L2_PythonCrashCourse

August 17, 2017

1 Lecture 2: Python Crash Course

CSCI 4360/6360: Data Science II

1.1 Part 1: Python Background

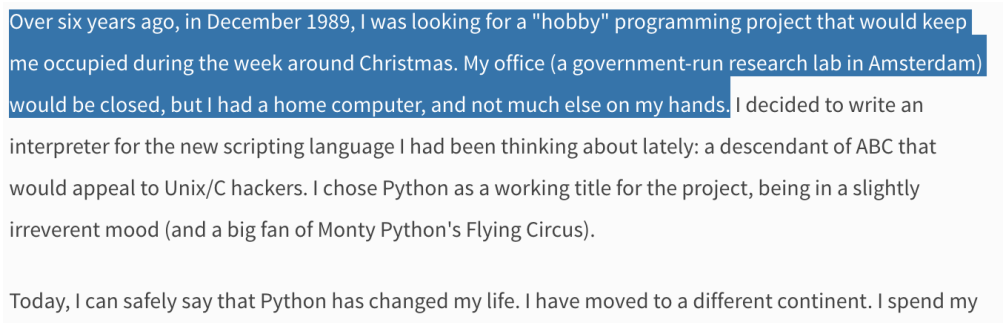
Python as a language was implemented from the start by Guido van Rossum. What was originally something of a snarkily-named hobby project to pass the holidays turned into a huge open source phenomenon used by millions.

1.1.1 Python's history

The original project began in 1989.

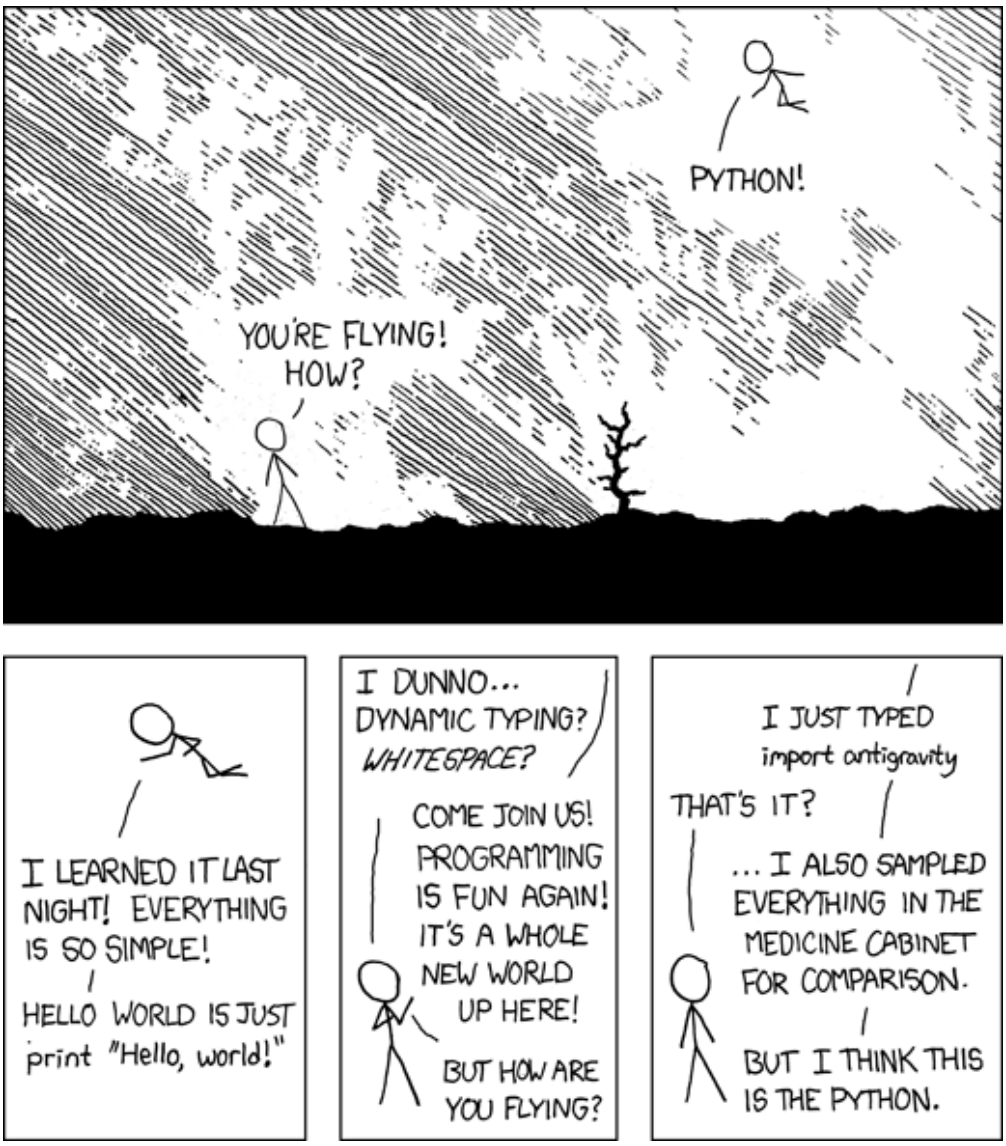
- Release of Python 2.0 in 2000
- Release of Python 3.0 in 2008
- Latest stable release of these branches are **2.7.13**--which Guido *emphatically* insists is the final, final, final release of the 2.x branch--and **3.6.2**.

