5, sigma=0.1
lamda = [0.3, 0.7, 0.2, 0.5, 0.5, 0.1] #create a "lamda" vector that sh
ould result in the same dataset
test_loss = gaussian_loss(lamda, x, y, m=2)
test_loss
```

```
Out[3]: 0.0
```

It looks like our loss function is working properly. Now we need to know how to find the derivative.

Automatic Differentiation

Derivatives are needed a lot in machine learning. One development that has emerged from the fields of optimization and computer science is the idea of automatic differentiation (https://en.wikipedia.org/wiki/Automatic_differentiation), also sometimes called "algorithmic differentiation". This is crucial to the success of well-known machine learning packages like "TensorFlow". The details of how it works are far too advanced for this course, and we will not use it often. However, it is definitely worth knowing about since many engineering applications also require derivatives.

The simple version is that automatic differentiation does exactly what it sounds like: it gives you the derivative of a function automatically! We do need to use some special tools to do this in Python. The `autograd` package is the simplest, since it works well with `numpy`. We also need to write our functions in a specific way so that they only take one argument.

In [4]:
```
# ! pip install autograd #<- use this block (or the command after ! in
 the conda prompt) to install autograd
```

In [5]:
```
import autograd.numpy as np   # autograd has its own "version" of numpy
that must be used
from autograd import grad # the "grad" function provides derivatives

def g(lamda, x=x, y=y, m=2):
    return gaussian_loss(lamda, x, y, m)

diff_g = grad(g)
print(g(lamda))
print(diff_g(lamda))
diff_g
```

```
0.0
[array(0.), array(0.), array(0.), array(0.), array(0.), array(0.)]
```

Out[5]:
```
<function autograd.wrap_util.unary_to_nary.<locals>.nary_operator.<loca
ls>.nary_f(*args, **kwargs)>
```

If you are not familiar with Python, this may look very odd. Essentially we are "wrapping" the `gaussian_loss` function in a new function `g`. Unlike `gaussian_loss`, `g` only takes a single argument, `lamda`, which is the argument we want to differentiate with respect to.

It is also worth noting that the `grad` function returns a function, not a value. This will also feel odd if you are new to Python. However, it is very convenient, because now we can use the new *function* `diff_g` to compute the derivative at any arbitrary value of λ:

In [6]:
```
print(diff_g(lamda))
lamda
```

```
[array(0.), array(0.), array(0.), array(0.), array(0.), array(0.)]
```

Out[6]:
```
[0.3, 0.7, 0.2, 0.5, 0.5, 0.1]
```

This is another sanity check: we know that the derivative should be zero if we are already at the optimum!

Let's try with some other guess for λ_j:

In [7]:
```
bad_guess = [0.1, 1.0, 0.5, 0.3, 0.1, 0.4]
print(g(bad_guess))
print(diff_g(bad_guess))
```

```
0.09629971881545715
[array(0.04543042), array(0.32856694), array(-0.01728235), array(-0.098
07934), array(0.08818821), array(0.39993077)]
```

Now we have solved the first problem: we know how to get $\frac{\partial g}{\partial \lambda_j}(\lambda_j)$. However, we do not have an analytical form (i.e. we can't write it down), so we still don't know how to solve for $\frac{\partial g}{\partial \lambda_j} = 0$.

Gradient Descent

There are many numerical techniques for solving the problem of finding $\vec{\lambda}^*$ such that $\frac{\partial g}{\partial \lambda_j}(\vec{\lambda}^*) = 0$. The two basic approaches, which should be familiar, are:

- Newton's method: Treat this as a root finding problem and use the second derivative, $\frac{\partial^2 L}{\partial \lambda_j \lambda_k}$ to iteratively optimize.
- Gradient descent/ascent: Increase or decrease the guess by "walking" along the gradient.

These are typically "iterative" methods, which means we start with some initial guess then iteratively improve it.

The simplest approach is to use gradient descent with a fixed step size, which we will explore here:

In [8]:
```
better_guess = [0.35, 0.75, 0.21, 0.52, 0.53, 0.11]
guess = bad_guess
print('Initial Loss: {:.4f}'.format(g(guess)))

