DIGITAL TWIN AI and Machine Learning: Introduction and Python Crash Course

Prof. Andrew D. Bagdanov
andrew.bagdanov AT unifi.it

Dipartimento di Ingegneria dell'Informazione
Università degli Studi di Firenze

9 January 2020

Outline

Introduction

Lecture presentations

- You can will find all class lecture presentations at this site:

  http://micc.unifi.it/bagdanov/digitaltwin/

- Published here will also be links to Colaboratory Notebooks and any supplementary material.

Defining Artificial Intelligence

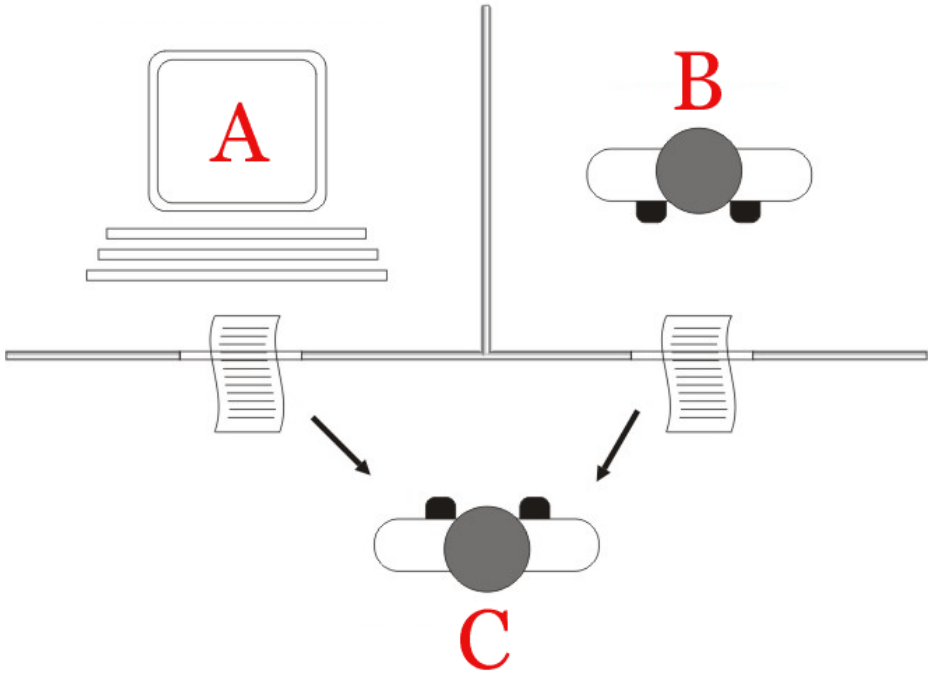
- Thinking Humanly:
  "The study of mental faculties through the use of computational models." – Charniak+McDermott, 1985.
- Thinking Rationally:
  "The branch of computer science that is concerned with the automation of intelligent behavior." – Luger+Stubblefield, 1993.
- Acting Humanly:
  "The study of how to make computers do things at which, at the moment, people are better." – Rich+Knight, 1991.
- Acting Rationally:
  "[The automation of] activities that we associate with human thinking, activities such as decision-making, problem solving, learning. . . " – Bellman, 1978.

Acting Humanly: The Turing Test

- Turing (1950): Computing machinery and intelligence:

"Can machines think?" $\longrightarrow$ "Can machines behave intelligently?"

- Operational test for intelligent behavior: the Imitation Game:



Acting Humanly: The Turing Test

- Turing predicted that by 2000, a machine might have a 30% chance of fooling a lay person for 5 minutes.
- Anticipated all major arguments against AI for 50 years.
- Suggested major components of AI: knowledge, reasoning, language understanding, learning.

Problem: Turing test is not reproducible, constructive, or amenable to mathematical analysis.

"It seems probable that once the machine thinking method had started, it would not take long to outstrip our feeble powers. They would be able to converse with each other to sharpen their wits. At some stage therefore, we should have to expect the machines to take control." – Alan Turing, 1951.

Thinking humanly: Cognitive Science

- 1960s cognitive revolution: information-processing psychology replaced prevailing orthodoxy of behaviorism.
- Requires scientific theories of internal activities of the brain:
  - What level of abstraction? "Knowledge" or "circuits"?
  - How to validate? Requires:
    1. Predicting and testing behavior of human subjects (top-down); or
    2. Direct identification from neurological data (bottom-up).
- Both approaches (roughly, Cognitive Science and Cognitive Neuroscience) are now distinct from AI.
- Steven Pinker on Cognitive Science

Thinking rationally: Laws of Thought

Normative (or prescriptive) rather than descriptive:

- Aristotle: what are correct arguments/thought processes?
- Several Greek schools developed various forms of logic: notation and rules of derivation for thoughts.
- This may or may not have proceeded to the idea of mechanization.
- Direct line through mathematics and philosophy to modern AI.

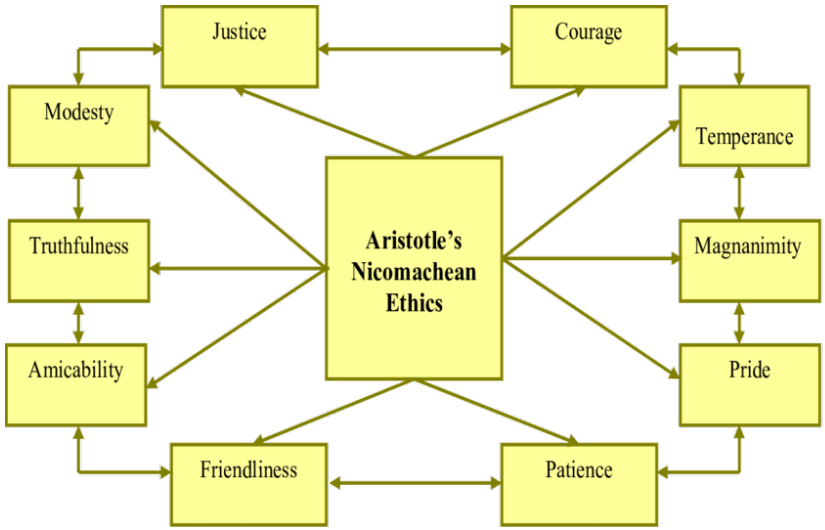
Problems:

1. Not all intelligent behavior is mediated by logical deliberation.
2. What is the purpose of thinking? What thoughts should I have?

Acting rationally

Rational behavior: doing the right thing.

- The right thing: that which is expected to maximize goal achievement, given the available information.
- Doesn't necessarily involve thinking – e.g., blinking reflex – but thinking should be in the service of rational action.
- Aristotle (Nicomachean Ethics): *Every art and every inquiry, and similarly every action and pursuit, is thought to aim at some good.*



Source: Using Accounting Reform to Stimulate Sustainability Practices in Higher Education, 2011.

A first formalism: Rational Agents

Definition (Rational Agents)

An agent is an entity that perceives and acts. This course is about designing rational agents. Abstractly, an agent is a function from percept histories to actions:

$$f : \mathcal{P}^* \to \mathcal{A}$$

For any given class of environments and tasks, we seek the agent (or class of agents) with the best performance.

Caveat: computational limitations make perfect rationality unachievable; so we design best program for given machine resources.

The devil is in the details

- This mathematical formalism doesn't even hint at a recipe for actually building artificially intelligent systems.
- For now we will use an operational definition of AI

Definition (Artificial Intelligence)

Artificial Intelligence refers to the design and implementation of algorithms, applications, and systems that perform tasks normally thought to require *human intelligence* to perform.

Some History

Classical roots of AI

- Philosophy: logic, methods of reasoning, mind as physical system, foundations of learning, language, rationality.
- Mathematics: formal representation and proof, algorithms, computation, (un)decidability, (in)tractability, probability.
- Psychology: adaptation, phenomena of perception and motor control, experimental techniques (psychophysics, etc).
- Linguistics: knowledge representation, grammars.
- Neuroscience: physical substrate for mental activity.
- Control theory: homeostatic systems, stability, simple optimal agent designs.

A prehistoric timeline

1943	McCulloch & Pitts: Boolean circuit model of brain.
1950	Turing's "Computing Machinery and Intelligence."
1952–69	Look, Ma, no hands!
1950s	Early AI programs, including Samuel's checkers program, Newell & Simon's Logic Theorist, Gelernter's Geometry Engine.
1956	Dartmouth meeting: "Artificial Intelligence" adopted.
1965	Robinson's complete algorithm for logical reasoning.
1966–74	AI discovers computational complexity. Neural network research almost disappears.
1969–79	Early development of knowledge-based systems.
1980–88	Expert systems industry booms.
1988–93	Expert systems industry busts: "AI Winter."
1985–95	Neural networks return to popularity.
1988–	Resurgence of probabilistic and decision-theoretic methods. Rapid increase in technical depth of mainstream AI. "Nouvelle AI": ALife, GAs, soft computing.