You're welcome to use whatever version you want, just be aware: the AutoLab autograders will be using **3.6.x** (unless otherwise noted).


Over six years ago, in December 1989, I was looking for a "hobby" programming project that would keep me occupied during the week around Christmas. My office (a government-run research lab in Amsterdam) would be closed, but I had a home computer, and not much else on my hands. I decided to write an interpreter for the new scripting language I had been thinking about lately: a descendant of ABC that would appeal to Unix/C hackers. I chose Python as a working title for the project, being in a slightly irreverent mood (and a big fan of Monty Python's Flying Circus).

Today, I can safely say that Python has changed my life. I have moved to a different continent. I spend my

guido



xkcd

1.1.2 Python, the Language

Python is an **intepreted** language.

- Contrast with **compiled** languages
- Performance, ease-of-use
- Modern intertwining and blurring of compiled vs interpreted languages

Python is a very **general** language.

- Not designed as a specialized language for performing a specific task. Instead, it relies on third-party developers to provide these extras.

Instead, as Jake VanderPlas put it:

> "Python syntax is the glue that holds your data science code together. As many scientists and statisticians have found, Python excels in that role because it is powerful, intuitive, quick to write, fun to use, and above all extremely useful in day-to-day data science tasks."

1.2 Part 2: Language Basics

The most basic thing possible: Hello, World!

```
In [1]: print("Hello, world!")

Hello, world!
```

Yep, that's all that's needed!

(Take note: the biggest different between Python 2 and 3 is the `print` function: it technically wasn't a function in Python 2 so much as a *language construct*, and so you didn't need parentheses around the string you wanted printed; in Python 3, it's a full-fledged *function*, and therefore requires parentheses)

1.2.1 Variables and Types

Python is dynamically-typed, meaning you don't have to declare types when you assign variables. Python is also *duck-typed*, a colloquialism that means it *infers* the best-suited variable type at runtime ("if it walks like a duck and quacks like a duck...")

```
In [2]: x = 5
        type(x)

Out[2]: int

In [3]: y = 5.5
        type(y)

Out[3]: float
```

It's important to note: even though you don't have to specify a type, Python still assigns a type to variables. It would behoove you to know the types so you don't run into tricky type-related bugs!

```
In [4]: x = 5 * 5
```

What's the type for `x`?

```
In [5]: type(x)

Out[5]: int

In [6]: y = 5 / 5
```

What's the type for `y`?

```
In [7]: type(y)
```

```
Out[7]: float
```

There are functions you can use to explicitly *cast* a variable from one type to another:

```
In [8]: x = 5 / 5
        type(x)
```

```
Out[8]: float
```

```
In [9]: y = int(x)
        type(y)
```

```
Out[9]: int
```

```
In [10]: z = str(y)
         type(z)
```

```
Out[10]: str
```

1.2.2 Data Structures

There are four main types of built-in Python data structures, each similar but ever-so-slightly different:

1. Lists (the Python workhorse)
2. Tuples
3. Sets
4. Dictionaries

(Note: generators and comprehensions are worthy of mention; definitely look into these as well)

Lists are basically your catch-all multi-element data structure; they can hold anything.

```
In [11]: some_list = [1, 2, 'something', 6.2, ["another", "list!"], 7371]
         print(some_list[3])
         type(some_list)
```

```
6.2


Out[11]: list
```

Tuples are like lists, except they're *immutable* once you've built them (and denoted by parentheses, instead of brackets).

```
In [12]: some_tuple = (1, 2, 'something', 6.2, ["another", "list!"], 7371)
         print(some_tuple[5])
         type(some_tuple)
```

```
7371
```

```
Out[12]: tuple
```

Sets are probably the most different: they are mutable (can be changed), but are *unordered* and *can only contain unique items* (they automatically drop duplicates you try to add). They are denoted by braces.

```
In [13]: some_set = {1, 1, 1, 1, 1, 86, "something", 73}
         some_set.add(1)
         print(some_set)
         type(some_set)
```

```
{'something', 1, 86, 73}
```

```
Out[13]: set
```

Finally, dictionaries. Other terms that may be more familiar include: maps, hashmaps, or associative arrays. They're a combination of sets (for their *key* mechanism) and lists (for their *value* mechanism).

```
In [14]: some_dict = {"key": "value", "another_key": [1, 3, 4], 3: ["this", "value"]}
         print(some_dict["another_key"])
         type(some_dict)
```

```
[1, 3, 4]
```

```
Out[14]: dict
```

Dictionaries explicitly set up a mapping between a *key*--keys are unique and unordered, exactly like sets--to *values*, which are an arbitrary list of items. These are very powerful structures for data science-y applications.