N_iter = 1000
h = 0.1
for i in range(N_iter):
    guess = guess - h*np.array(diff_g(guess))

print('Final Loss: {:.4f}'.format(g(guess)))
```

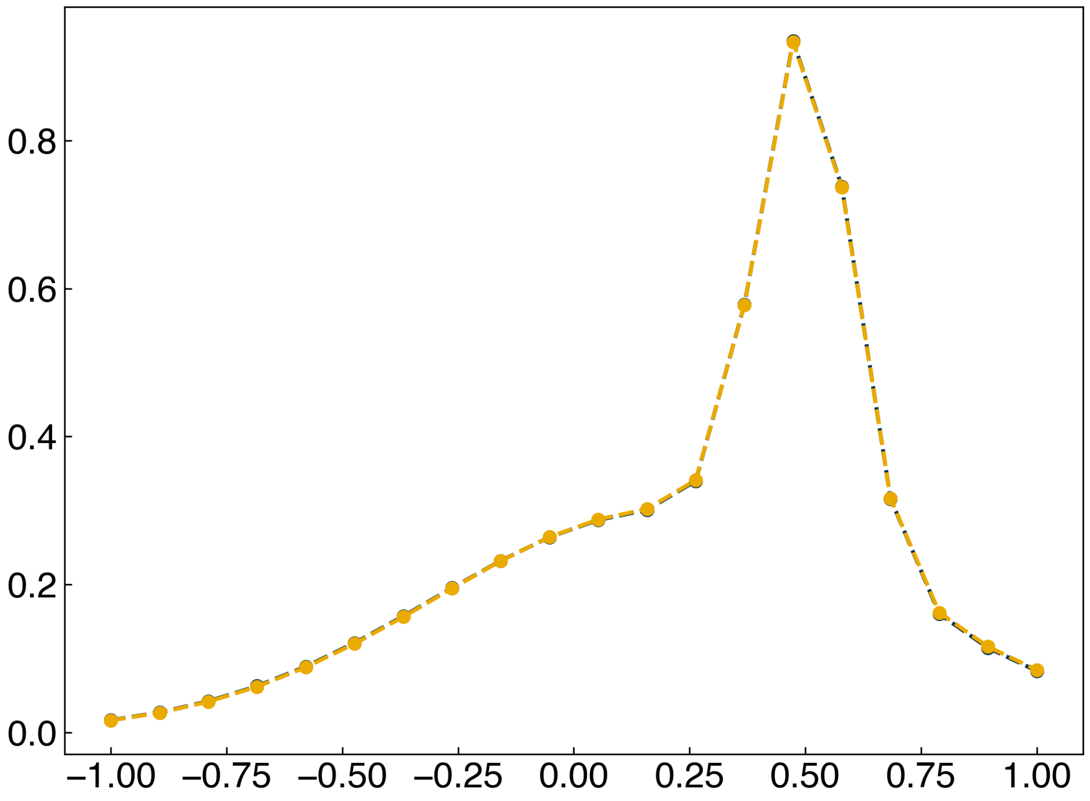
```
Initial Loss: 0.0963
Final Loss: 0.0000
```

We can see that the loss decreases after 100 iterations of gradient descent. Let's compare the results:

In [9]:
```
print('Actual Input: {}'.format(str(lamda)))
print('Regression Result: {}'.format(str(guess)))


def two_gaussians(lamda, x):
    w_0, w_1, mu_0, mu_1, sigma_0, sigma_1 = lamda
    y = w_0*np.exp((-(x-mu_0)**2)/(2*(sigma_0**2))) + w_1*np.exp(-(x-mu
_1)**2/(2*sigma_1**2))
    return y