Early successes

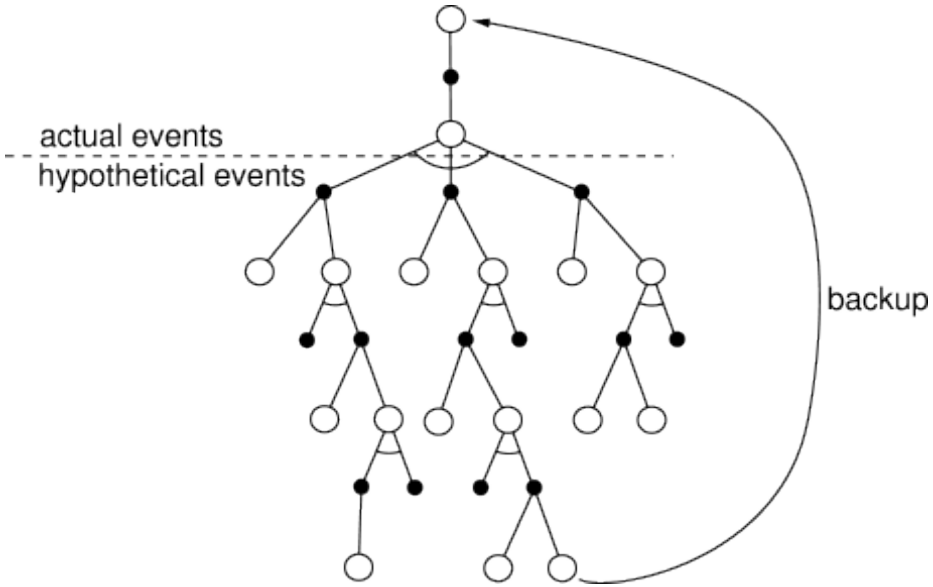
Game playing

- Some of the earliest work in applied AI involved games.
- Arthur Samuel built a system to play checkers in the mid-1950s
- This seminal work defined much of the foundations for classical, search-based AI.
- Game AI has continued to be a benchmark for progress in AI.

Early successes (continued)

Theorem proving

- In 1955, Allen Newell and Herbert A. Simon created the Logic Theorist.
- The program would eventually prove 38 of the first 52 theorems in Russell and Whitehead's *Principia Mathematica*
- It would even find *new and more elegant proofs* for some.



Early optimism

The Dartmouth Workshop

- The Dartmouth Summer Research Project on Artificial Intelligence was a 1956 summer workshop widely considered to be the founding event of artificial intelligence as a field.
- The workshop hosted then (and soon to be) luminaries of the field: Marvin Minsky, John McCarthy, Claude Shannon, Oliver Selfridge, Allen Newell, Herbert Simon, John Nash.
- The organizers thought the general question of artificial intelligence could be resolved (or at least significant progress made on it) over the course of one summer.

Early optimism (continued)

"We propose that a 2-month, 10-man study of artificial intelligence be carried out during the summer of 1956 at Dartmouth College. The study is to proceed on the basis of the conjecture that every aspect of learning or any other feature of intelligence can in principle be so precisely described that a machine can be made to simulate it. An attempt will be made to find how to make machines use language, form abstractions and concepts, solve kinds of problems now reserved for humans, and improve themselves."

Insurmountable problems

Combinatorial explosion

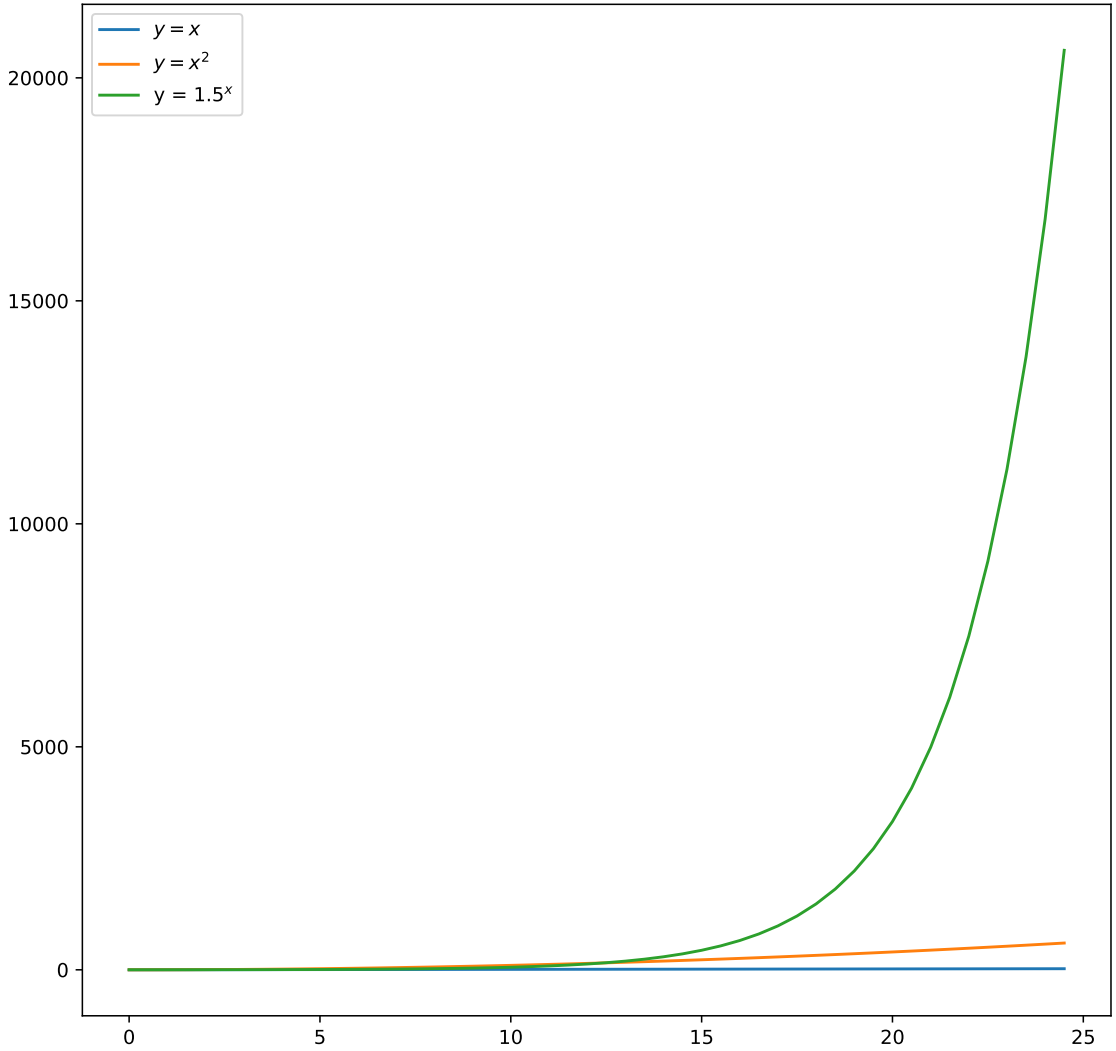
- Most early approaches to AI were based on searching in configuration spaces.
- Even theorem proving is a (very general) type of search in deduction space.
- AI programs could play checkers and prove theorems with relatively few inference steps.
- But going beyond these early successes proved extremely difficult due to the exponential nature of the search space.

Lack of "common knowledge"

- Another difficulty was that solving many types of problems (e.g. recognizing faces or navigating cluttered environments) requires a surprising amount of background knowledge.
- Researchers soon discovered that this was a truly vast amount of information.
- No one in 1970 could build a database so large and no one knew how a program might learn so much information.

Insurmountable problems (continued)

- We often don't appreciate what exponential growth really means:



Insurmountable problems (continued)

Limited computing power

- There was not enough memory or processing speed to do anything truly useful.
- In 2011, computer vision applications required 10,000 to 1,000,000 MIPS.
- By comparison, the fastest supercomputer in 1976, the Cray-1 (retailing at $5 million to $8 million), was only capable of around 80 to 130 MIPS, and a typical desktop computer less than 1 MIPS.

Infighting

- The perceptron neural network was introduced in 1958 by Frank Rosenblatt.
- There was an active research program throughout the 1960s but it came to a sudden halt with the publication of Minsky and Papert's 1969 book Perceptrons.
- It suggested that there were severe limitations to what perceptrons could do and that Rosenblatt's predictions had been grossly exaggerated.
- The effect of the book was devastating: *virtually no research at all was done in connectionism for 10 years*.

The 80s Boom

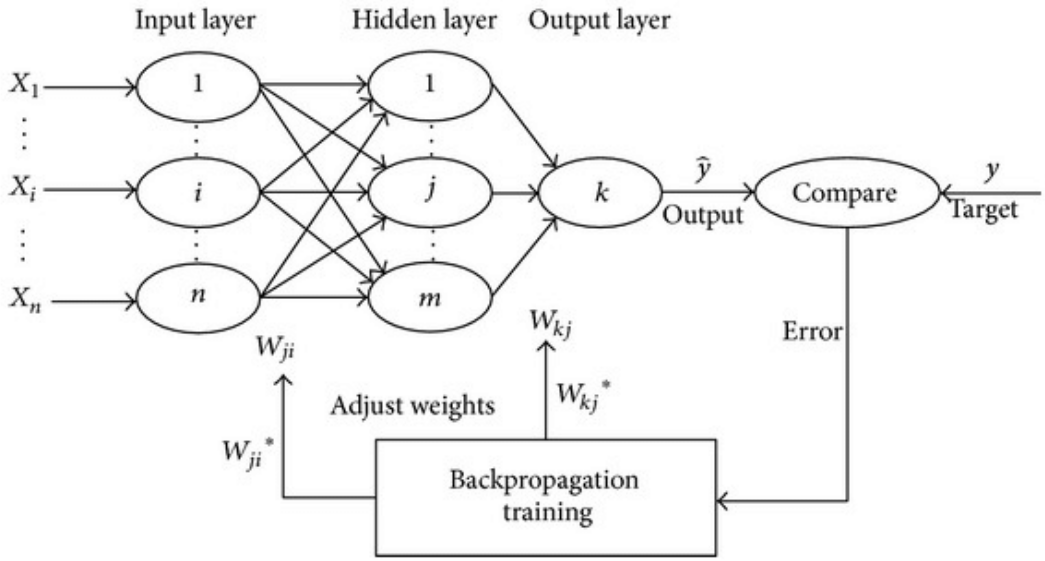
Expert systems

- In the 1980s a form of AI program called an expert system was adopted by corporations around the world.
- Knowledge became the focus of mainstream AI research.
- An expert system is a program that answers questions or solves problems about a specific domain of knowledge and logical rules derived from experts.
- They were part of a new direction in AI research that had been gaining ground throughout the 70s.
- AI researchers were beginning to suspect that intelligence might very well be based on the ability to use large amounts of diverse knowledge in different ways.
- Expert systems restricted themselves to a small domain of specific knowledge (thus avoiding the commonsense knowledge problem).
- All in all, the programs proved to be useful: *something that AI had not been able to achieve up to this point.*

The 80s Boom (continued)

The connectionist revival

- In 1982, John Hopfield proved that a form of neural network (now called a Hopfield net) could learn and process information.
- Around the same time, Geoffrey Hinton and David Rumelhart popularized a method for training neural networks called backpropagation.
- Neural networks would become commercially successful in the 1990s for OCR and speech recognition.