1.2.3 Slicing and Indexing

Ordered data structures in Python are 0-indexed (like C, C++, and Java). This means the first elements are at index 0:

```
In [15]: print(some_list)
```

```
[1, 2, 'something', 6.2, ['another', 'list!'], 7371]
```

```
In [16]: index = 0
         print(some_list[index])
```

```
1
```

However, using colon notation, you can "slice out" entire sections of ordered structures.

```
In [17]: start = 0
         end = 3
         print(some_list[start : end])

[1, 2, 'something']
```

Note that the starting index is *inclusive*, but the ending index is *exclusive*. Also, if you omit the starting index, Python assumes you mean 0 (start at the beginning); likewise, if you omit the ending index, Python assumes you mean "go to the very end".

```
In [18]: print(some_list[:end])

[1, 2, 'something']
```

```
In [19]: start = 1
         print(some_list[start:])

[2, 'something', 6.2, ['another', 'list!'], 7371]
```

1.2.4 Loops

Python supports two kinds of loops: `for` and `while`

`for` loops in Python are, in practice, closer to *for each* loops in other languages: they iterate through collections of items, rather than incrementing indices.

```
In [20]: for item in some_list:
             print(item)

1
2
something
6.2
['another', 'list!']
7371
```

- the collection to be iterated through is at the end (`some_list`)
- the current item being iterated over is given a variable after the `for` statement (`item`)
- the loop body says what to do in an iteration (`print(item)`)

But if you need to iterate by index, check out the `enumerate` function:

```
In [21]: for index, item in enumerate(some_list):
             print("{}: {}".format(index, item))
```

```
0: 1
1: 2
2: something
3: 6.2
4: ['another', 'list!']
5: 7371
```

`while` loops operate as you've probably come to expect: there is some associated boolean condition, and as long as that condition remains `True`, the loop will keep happening.

```
In [22]: i = 0
         while i < 10:
             print(i)
             i += 2
```

```
0
2
4
6
8
```

IMPORTANT: Do not forget to perform the *update* step in the body of the `while` loop! After using `for` loops, it's easy to become complacent and think that Python will update things automatically for you. If you forget that critical `i += 2` line in the loop body, this loop will go on forever...

Another cool looping utility when you have multiple collections of identical length you want to loop through simultaneously: the `zip()` function

```
In [23]: list1 = [1, 2, 3]
         list2 = [4, 5, 6]
         list3 = [7, 8, 9]

         for x, y, z in zip(list1, list2, list3):
             print("{} {} {}".format(x, y, z))
```

```
1 4 7
2 5 8
3 6 9
```

This "zips" together the lists and picks corresponding elements from each for every loop iteration. Way easier than trying to set up a numerical index to loop through all three simultaneously, but you can even combine this with `enumerate` to do exactly that:

```
In [24]: for index, (x, y, z) in enumerate(zip(list1, list2, list3)):
             print("{}: ({}, {}, {})".format(index, x, y, z))
```

```
0: (1, 4, 7)
1: (2, 5, 8)
2: (3, 6, 9)
```

1.2.5 Conditionals

Conditionals, or `if` statements, allow you to branch the execution of your code depending on certain circumstances.

In Python, this entails three keywords: `if`, `elif`, and `else`.

```
In [25]: grade = 82
         if grade > 90:
             print("A")
         elif grade > 80:
             print("B")
         else:
             print("Something else")
```

```
B
```

A couple important differences from C/C++/Java parlance: - **NO** parentheses around the boolean condition! - It's not "`else if`" or "`elseif`", just "`elif`". It's admittedly weird, but it's Python

Conditionals, when used with loops, offer a powerful way of slightly tweaking loop behavior with two keywords: `continue` and `break`.

The former is used when you want to skip an iteration of the loop, but nonetheless keep going on to the *next* iteration.

```
In [26]: list_of_data = [4.4, 1.2, 6898.32, "bad data!", 5289.24, 25.1, "other bad data!", 52.4]

         for x in list_of_data:
             if type(x) == str:
                 continue

             # This stuff gets skipped anytime the "continue" is run
             print(x)
```

```
4.4
1.2
6898.32
5289.24
25.1
52.4
```

`break`, on the other hand, literally slams the brakes on a loop, pulling you out one level of indentation immediately.

```
In [27]: import random

         i = 0
         iters = 0
```

```
        while True:
            iters += 1
            i += random.randint(0, 10)
            if i > 1000:
                break

        print(iters)
```

```
211
```

1.2.6 File I/O

Python has a great file I/O library. There are usually third-party libraries that expedite reading certain often-used formats (JSON, XML, binary formats, etc), but you should still be familiar with input/output handles and how they work:

```
In [28]: text_to_write = "I want to save this to a file."
         f = open("some_file.txt", "w")
         f.write(text_to_write)
         f.close()
```

This code writes the string on the first line to a file named `some_file.txt`. We can read it back:

```
In [29]: f = open("some_file.txt", "r")
         from_file = f.read()
         f.close()
         print(from_file)
```

```
I want to save this to a file.
```

Take note what changed: when writing, we used a `"w"` character in the `open` argument, but when reading we used `"r"`. Hopefully this is easy to remember.

Also, when reading/writing *binary* files, you have to include a "b": `"rb"` or `"wb"`.

1.2.7 Functions

A core tenet in writing functions is that **functions should do one thing, and do it well**.

Writing good functions makes code *much* easier to troubleshoot and debug, as the code is already logically separated into components that perform very specific tasks. Thus, if your application is breaking, you usually have a good idea where to start looking.