y = two_gaussians(lamda, x)
yhat = two_gaussians(guess, x)
```

```
fig, ax = plt.subplots()
ax.plot(x, y)
ax.plot(x, yhat, ls='--');
```

```
Actual Input: [0.3, 0.7, 0.2, 0.5, 0.5, 0.1]
Regression Result: [0.69639616 0.30146582 0.49997652 0.20386904 0.09978
107 0.49952471]
```



We see that this looks pretty good! The parameters look different, but it turns out that they are pretty close if you switch the order of the two peaks.

Discussion: What happens if we change the parameters of the steepest descent algorithm (initial guess, number of steps, step size)?

The details of numerical algorithms for multi-dimensional optimization are tricky! However, most optimizers use a form of gradient descent. From now on we will typically let a built-in optimizer handle the hard work!

Optimization with Scipy

The `scipy` package is another commonly-used package that comes with lots of algorithms. In particular, there are a number of numerical optimization algorithms available through the `scipy.minimize` package. In practice, we will typically rely on `scipy.minimize` to handle minimization rather than writing our own algorithms. These algorithms will be more efficient, and have built-in techniques for estimating derivatives (or manage to optimize without derivatives at all).

One of the most commonly-used algorithms is the "BFGS" algorithm (https://en.wikipedia.org/wiki/Broyden%E2%80%93Fletcher%E2%80%93Goldfarb%E2%80%93Shanno_algorithn named after it's creators Broyden, Fletcher, Goldfarb, and Shanno. When in doubt, this is a good algorithm to try first. Let's see how it works for our problem:

In [10]:
```
from scipy.optimize import minimize

result = minimize(g, bad_guess, method='BFGS')
result
```

Out[10]:
```
      fun: 4.535834086328743e-13
 hess_inv: array([[ 1.08648965e+01, -2.31102013e+00,  8.04786981e-02,
        -4.86071429e+00, -1.20650239e-01,  1.95091675e-01],
       [-2.31102013e+00,  3.48437293e+00,  1.64287106e-01,
         2.27707455e+00, -5.99422521e-01, -2.63840446e+00],
       [ 8.04786981e-02,  1.64287106e-01,  2.64088306e-01,
        -4.39759929e-01, -9.86221948e-03, -4.11746751e-01],
       [-4.86071429e+00,  2.27707455e+00, -4.39759929e-01,
         1.28831480e+01, -1.04319377e+00,  1.39259269e+00],
       [-1.20650239e-01, -5.99422521e-01, -9.86221948e-03,
        -1.04319377e+00,  3.66085479e-01,  2.17736828e-02],
       [ 1.95091675e-01, -2.63840446e+00, -4.11746751e-01,
         1.39259269e+00,  2.17736828e-02,  9.05845978e+00]])
      jac: array([-1.50786322e-07,  2.82043245e-08, -8.94333746e-07, -
3.18042594e-07,
       -6.58598643e-07,  7.75125258e-08])
  message: 'Optimization terminated successfully.'
     nfev: 217
      nit: 28
     njev: 31
   status: 0
  success: True
        x: array([0.69999993, 0.29999964, 0.49999979, 0.19999771, 0.100
0001 ,
       0.50000067])
```

A few things to note here:

- We still had to use the function `g` that only takes a single argument (the variable we want to optimize with respect to).
- The output is not a single number, but rather a Python class with various attributes.
- This was really fast!

We can investigate the attributes of the output using the `dir` function:

In [11]:
```
dir(result)
```

Out[11]:
```
['fun',
 'hess_inv',
 'jac',
 'message',
 'nfev',
 'nit',
 'njev',
 'status',
 'success',
 'x']
```

Now we can check various aspects of the result using the `.` operator:

```
In [12]: result.success
```

```
Out[12]: True
```

```
In [13]: result.x
```

```
Out[13]: array([0.69999993, 0.29999964, 0.49999979, 0.19999771, 0.1000001 ,
                0.50000067])
```

This tells us that the optimization was successful, and gives us the final result. Let's compare this to the original input:

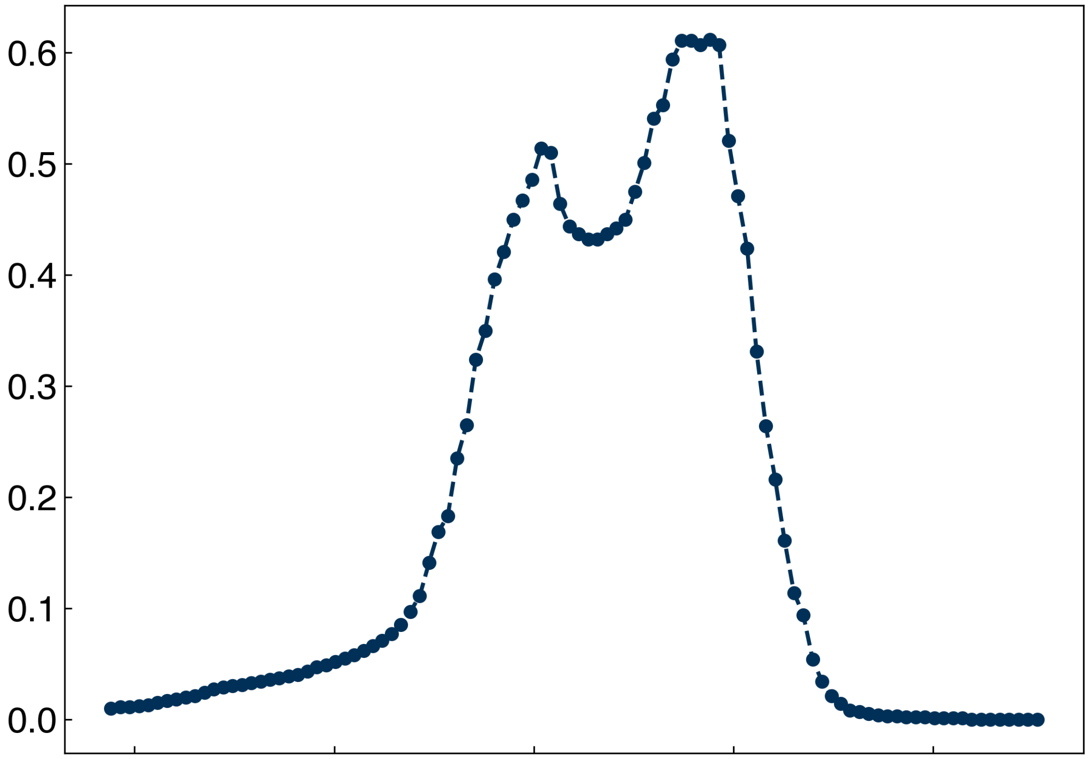
```
In [14]: print('Actual Input: {}'.format(str(result.x)))
         print('Regression Result: {}'.format(str(lamda)))
```

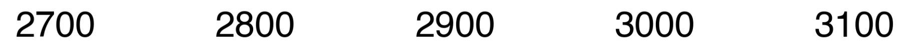
```
Actual Input: [0.69999993 0.29999964 0.49999979 0.19999771 0.1000001
0.50000067]
Regression Result: [0.3, 0.7, 0.2, 0.5, 0.5, 0.1]
```

We can see that this was much faster than our naive gradient descent, and is also more accurate (although the order of the peaks is still switched due to the bad initial guess).

Let's revisit the real spectra we worked with earlier and try to optimize the peak positions and widths:

```
In [15]: fig, ax = plt.subplots()
         ax.plot(x_peak, y_peak);
```



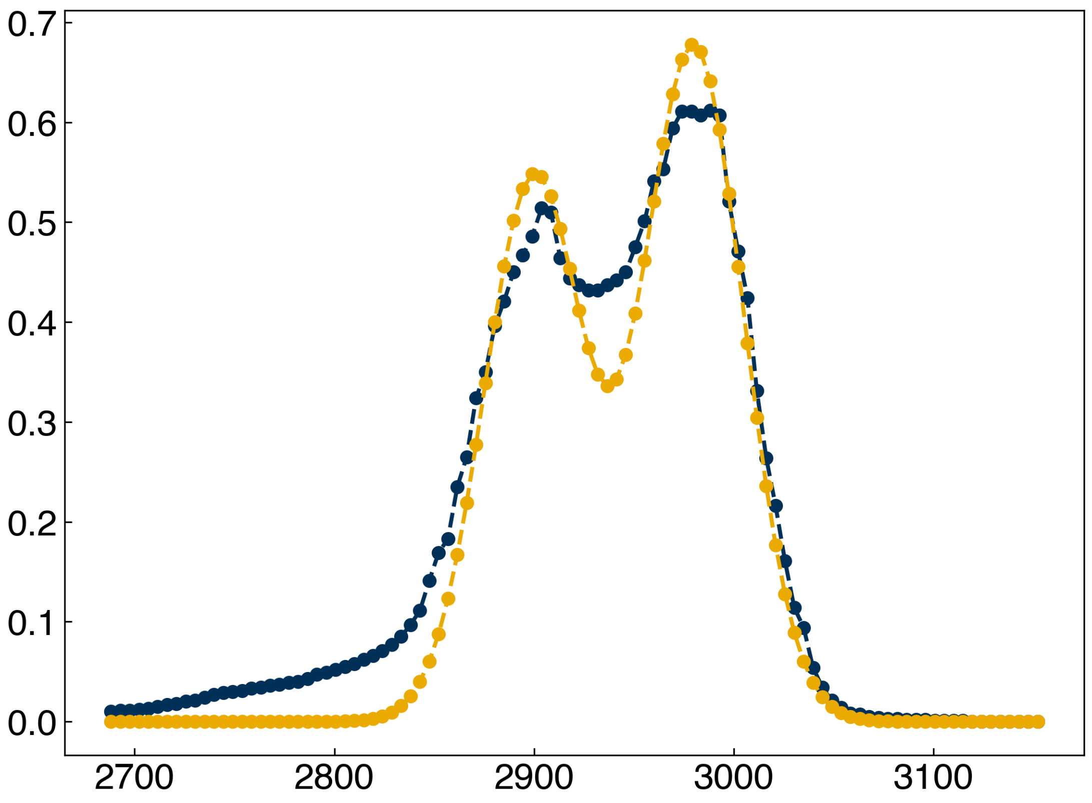


Remember that we used guesses of the peak position and width:

$$y_i = w_0 \exp\left(-\frac{(x_i - 2900)^2}{2(25^2)}\right) + w_1 \exp\left(-\frac{(x_i - 2980)^2}{2(25^2)}\right)$$

then we optimized the parameters, $\vec{w}$, and found $w_0 = 0.545$ and $w_1 = 0.675$. We can convert these parameters into the $\vec{\lambda}$ format use our `two_gaussians` function to check the initial guess:

In [16]:
```
guess = [0.545, 0.675, 2900, 2980, 25, 25]
y_guess = two_gaussians(guess, x_peak)

fig, ax = plt.subplots()
ax.plot(x_peak, y_peak)
ax.plot(x_peak, y_guess);
```



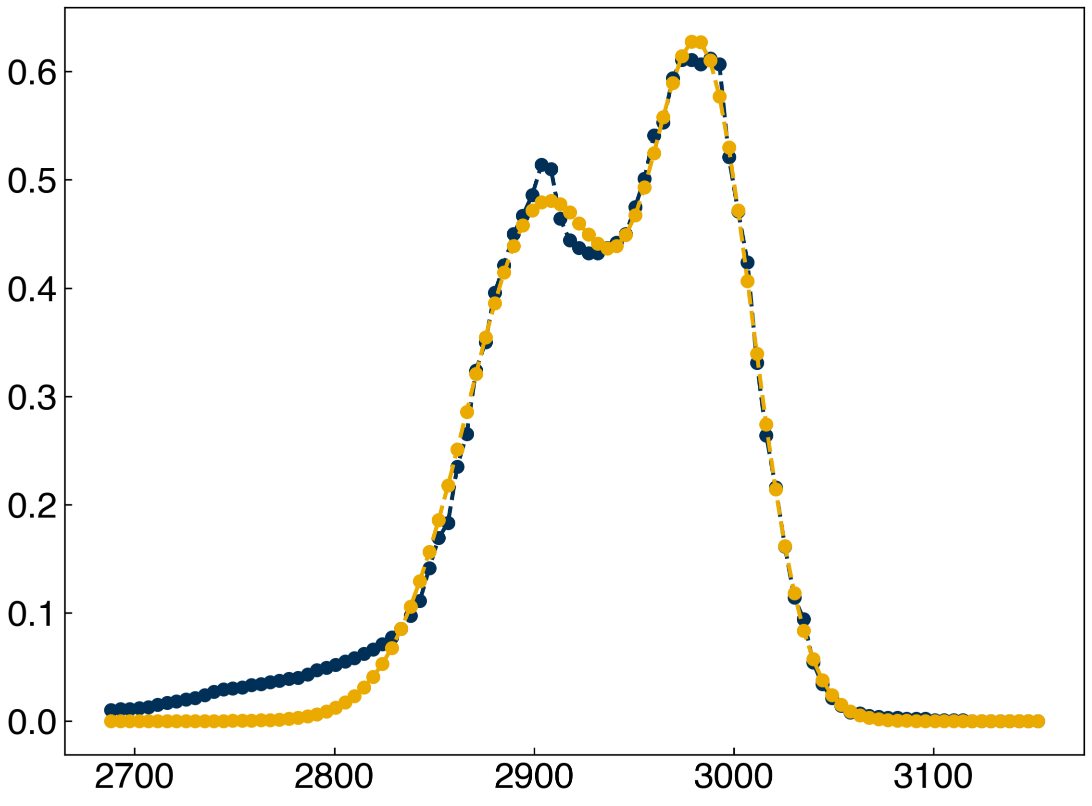
We can use the same loss function as before to optimize the other parameters:

In [17]:
```
def g(lamda, x=x_peak, y=y_peak, m=2):
    return gaussian_loss(lamda, x, y, m)