The Bubble Phenomenon

Expansion and crash

- The business community's fascination with AI rose and fell in the 1980s in the classic pattern of an economic bubble.
- The collapse was in the perception of AI by government agencies and investors – the field continued to make advances despite the criticism.
- The first indication of a crash was the sudden collapse of the market for specialized AI hardware in 1987.
- Desktop computers from Apple and IBM were gaining speed and power and were soon more powerful than the more expensive Lisp machines made by Symbolics and others.
- There was no longer a good reason to buy them, and an entire industry worth half a billion dollars was demolished overnight.

Not all bad news

Follow the money (or not)

- In the late 1980s most public funding for AI dried up.
- Despite this, the true believers continued to make steady theoretical and applied progress.
- People like Jürgen Schmidhuber, Yann LeCun, Geoff Hinton, and Yoshua Bengio make significant progress during the Second AI Winter.
- As the community came to grips with the fact that expert systems don't scale very well and are expensive to maintain, neural networks resurfaced as a viable contender for the way forward.
- In particular, the first viable Convolutional Neural Networks (CNNs) were demonstrated and the backpropagation algorithm was proven scalable (and controllable).

The Right Place and Time

Deep learning

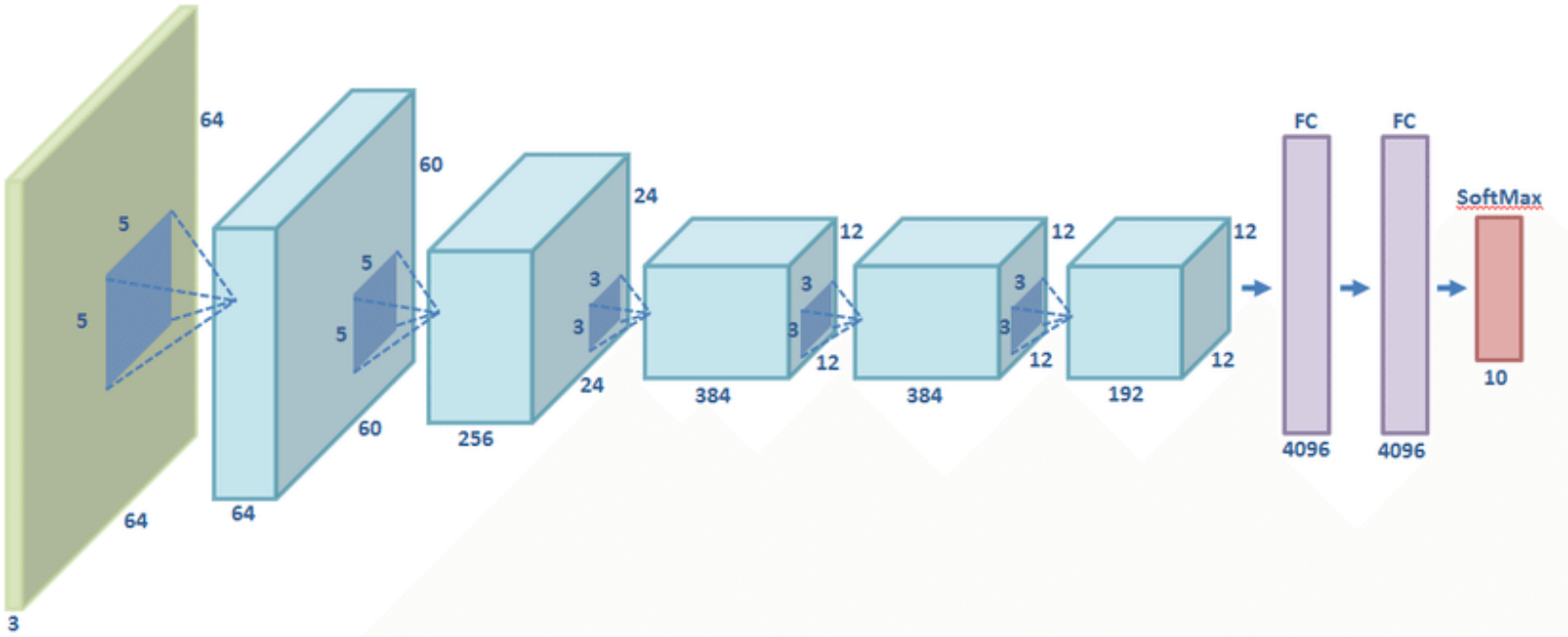
- Deep Learning – compositional, multi-layer neural networks – has been around since the 1970s.
- However, we did not understand how to effectively learn the vast number of parameters they have from data.
- And, the computers of the day were just not up to the challenge of fitting these models.

ImageNet and GPUs

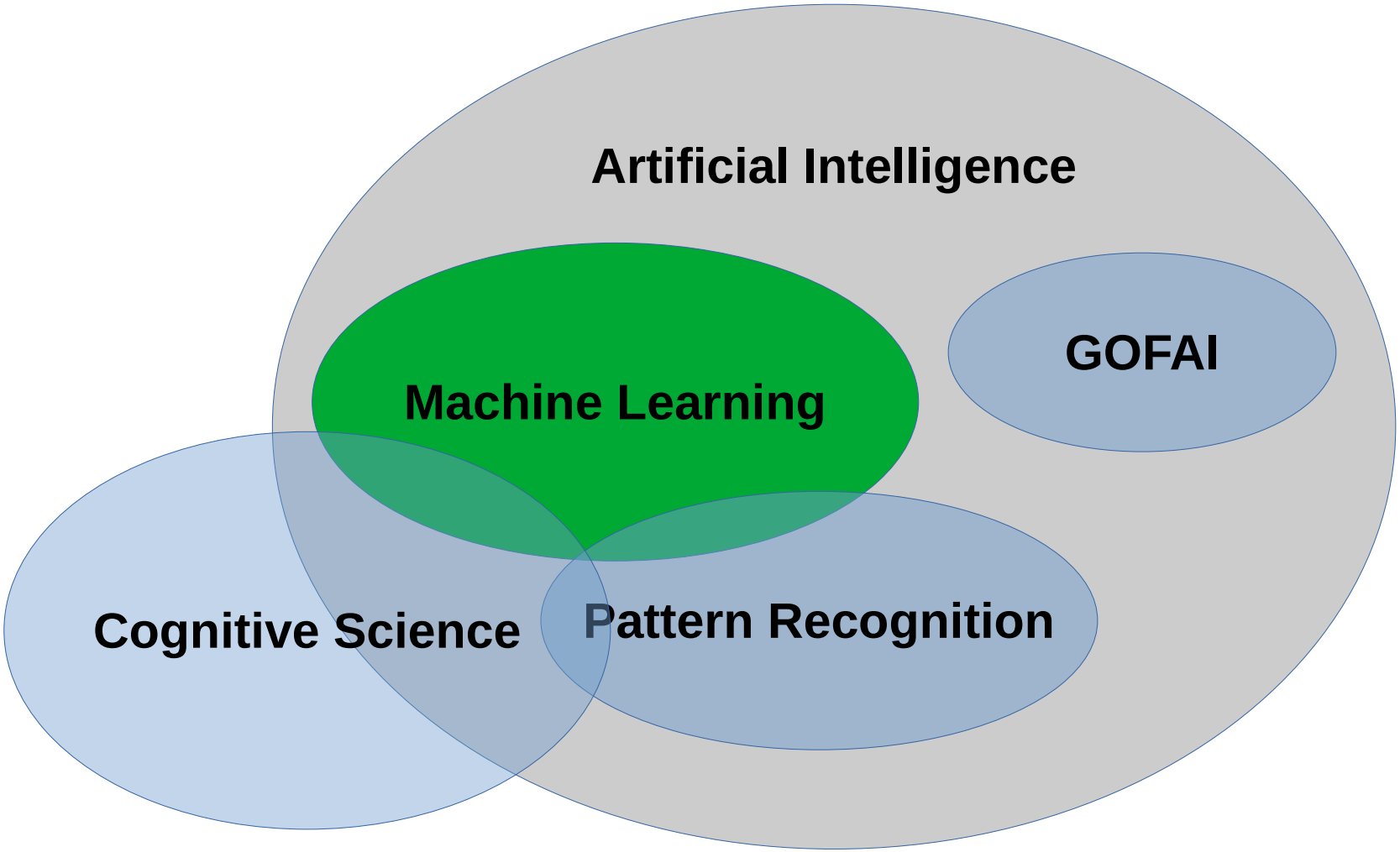
- In the early 2010s, however, the confluence of several technological and theoretical factors combined.
- ImageNet: the availability of massive amounts of labeled data made deep learning feasible (in principle).
- GPUs: Graphics Processing Units addressed some of the computational issues related to training deep models.