WARNING: It's very easy to get caught up writing "god functions": one or two massive functions that essentially do everything you need your program to do. But if something breaks, this design is very difficult to debug.

Homework assignments will often require you to break your code into functions so different portions can be autograded.

Functions have a header definition and a body:

```
In [30]: def some_function():  # This line is the header
             pass              # Everything after (that's indented) is the body
```

This function doesn't do anything, but it's perfectly valid. We can call it:

```
In [31]: some_function()
```

Not terribly interesting, but a good outline. To make it interesting, we should add input arguments and return values:

```
In [32]: def vector_magnitude(vector):
             d = 0.0
             for x in vector:
                 d += x ** 2
             return d ** 0.5

In [33]: v1 = [1, 1]
         d1 = vector_magnitude(v1)
         print(d1)
```

```
1.4142135623730951
```

```
In [34]: v2 = [53.3, 13.4]
         d2 = vector_magnitude(v2)
         print(d2)
```

```
54.95862079783298
```

1.2.8 NumPy Arrays

If you looked at our previous `vector_magnitude` function and thought "there must be an easier way to do this", then you were correct: that easier way is NumPy arrays.

NumPy arrays are the result of taking Python lists and adding a ton of back-end C++ code to make them *really* efficient.

Two areas where they excel: *vectorized programming* and *fancy indexing*.

Vectorized programming is perfectly demonstrated with our previous `vector_magnitude` function: since we're performing the same operation on every element of the vector, NumPy allows us to build code that implicitly handles the loop

```
In [35]: import numpy as np

         def vectorized_magnitude(vector):
             return (vector ** 2).sum() ** 0.5

In [36]: v1 = np.array([1, 1])
         d1 = vectorized_magnitude(v1)
         print(d1)
```

```
1.41421356237
```

```
In [37]: v2 = np.array([53.3, 13.4])
         d2 = vectorized_magnitude(v2)
         print(d2)
```

```
54.9586207978
```

We've also seen indexing and slicing before; here, however, NumPy really shines. Let's say we have some super high-dimensional data:

```
In [38]: X = np.random.random((500, 600, 250))
```