result = minimize(g, guess, method='BFGS')
result
```

Out[17]:
```
      fun: 0.0003472560734252687
 hess_inv: array([[ 7.06131339e+00,  1.32015580e+00,  2.02380230e+02,
         1.14148647e+02, -2.40727490e+02, -1.88763946e+02],
       [ 1.32015580e+00,  2.62539867e+01, -2.23167548e+03,
        -1.21555161e+03, -2.12368258e+03,  5.05422778e+02],
       [ 2.02380230e+02, -2.23167548e+03,  3.10111804e+05,
          1.57436153e+05,  2.28256174e+05, -1.01406914e+05],
        [ 1.14148647e+02, -1.21555161e+03,  1.57436153e+05,
          1.05767010e+05,  1.34286069e+05, -5.54776508e+04],
        [-2.40727490e+02, -2.12368258e+03,  2.28256174e+05,
          1.34286069e+05,  2.48853050e+05, -7.83859363e+04],
        [-1.88763946e+02,  5.05422778e+02, -1.01406914e+05,
         -5.54776508e+04, -7.83859363e+04,  5.66555361e+04]])
      jac: array([ 3.42427666e-06, -2.04337994e-07, -7.09405867e-09, -
9.84437065e-08,
        1.01063051e-07,  5.09317033e-09])
  message: 'Optimization terminated successfully.'
     nfev: 588
      nit: 79
     njev: 84
   status: 0
  success: True
        x: array([4.76315423e-01, 5.64860471e-01, 2.90549379e+03, 2.985
08206e+03,
       3.88244958e+01, 2.53865443e+01])
```

It looks successful! Let's see how well it worked:

```
In [18]: fitted = result.x
         y_fitted = two_gaussians(fitted, x_peak)