AlexNet

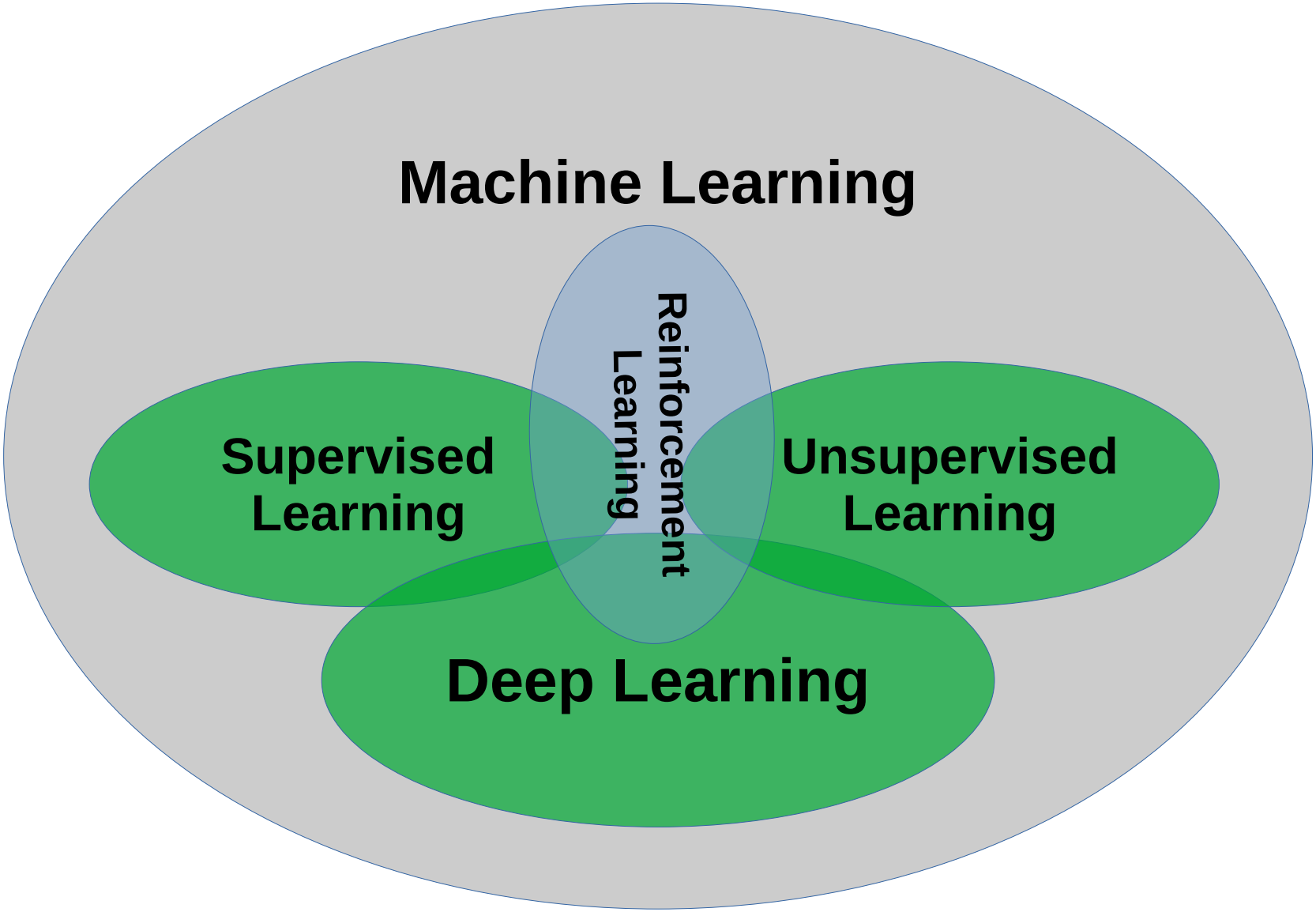
- In 2012 the AlexNet Deep Convolutional Network made history.
- *Everything had changed*: AlexNet surpassed the current state-of-the-art by nearly 20%.



A global view



A local view (this course)



Course overview

Something for everyone

- This course is designed for a broad and diverse audience.
- Some mathematical background is assumed, as well as some exposure to high-level programming (e.g. Python, Java, C/C++, Lisp, heck even Visual Basic).
- [INFORMAL SURVEY]

For the curious

- So everyone is talking about deep learning and General Artificial Intelligence.
- But what is all the hype about?
- This course will give you a broad overview (without the hype) of the fundamental concepts surrounding these recent advances.
- Because it's not all hype. There is an ongoing and sweeping sea change happening today.
- Artificial intelligence is already having significant impact in manufacturing, advertising, smart city management, transportation, entertainment, you name it.
- For the curious: this course should provide you the grounding in essential concepts needed to interpret, understand, and exploit these new developments.

For the studious practitioner

- So everyone is talking about deep learning and General Artificial Intelligence.
- For those of you actually working daily with large (or even massive) amounts of data, what does this mean for you?
- This course will give you hands-on experience with the tools, models, and frameworks used today.
- You will learn how to manipulate data and extrapolate models from it that generalize well to unseen data.

For everyone: a sign of the times...

- So everyone is talking about deep learning and General Artificial Intelligence.
- Articles and whole courses are appearing daily with tantalizing titles like "Deep Learning Zero to Hero" and "How to learn Deep Learning in One Week!!1!".
- It has become difficult to *sift the wheat from the chaff* because the signal-to-noise ratio is becoming infinitesimal.
- For everyone: this course should ground you in the fundamental concepts and tools needed to discern charlatans from quality sources and to make sense of AI and deep learning advances.

Math and Programming

- History and Python Crash Course (today!):
  - We have already contextualized the modern AI explosion in the history of the field.
  - In the second part of the lecture we will have a brief Python crash course to introduce the programming language used for examples, exercises and labs.
- Mathematical Foundations (tomorrow):
  - Linear algebra has been called the mathematics of the 21st century – and it is essential to understanding all of the models, techniques, and algorithms we will see.
  - Calculus is also central to how we actually learn from data and we will see (from a high level) how learning can be formulated as a optimization problem.

Working with Tsunamis of Data

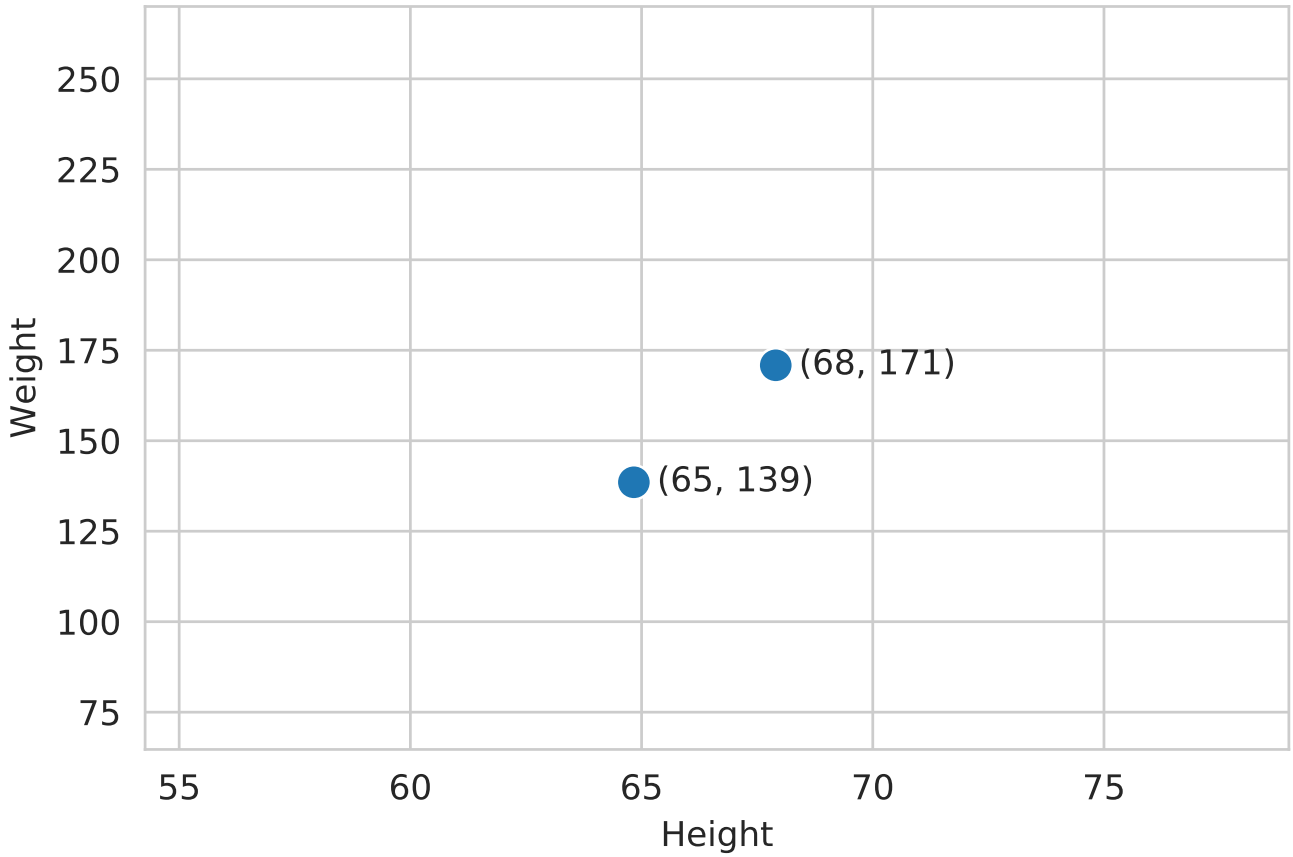
- Visualization:
  - Central to working with massive amounts of data is effective visualization of high dimensional data.
  - We will use techniques like histograms, scatter plots, t-SNE, and others to monitor learning progress and to summarize results.
- Data Science:
  - Also central to working with Big Data is the need to manage data in flexible and abstract ways.
  - We will use Jupyter notebooks (in the form of Google Colaboratory) to organize, document, and guarantee reproducibility.
  - We will use the Pandas library to perform data analysis, to manage data and datasets, and to prepare data for our experiments.