We can take statistics of any dimension or slice we want:

```
In [39]: X[:400, 100:200, 0].mean()
```

```
Out[39]: 0.4999548512446948
```

```
In [40]: X[X < 0.01].std()
```

```
Out[40]: 0.0028883487428047415
```

```
In [41]: X[:400, 100:200, 0].mean(axis = 1)
```

```
Out[41]: array([ 0.51867524,  0.51729758,  0.46728925,  0.51712595,  0.48342916,
                 0.49878547,  0.55277146,  0.481717  ,  0.47940307,  0.50020324,
                 0.49276312,  0.47972106,  0.46755515,  0.47643823,  0.52474592,
                 0.49044803,  0.47181944,  0.4414494 ,  0.49613891,  0.5069655 ,
                 0.53519317,  0.48736629,  0.48582798,  0.48059914,  0.49855154,
                 0.50373656,  0.48236145,  0.54419453,  0.53897251,  0.43914593,
                 0.51173516,  0.52868761,  0.51176298,  0.51731556,  0.52547951,
                 0.53169123,  0.52894985,  0.46413039,  0.46841093,  0.48413696,
                 0.51681629,  0.4906845 ,  0.47991217,  0.48928538,  0.51898214,
                 0.46979085,  0.51911555,  0.49553128,  0.5362568 ,  0.48612895,
                 0.47753441,  0.50153655,  0.52788072,  0.52272401,  0.469157  ,
                 0.49238255,  0.47559093,  0.49801046,  0.52056792,  0.51585418,
                 0.469604  ,  0.49038037,  0.55188759,  0.50849742,  0.51667988,
                 0.5342947 ,  0.50331315,  0.44969043,  0.51828591,  0.43520427,
                 0.48280764,  0.47801979,  0.55527671,  0.54282912,  0.47998791,
                 0.53462367,  0.51330914,  0.46526063,  0.48525525,  0.55084526,
                 0.53366459,  0.46832472,  0.5182397 ,  0.46299839,  0.50300485,
                 0.52058732,  0.47696937,  0.43319653,  0.53635848,  0.52606759,
                 0.44742076,  0.53064638,  0.45443005,  0.48121565,  0.489155  ,
                 0.46740624,  0.52551917,  0.49859357,  0.50319623,  0.49630816,
                 0.48928332,  0.49311857,  0.48556844,  0.4731037 ,  0.46989062,
                 0.47058212,  0.5219647 ,  0.52420511,  0.48058429,  0.48308218,
                 0.48383466,  0.47922008,  0.45676722,  0.479294  ,  0.48776054,
```

```
0.51039874,  0.48538941,  0.482676  ,  0.44476912,  0.48895528,
0.50207127,  0.53486804,  0.49224779,  0.5057927 ,  0.50089204,
0.49259557,  0.44914932,  0.47931206,  0.47980468,  0.48720734,
0.49625651,  0.519241  ,  0.48548304,  0.50987037,  0.48421921,
0.48875296,  0.54722896,  0.51664063,  0.47688314,  0.4758808 ,
0.50993576,  0.56979423,  0.53983663,  0.48248539,  0.47952436,
0.46437517,  0.54340922,  0.45194619,  0.45818344,  0.52343475,
0.57352936,  0.53265172,  0.44577159,  0.49278033,  0.52293339,
0.53243724,  0.49470642,  0.51400523,  0.49752551,  0.50414942,
0.52920021,  0.54674973,  0.52547939,  0.50266327,  0.50805356,
0.48169875,  0.55863358,  0.46440341,  0.51636321,  0.45606104,
0.48065807,  0.54691791,  0.51672346,  0.49402188,  0.53044119,
0.48142973,  0.50331395,  0.50838829,  0.53598983,  0.49939437,
0.50970022,  0.50900828,  0.47224788,  0.48400372,  0.54702908,
0.6038528 ,  0.54586053,  0.45575719,  0.47799725,  0.51258097,
0.45549159,  0.54565552,  0.52947205,  0.54935615,  0.51325031,
0.49821494,  0.49455383,  0.52963757,  0.51744372,  0.5432769 ,
0.50843641,  0.49398491,  0.50603989,  0.49051012,  0.46860556,
0.54842711,  0.49851335,  0.50006079,  0.5288276 ,  0.4565634 ,
0.50009507,  0.49696259,  0.55482749,  0.50158645,  0.53907227,
0.55335015,  0.50565859,  0.49504051,  0.47154726,  0.49958723,
0.50459533,  0.4601803 ,  0.52646739,  0.48584911,  0.50740882,
0.52598056,  0.48098476,  0.55930797,  0.49809867,  0.45366715,
0.50646057,  0.46895451,  0.53895226,  0.49611525,  0.51050281,
0.52999852,  0.52002761,  0.46488792,  0.54325472,  0.47775997,
0.55790064,  0.43938541,  0.51835798,  0.50141219,  0.50459811,
0.48380416,  0.53934964,  0.48985159,  0.51079463,  0.47167189,
0.45118765,  0.51404674,  0.50198744,  0.47907865,  0.50424693,
0.50845228,  0.51585481,  0.47079571,  0.44718341,  0.5015922 ,
0.50932904,  0.49724879,  0.44532206,  0.52902114,  0.53207428,
0.45166003,  0.52560117,  0.52039181,  0.49152202,  0.5351256 ,
0.52300715,  0.51161438,  0.500493  ,  0.50991224,  0.49017083,
0.53667429,  0.54666195,  0.50169257,  0.48367601,  0.50505634,
0.51529964,  0.50867079,  0.5012247 ,  0.51413904,  0.48003921,
0.48788705,  0.52335261,  0.48861606,  0.50726088,  0.48286214,
0.49508184,  0.50207396,  0.50320645,  0.52371617,  0.51244911,
0.51063896,  0.451227  ,  0.46406436,  0.51526914,  0.46889669,
0.49210771,  0.50886323,  0.47401079,  0.49884928,  0.47263763,
0.49168413,  0.49949037,  0.52709994,  0.50780375,  0.53584516,
0.51078143,  0.52435315,  0.51844322,  0.51138457,  0.49690471,
0.50502691,  0.49701969,  0.46849427,  0.52716325,  0.49590413,
0.52929819,  0.49757602,  0.49687554,  0.49016215,  0.53120804,
0.50878902,  0.46620647,  0.47068681,  0.48963287,  0.50216615,
0.47197334,  0.47046874,  0.53400009,  0.48733264,  0.45192248,
0.47128717,  0.47485001,  0.51000274,  0.52965288,  0.49683928,
0.45040037,  0.44120822,  0.45106112,  0.52628488,  0.51305737,
0.52952149,  0.53244068,  0.51292314,  0.48635673,  0.46946239,
0.46254022,  0.52060135,  0.49953953,  0.48472247,  0.52934475,
```

```
0.50739332,  0.51218768,  0.48863851,  0.48882609,  0.45735577,
0.54044169,  0.46743286,  0.49637654,  0.44815554,  0.4792592 ,
0.46523942,  0.50832695,  0.47146546,  0.5224416 ,  0.46677961,
0.46138582,  0.55838296,  0.48326638,  0.50875215,  0.50351185,
0.53086144,  0.50254595,  0.51719558,  0.50168972,  0.56349876,
0.48710207,  0.51634076,  0.52657891,  0.4993885 ,  0.47442877,
0.47739874,  0.51438403,  0.50740797,  0.52884893,  0.46762566,
0.48455616,  0.49551488,  0.47547752,  0.48632515,  0.49914437,
0.5058193 ,  0.49056452,  0.49961336,  0.45312081,  0.49036715])
```

1.3 Part 3: Document Classification with Python

We'll end our Python crash-course with a bit of a review from 3360 or your previous intro-to-ML experience: document classification with **Naive Bayes** and **Logistic Regression**.

1.3.1 Bag of words

Hopefully you're familiar with this abstraction for modeling documents.

This model assumes that each word in a document is drawn independently from a multinomial distribution over possible words (a multinomial distribution is a generalization of a Bernoulli distribution to multiple values). Although this model ignores the ordering of words in a document, it works surprisingly well for a number of tasks, including classification.

In short, it says: word *order* doesn't matter nearly as much--or perhaps, at all--as word *frequency*.