         fig, ax = plt.subplots()
         ax.plot(x_peak, y_peak)
         ax.plot(x_peak, y_fitted);
```

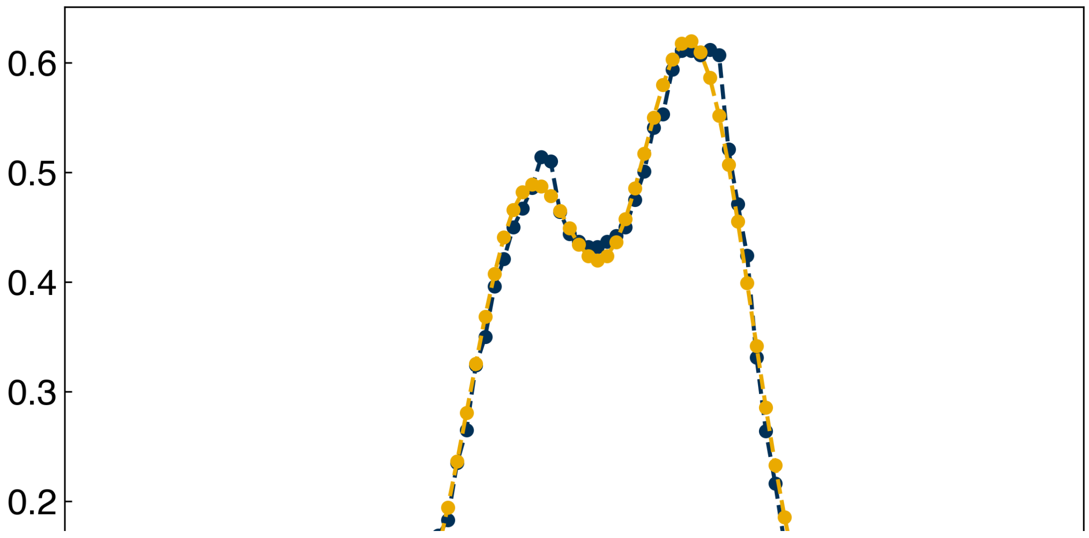


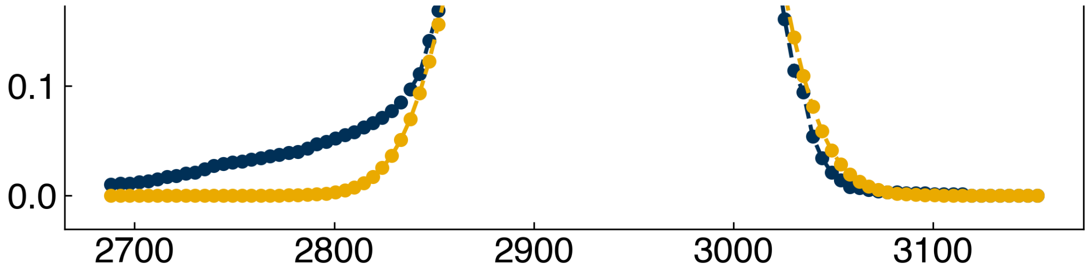
This looks much better!

We can also add constraints to the loss function. For example, we might expect that the peak width (standard deviation) should be similar for both peaks. We can enforce this by adding an additional term to the loss function:

In [19]:
```
def g_simwidth(lamda, x=x_peak, y=y_peak, N=2):
    return gaussian_loss(lamda, x, y, N) + (lamda[-2] - lamda[-1])**2

result = minimize(g_simwidth, guess, method='BFGS')
fitted = result.x
y_fitted = two_gaussians(fitted, x_peak)

fig, ax = plt.subplots()
ax.plot(x_peak, y_peak)
ax.plot(x_peak, y_fitted);
result
```

Out[19]:
```
      fun: 0.0005425736558171898
 hess_inv: array([[ 7.29406267e+00,  1.66690044e+00,  1.66723252e+02,
         9.73508763e+01, -2.07904190e+02, -2.07882416e+02],
       [ 1.66690044e+00,  7.23778080e+00, -6.31573103e+01,
        -5.85153311e+01, -2.08918747e+02, -2.08897266e+02],
       [ 1.66723252e+02, -6.31573103e+01,  5.07937975e+04,
         1.21496944e+04, -7.76852567e+03, -7.76870884e+03],
       [ 9.73508763e+01, -5.85153311e+01,  1.21496944e+04,
         2.20237169e+04,  6.42763677e+01,  6.43397043e+01],
       [-2.07904190e+02, -2.08918747e+02, -7.76852567e+03,
         6.42763677e+01,  1.61044075e+04,  1.61039659e+04],
       [-2.07882416e+02, -2.08897266e+02, -7.76870884e+03,
         6.43397043e+01,  1.61039659e+04,  1.61041109e+04]])
      jac: array([-7.10971653e-06,  2.52791506e-06, -2.78574589e-07, -
1.17989839e-06,
       -1.37514144e-06,  1.59677438e-06])
  message: 'Optimization terminated successfully.'
     nfev: 413
      nit: 55
     njev: 59
   status: 0
  success: True
        x: array([4.72325212e-01, 6.08042493e-01, 2.89704679e+03, 2.979
16802e+03,
       3.01336263e+01, 3.01336118e+01])
```





We can see that the fit quality is similar, but now the peak widths are nearly identical. However, they are not exactly the same, since the loss function constraint is "soft" -- the peak widths will deviate if it makes the fit much better.

Discussion: How would you create a loss function that ensures that all weights w_i are positive?