Supervised learning

- Let's say are analyzing the correlation between height and weight.
- (Aside: we will often use synthetic examples of this type to illustrate key concepts and techniques.)
- And let's say that we have only two data points:
$$(67.9, 170.85) \text{ and } (61.9, 122.5).$$
- Ideally, we wish to infer a relation between height and weight that explains the data.
- A good first step us usually to visualize.

Supervised learning (continued)

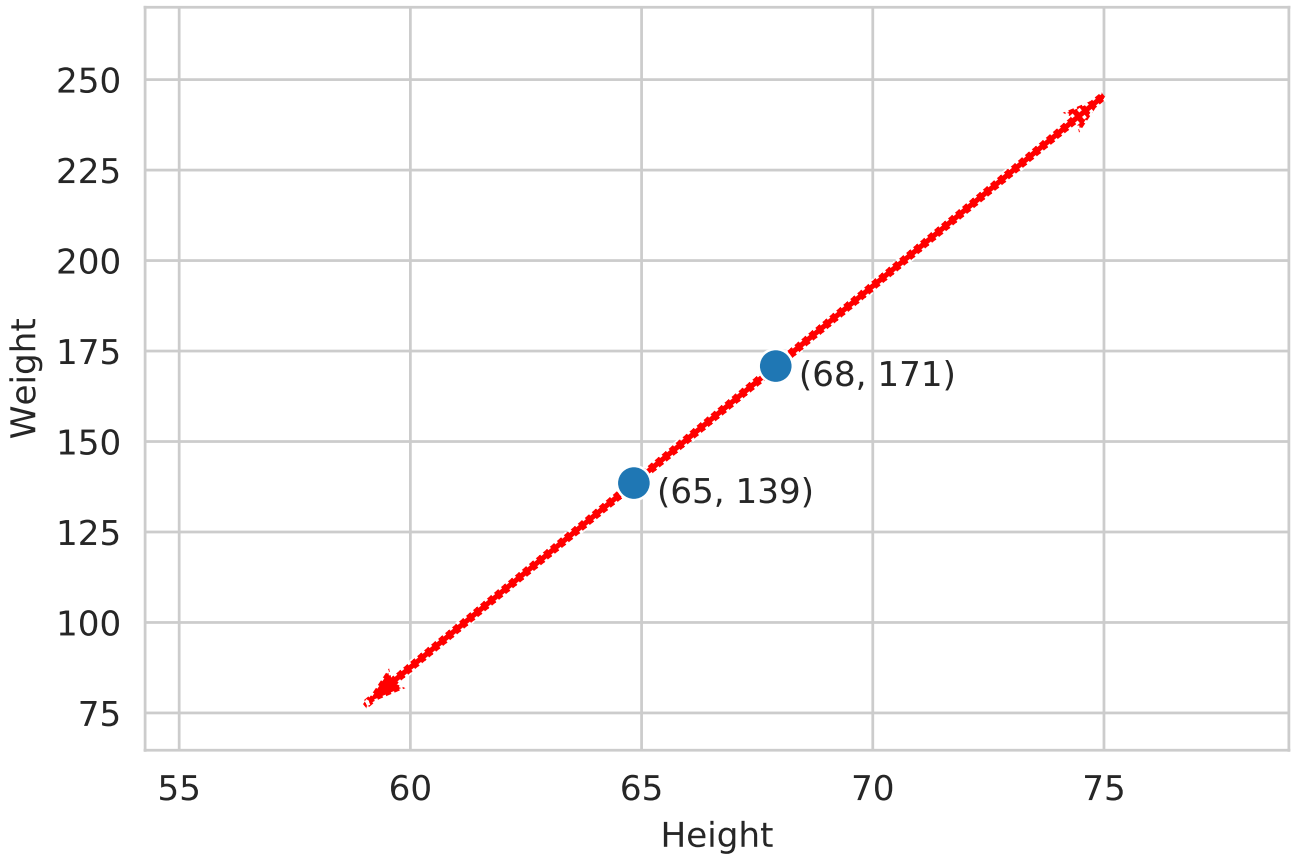
- So, we have a situation like this. . .
- What can we do?



Supervised learning (continued)

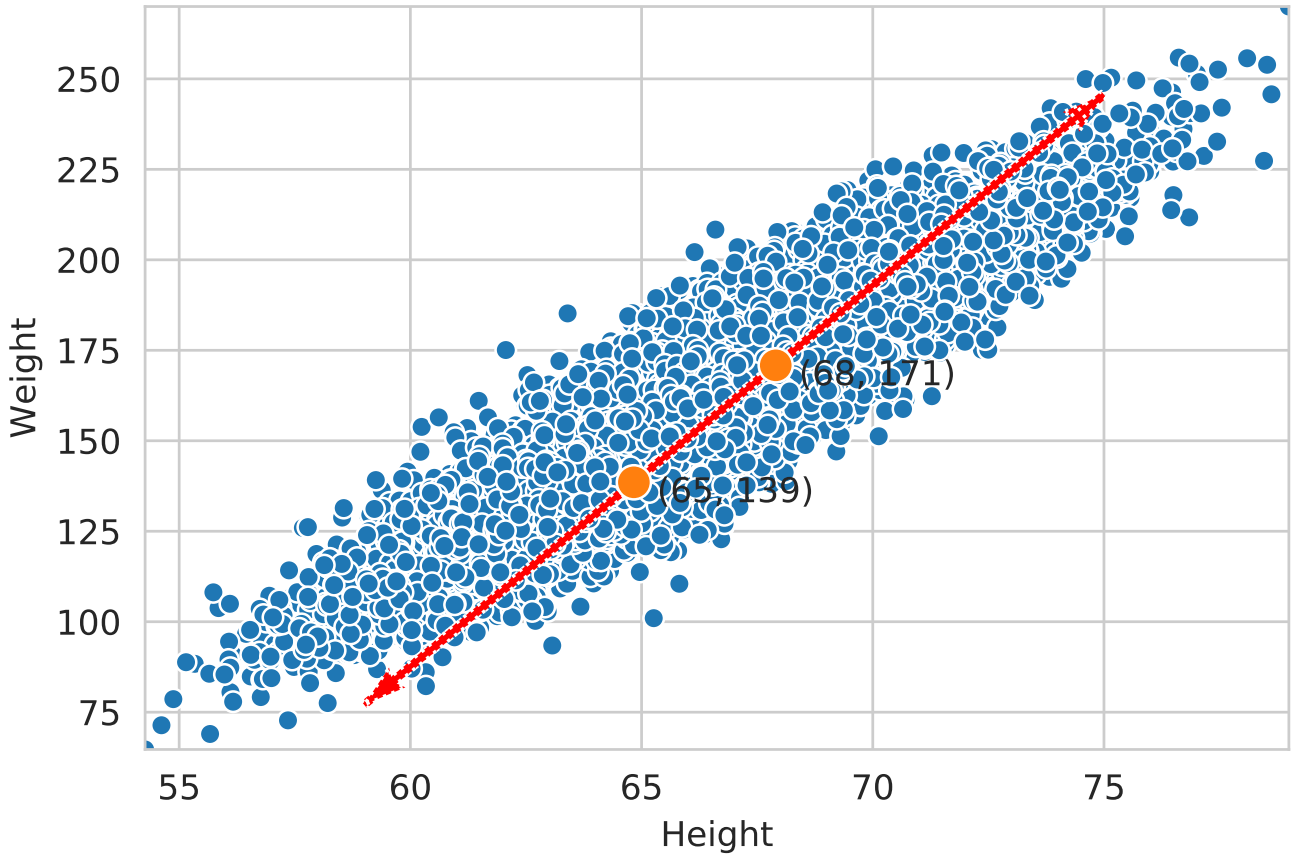
- Well, some grade-school algebra lets us connect the dots:

$$y = 8.013x - 373.247$$



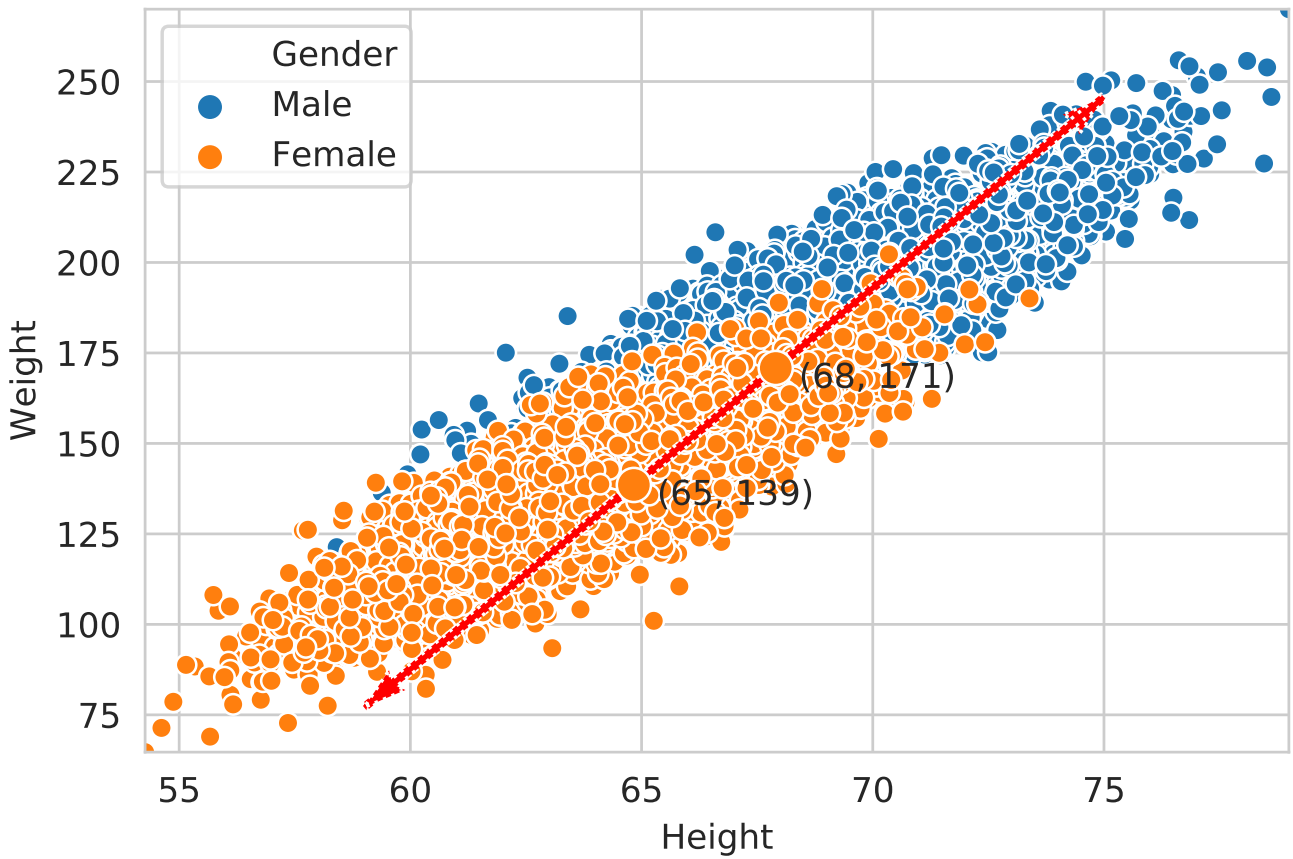
Supervised learning (continued)

- Now lets say that we have a lot more data.
- Does our "model" generalize?



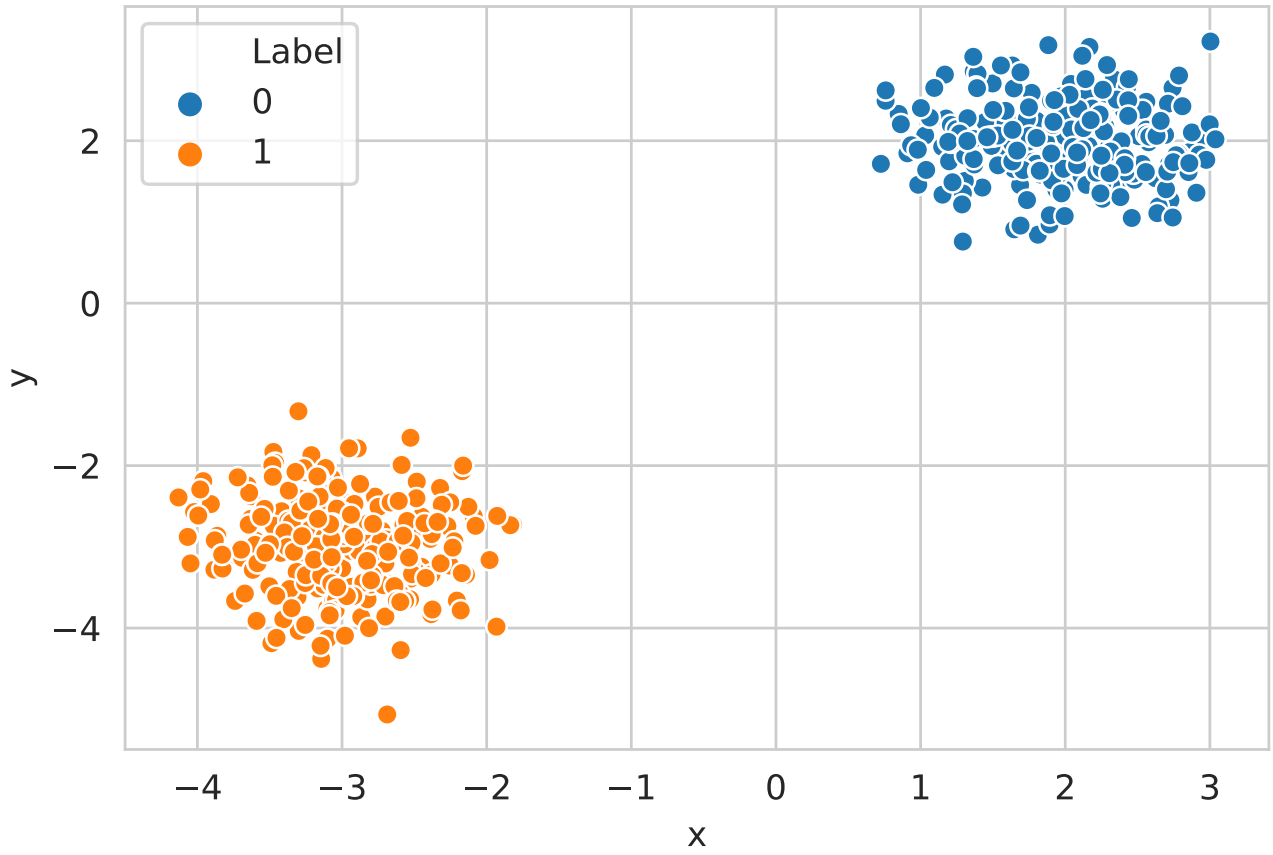
Supervised learning (continued)

- Scratching the surface a bit more, we discover not a single distribution, but rather two.



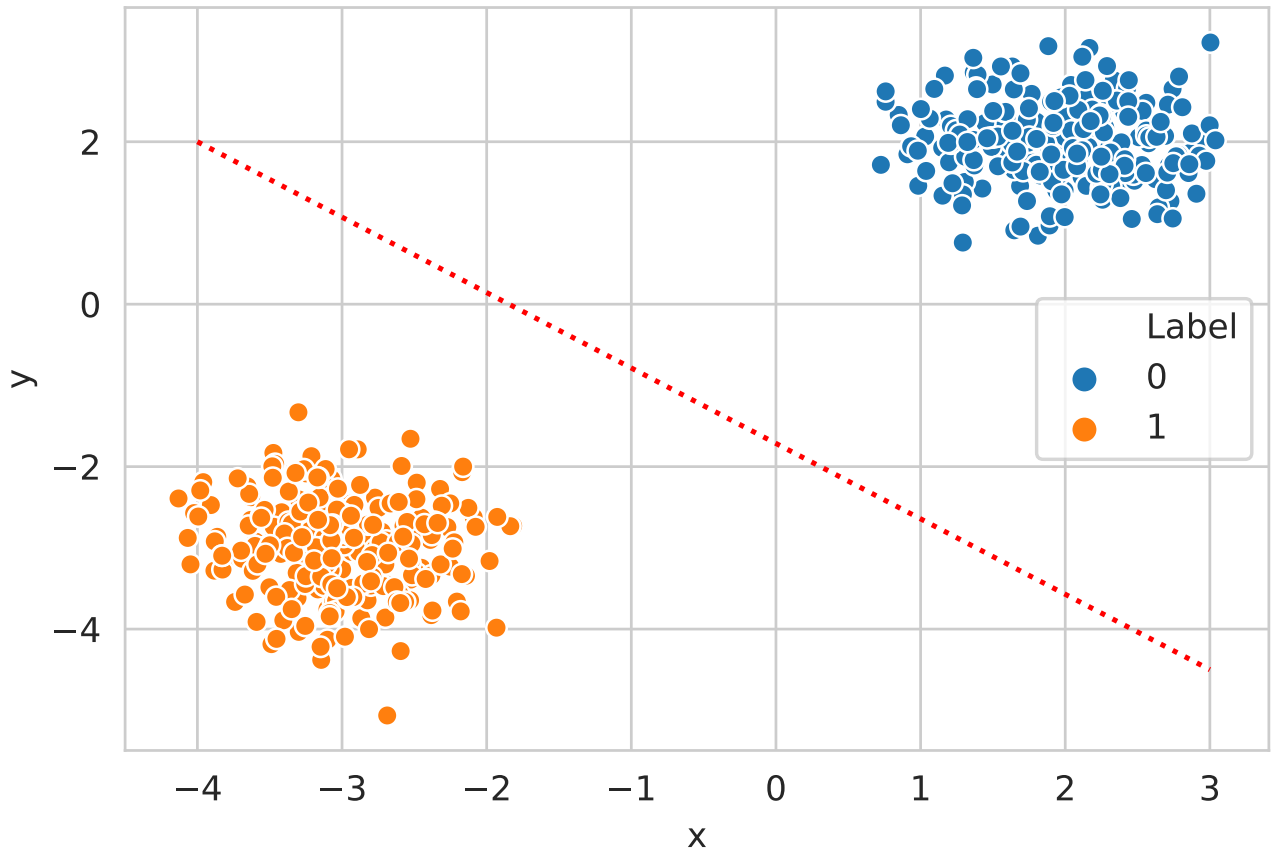
Supervised learning (continued)

- What if our goal is to classify samples into one of two classes?
- We must infer a decision boundary that generalizes to new data.