1.3.2 Naive Bayes

With any (discriminative) classification problem, you're asking: what's the probability of a label given the data? In our document classification example, this question is: what is the probability of the document class, given the document itself?

Formally, for a document x and label y: $P(y|x)$

If we're using individual word counts as features (x_1 is word 1, x_2 is word 2, and so on), then by the rules of conditional probability, this probability would expand into something like this:

$$P(y|x_1, x_2, ..., x_n) = \frac{P(y)P(x_1, x_2, ..., x_n)}{P(x_1, x_2, ..., x_n)}$$

This is, for all practical purposes, intractable. Hence, "naive": we make each word conditionally independent of the others, *given* the label:

$$P(x_i|y, x_1, x_2, ..., x_i, x_{i+1}, ..., x_n) = P(x_i|y)$$

For any given word x_i then, the original problem reduces to:

$$P(y|x_1, x_2, ..., x_n) = \frac{P(y)\Pi_{i=1}^{n} P(x_i|y)}{P(x_1, x_2, ..., x_n)}$$

And since the denominator is the same across all documents, we can effectively ignore it as a constant, thereby giving us a decision:

$$\hat{y} = \text{argmax}_y P(y)\Pi_{i=1}^{n} P(x_i|y)$$

bow

If you really want to dig into what makes Naive Bayes an improvement over the "optimal Bayes classifier", you can count exactly how many *parameters* are required in either case.

We'll take the simple example: the decision variable Y is boolean, and the observations X have n attributes, each of which is also boolean. Formally, that looks like this:

$$\theta_{ij} = P(X = x_i | Y = y_i)$$

where i takes on 2^n possible values (one for each of the possible combinations of boolean values in the array X, and j takes on 2 possible values (true or false). For any fixed j the sum over i of θ_{ij} has to be 1 (probability). So for any particular y_j, you have the 2^n values of x_i, so you need $2^n - 1$ parameters. Given two possible values for j (since Y is boolean!), we must estimate a total of $2(2^n - 1)$ such θ_{ij} parameters.

This is a problem!

This means that, if our observations X have three attributes--3-dimensional data--we need 14 distinct data points *at least*, one for each possible boolean combination of attributes in X and label Y. It gets exponentially worse as the number of boolean attributes increases--if X has 30 boolean attributes, we'll have to estimate **30 billion parameters.**

This is why the conditional independence assumption of Naive Bayes is so critical: more than anything, it **substantially** reduces the number of required estimated parameters. If, through conditional independence, we have

$$P(X_1, X_2, ..., X_n | Y) = \Pi_{i=1}^{n} P(X_i | Y)$$

or, to illustrate more concretely, observations X with 3 attributes each

$$P(X_1, X_2, X_3 | Y) = P(X_1 | Y) P(X_2 | Y) P(X_3 | Y)$$

we've just gone from requiring the aforementioned 14 parameters, to 6!

Formally: we've gone from requiring $2(2^n - 1)$ parameters to $2n$.

Naive Bayes is a fantastic algorithm and works well in practice. However, it has some important drawbacks to be aware of:

- **Data may not be conditionally independent.** The easiest example of this: replicating a single observation multiple times. These are *clearly* dependent entities, but Naive Bayes will treat them as independent of each other, given the class label. In practice this isn't a common occurrence but can happen.

- **What about continuous attributes?** We've only looked so far at data with boolean attributes; most data, including documents, are not boolean. Rather, they are *continuous*. We can fairly easily modify Naive Bayes to *Gaussian Naive Bayes,* where each attribute is an i.i.d. Gaussian, but this introduces new problems: now we're assuming our data are Gaussian, which like conditional independence, may not be true in practice.

- **Observing data in *testing* that was not observed in *training*.** With the document classification example, training the Naive Bayes parameters essentially consists of word counting. However, what happens when you encounter a word X_i in a test set for which you do not have a corresponding $P(X_i | Y)$? By default, this sets a probability of 0, but this is problematic in a Naive Bayes setting: since you're computing $P(X_1 | Y) P(X_2 | Y) ... P(X_n | Y)$ for a document, a single probability of 0 in that string of multiplication nukes the entire statement!

1.3.3 Logistic Regression

Logistic regression is a bit different. Rather than estimating the parametric form of the data $P(x_i|y)$ and $P(y)$ in order to get to the posterior $P(y|x)$, here we're learning the decision boundary $P(y|x)$ *directly*.