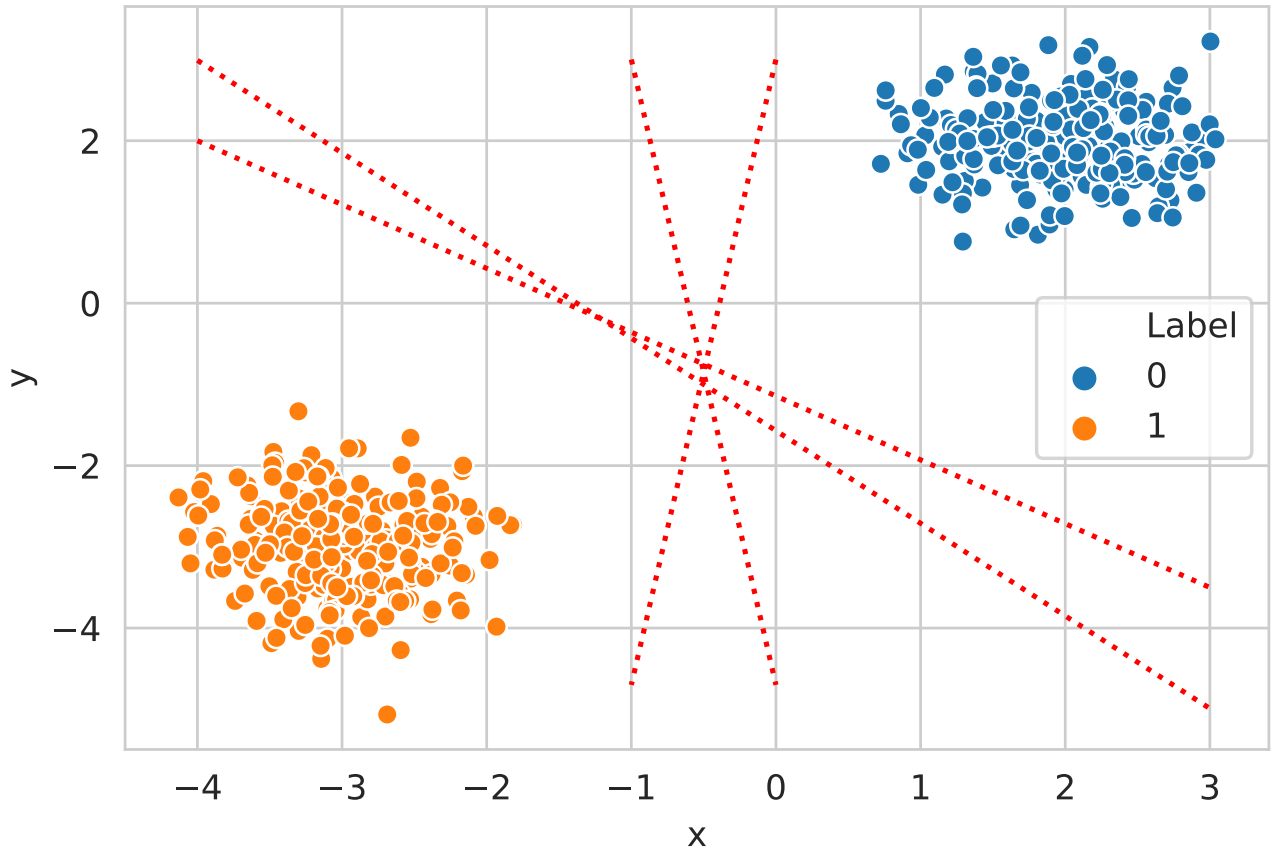
Supervised learning (continued)

- OK, that seems simple.
- But, why should we prefer one solution over another? Or another?



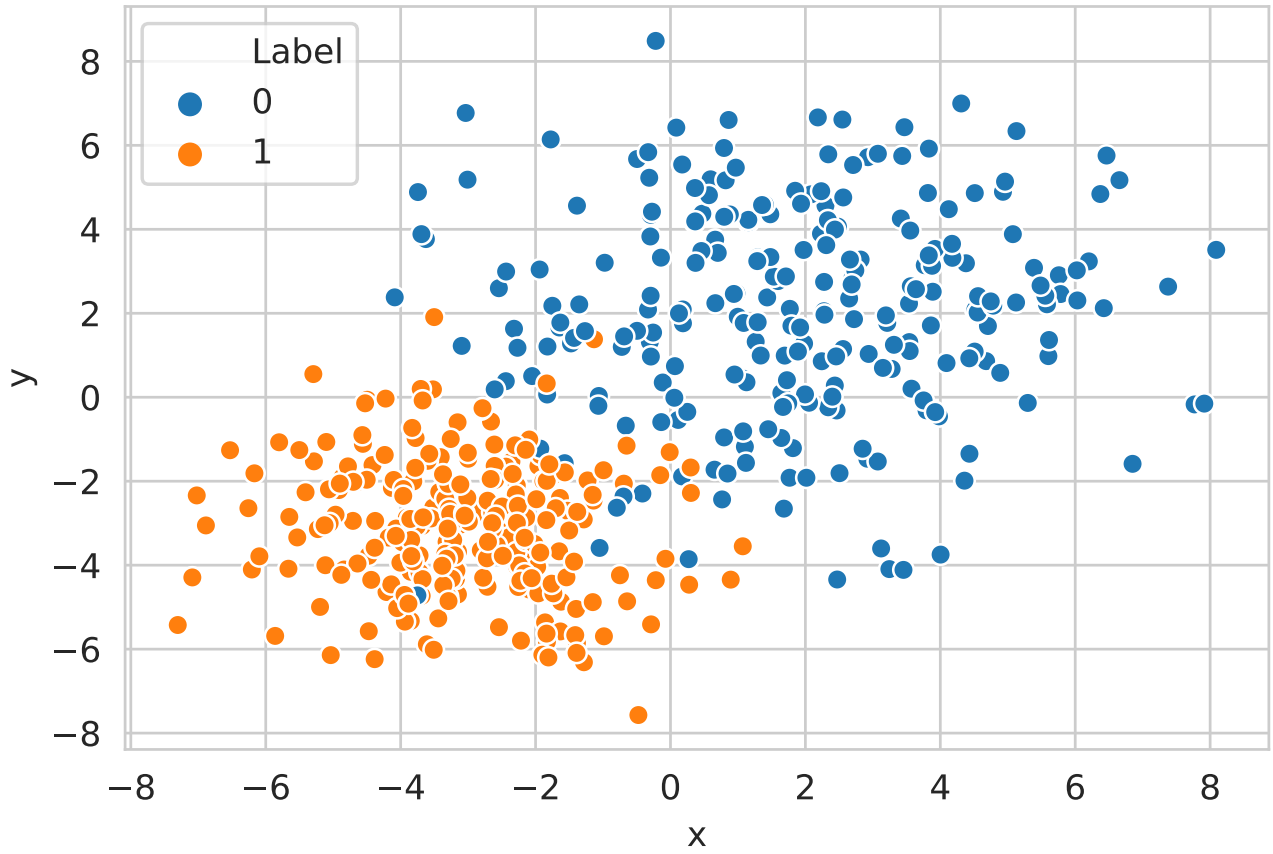
Supervised learning (continued)

- OK, that seems simple.
- But, why should we prefer one solution over another? Or another?



Supervised learning (continued)

- And what the heck do we do here?
- And how should this look in more than two dimensions?



Supervised learning: take home message

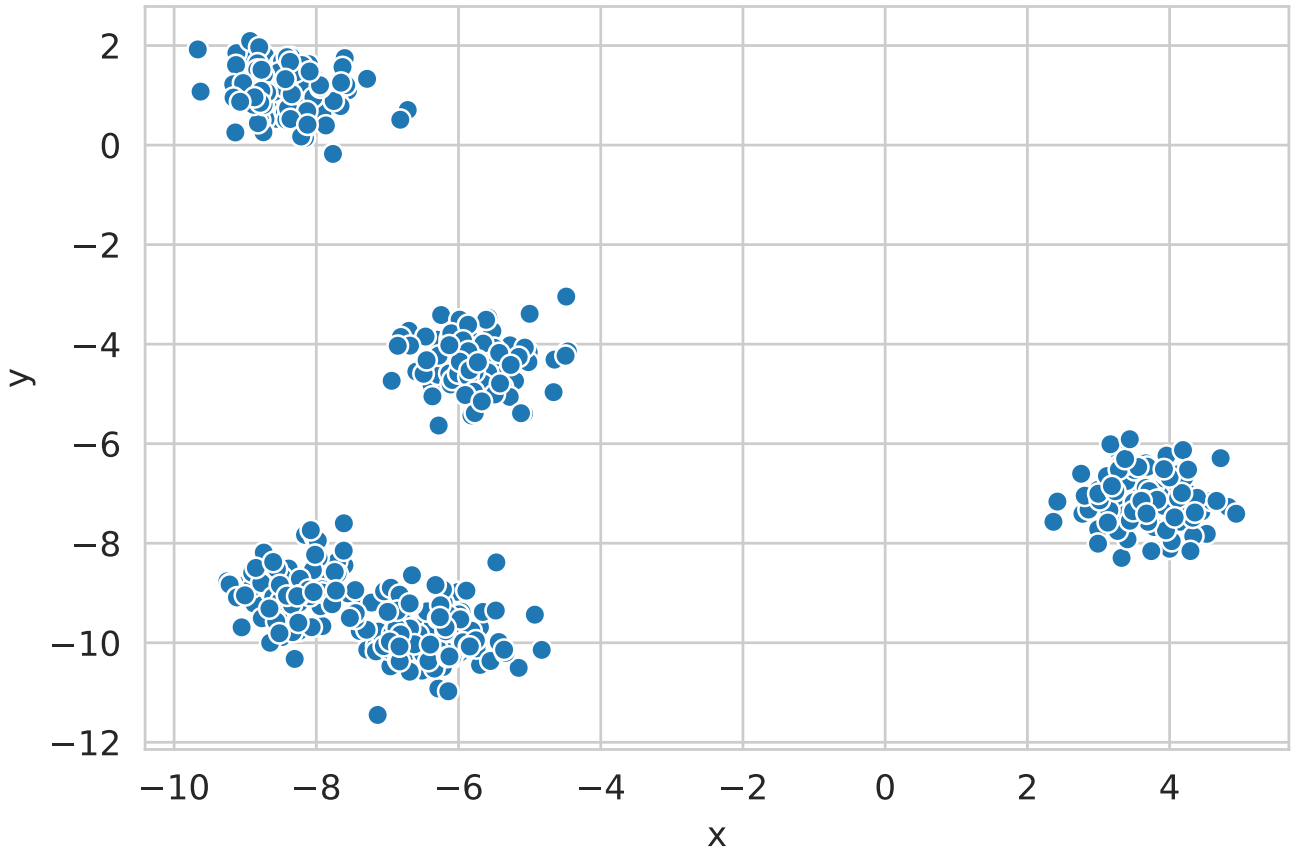
- Supervised learning is about learning from labeled examples – it is sometimes called *learning from a teacher*.
- The goal is to learn a model (i.e. to fit model parameters) that explains the observed data.
- And at the same time is able to generalize to unseen data.
- We will see that there is a delicate balance between fitting the data and guaranteeing generalization to new data – which is the ultimate goal.
- Models we will see: linear discriminants and regression, Support Vector Machines (SVMs), kernel machines, decision trees.