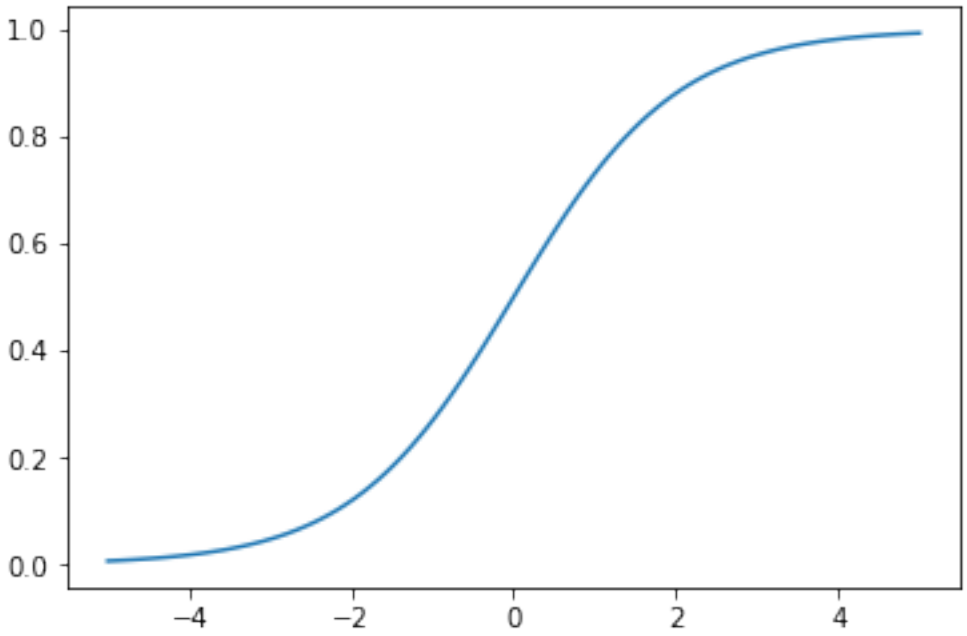
Ideally we want some kind of output function between 0 and 1--so let's just go with the *logit*

```
In [42]: %matplotlib inline
         import matplotlib.pyplot as plt

         x = np.linspace(-5, 5, 100)
         y = 1 / (1 + np.exp(-x))

         plt.plot(x, y)
```

```
Out[42]: [<matplotlib.lines.Line2D at 0x135ccadd8>]
```



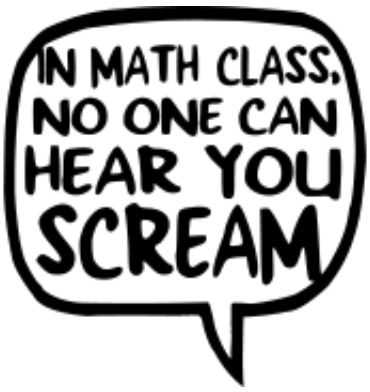
We just adapt the logit function to work our document features x_i, and some weights w_i:

$$P(Y = 0|X) = \frac{1}{1 + \exp(w_0 + \sum_i w_i X_i)}$$

Then finding $P(Y = 1|X)$ is just $1 - P(Y = 0|X)$, or

$$P(Y = 1|X) = \frac{\exp(w_0 + \sum_i w_i X_i)}{1 + \exp(w_0 + \sum_i w_i X_i)}$$

This second equation, for $P(Y = 1|X)$, arises directly from the fact that these two terms must sum to 1. Write it out yourself if you need convincing!



scream

So how do we train a logistic regression model? Here's where things get a tiny bit trickier than Naive Bayes.

In Naive Bayes, the bag-of-words model was 90% of the classifier. Sure, we needed some marginal probabilities and priors, but the word counting was easily the bulk of it.

Here, the word counting is still important, but now we have this entire array of *weights* we didn't have before. These weights correspond to feature relevance--how important the features are to prediction. In Naive Bayes we just kind of assumed that was implicit in the count of the words--higher counts, more relevance. But logistic regression separates these concepts, meaning we now have to learn the weights *on our own*.

We have our training data: $\{(X^{(j)}, y^{(j)})\}_{j=1}^{n}$, and each $X^{(j)} = (x_1^{(j)}, ..., x_d^{(j)})$ for d features/dimensions/words.

And we want to learn: $\hat{\mathbf{w}} = \text{argmax}_{\mathbf{w}} \Pi_{j=1}^{n} P(y^{(j)}|X^{(j)}, \mathbf{w})$

Our conditional log likelihood then takes the form: $l(\mathbf{w}) = \ln \Pi_j P(y^j|\vec{x}^j, \mathbf{w})$

$$= \sum_j \left[y^j (w_0 + \sum_i^d w_i x_i^j) - \ln(1 - \exp(w_0 + \sum_i^d w_i x_i^j)) \right]$$

How did we get here?

First, note that the likelihood function is typically formally denoted as

$$W \leftarrow \arg\max_W \Pi_l P(Y^l|X^l, W)$$

for each training example X^l with corresponding ground-truth label Y^l (they are multipled together because we assume each observation is *independent* of the other). We include the weights W in this expression because the probability is absolutely a function of the weights, and we want to pick the combination of weights W that make the probability expression *as large as possible*.

Second, because we're both pragmatic enough to use a short-cut whenever we can and evil enough to know it'll confuse other people, we never actually work directly with the likelihood as stated above. Instead, we work with the *log*-likelihood, by literally taking the log of the function:

$$W \leftarrow \arg\max_W \sum_l \ln P(Y^l|X^l, W)$$

Recall that the log of a product is equivalent to the sum of logs.

Third, the probability statement $P(Y^l|X^l, W)$ has two main terms, since Y can be either 1 or 0; we want to pick the one with the largest probability. So we expand that term into the following:

$$l(W) = \sum_l Y^l \ln P(Y^l = 1|X^l, W) + (1 - Y^l) \ln P(Y^l = 0|X^l, W)$$

where $l(W)$ is our log-likehood function.

Hopefully this looks somewhat familiar to you: it's a lot like finding the expected value $E[X]$ of a discrete random variable X, where you take each possible value $X = x$ and multiply it by its probability $P(X = x)$, summing them all together. You can see the case $Y = 1$ on the left, and $Y = 0$ on the right, both being multiplied by their corresponding conditional probabilities.

Hopefully you'll *also* note: since you're using this equation for training, Y^l will take ONLY 1 or 0, therefore zero-ing out one side of the equation or the other for every single training instance. So that's kinda nice?

Fourth, get ready for some math! If we have

$$l(W) = \sum_l Y^l \ln P(Y^l = 1|X^l, W) + (1 - Y^l) \ln P(Y^l = 0|X^l, W)$$

Expand the last term:

$$l(W) = \sum_l Y^l \ln P(Y^l = 1|X^l, W) + \ln P(Y^l = 0|X^l, W) - Y^l \ln P(Y^l = 0|X^l, W)$$

Combine terms with the same Y^l coefficient (first and third terms):

$$l(W) = \sum_l Y^l \left[\ln P(Y^l = 1|X^l, W) - \ln P(Y^l = 0|X^l, W)\right] + \ln P(Y^l = 0|X^l, W)$$

Recall properties of logarithms--when subtracting two logs with the same base, you can combine their arguments into a single log dividing the two:

$$l(W) = \sum_l Y^l \left[\ln \frac{P(Y^l = 1|X^l, W)}{P(Y^l = 0|X^l, W)}\right] + \ln P(Y^l = 0|X^l, W)$$

Now things get interesting--remember earlier where we defined exact parametric forms of $P(Y = 1|X)$ and $P(Y = 0|X)$? Substitute those back in, and you'll get:

$$l(W) = \sum_l \left[Y^l (w_0 + \sum_i^d w_i X_i^l) - \ln(1 - \exp(w_0 + \sum_i^d w_i X_i^l))\right]$$

which is exactly the equation we had before we started going through these proofs.

Good news! $l(\mathbf{w})$ is a concave function of $\mathbf{w}$, meaning no pesky local optima.

Bad news! No closed-form version of $l(\mathbf{w})$ to find explicit values (feel free to try and take its derivative, set it to 0, and solve; it's a transcendental function, so it has no closed-form solution).

Good news! Concave (convex) functions are easy to optimize!

Maximum of a concave function = minimum of a convex function

- Gradient ascent (concave) = gradient descent (convex)

Gradient ascent algorithm: iterate until change < ε

$$w_0^{(t+1)} \leftarrow w_0^{(t)} + \eta \sum_j [y^j - \hat{P}(Y^j = 1 \mid \mathbf{x}^j, \mathbf{w}^{(t)})]$$

For i=1,...,d,

$$w_i^{(t+1)} \leftarrow w_i^{(t)} + \eta \sum_j x_i^j [y^j - \hat{P}(Y^j = 1 \mid \mathbf{x}^j, \mathbf{w}^{(t)})]$$

Predict what current weight thinks label Y should be

repeat

lr_grad

Gradient:

$$\nabla_{\mathbf{w}} l(\mathbf{w}) = \left[\frac{\partial l(\mathbf{w})}{\partial w_0}, ..., \frac{\partial l(\mathbf{w})}{\partial w_n} \right]$$

Update rule: $w_i^{(t+1)} = w_i^{(t)} + \eta \frac{\partial l(\mathbf{w})}{\partial w_i}$

Which ultimately leads us to **gradient ascent for logistic regression**.

This is Assignment 1!

In addition to going over some basic concepts in probability, Naive Bayes, and Logistic Regression, you'll also implement some document classification code from scratch (don't let me catch anyone using scikit-learn, mmk).

The hardest part in the coding will be implementing gradient descent! It's not a lot of code--**especially if you use NumPy vectorized programming**--but it will take some sitting-and-thinking-and-whiteboarding time (unless you know this stuff cold already, I suppose)!

There is also some theory and small proofs.

Don't be intimidated. I purposely made this homework tricky both to get an idea of your level of understanding of the topics so I can gauge how to proceed in the course, and also so you have an idea where your weaknesses are.

ASK ME FOR HELP! Helping students is *literally* my day job. Don't be shy; if you're stuck, reach out for help, both from me AND your student colleagues!

1.4 Administrivia

- **Assignment 1 is out on AutoLab.** This is a warm-up to familiarize you with Python, AutoLab, and to make sure you're up to speed on the basics of machine learning and probability. You'll be implementing Multinomial Naive Bayes and Logistic Regression *from first principles*. Should be fun! Assignment 1 is due *Thursday, August 31 by 11:59pm.*

- The first Workshop is *Wednesday, August 30.* If you have not yet signed up for workshops, please do so here: https://docs.google.com/spreadsheets/d/1SmEhL30kBvjPa6WioR34zM8HP5HuY38N3vGzQEwGVS0/e (Just FYI: after this weekend, I will *auto-assign* anyone who has not yet signed up).

- I will be out of town **all next week and part of the week after.** I agree it's bad timing, but I will be in touch by Slack and email. In the meantime, we'll have guest lecturers discussing their work in different areas of data science, so please be sure to attend if you're at all interested in seeing what different forms data science can take!