Unsupervised learning: learning without teachers

- Can we learn even without a teacher?
- Well, even very small children are able to learn via exploration of their environment.
- And, after all the amount of unlabeled data vastly outnumbers the available labeled data.

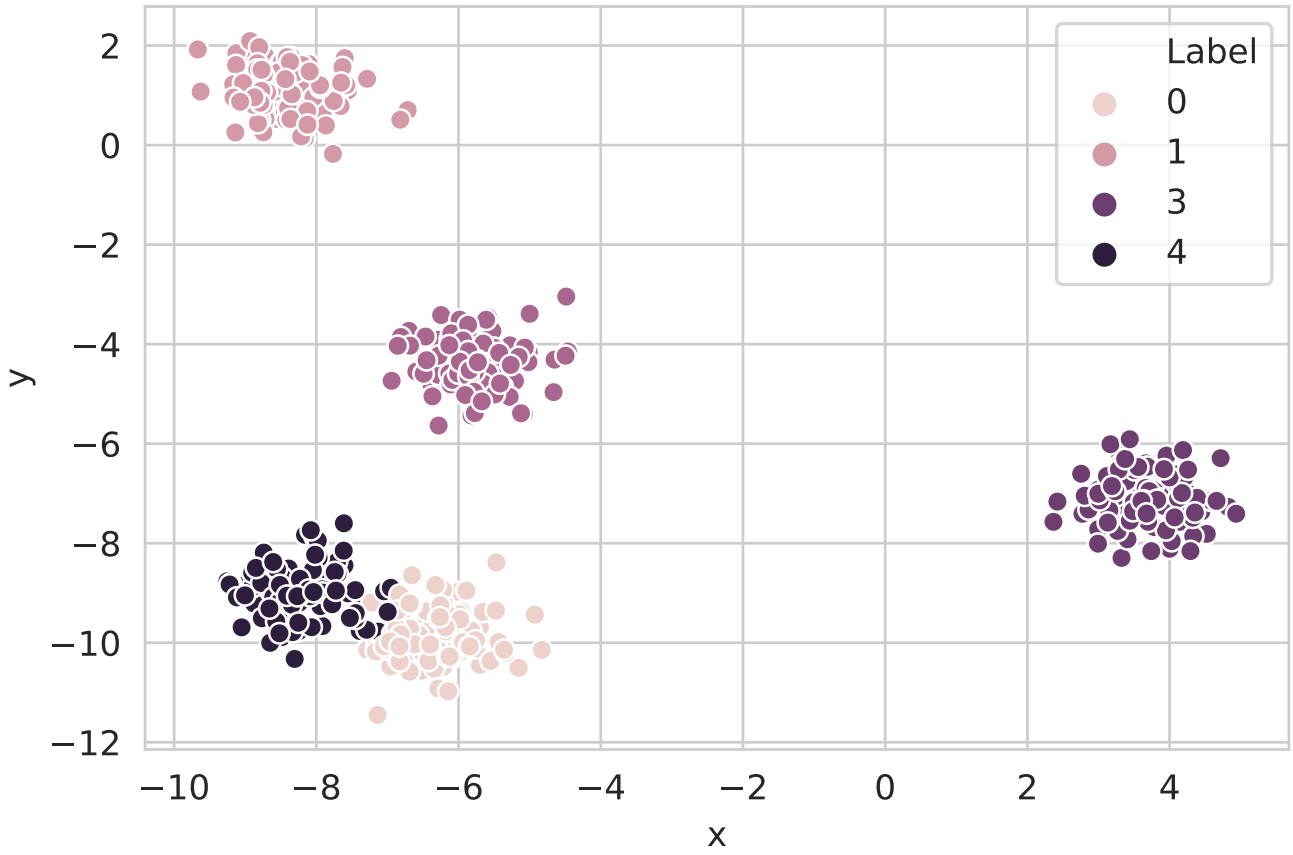
Unsupervised learning: pure data

- Let's say someone gives us some data (in two dimensions).
- Say, something like this:



Unsupervised learning: recovering latent structure

- We would like to learn the structure of the data.
- And recover a hypothesis like this:



Unsupervised learning: take home message

- Again, we would like to learn a hypothesis that generalizes to new data we want to apply the model to.
- Unsupervised learning is about learning from unlabeled data.
- There is actually a spectrum of supervision regimes: unsupervised, semi-supervised, weakly-supervised, self-supervised, fully-supervised...
- Learning from non-fully supervised data is an extremely hot topic in machine learning today.
- Models we will see: K-means clustering, agglomerative clustering, Principal Component Analysis (PCA), t-SNE.

Unsupervised learning: take home message

- A slide borrowed from Yann LeCun:

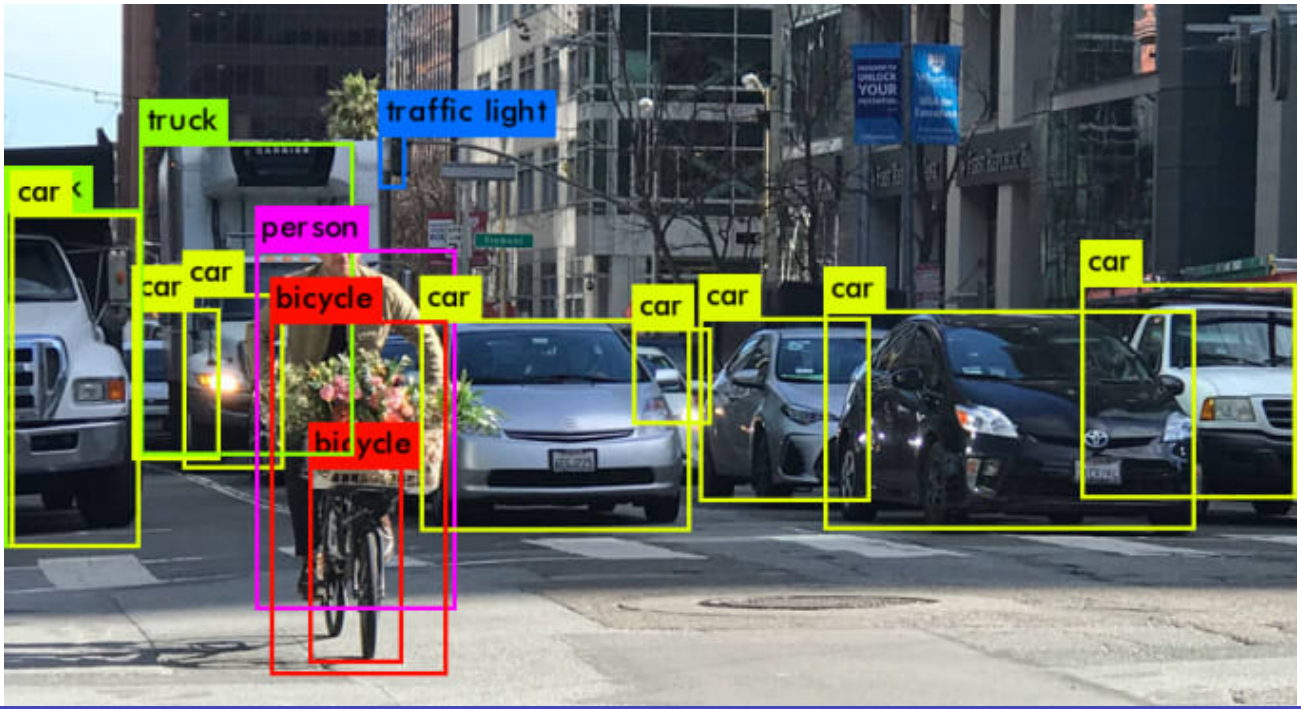
"Pure" Reinforcement Learning (cherry)
- The machine predicts a scalar reward given once in a while.
- **A few bits for some samples**

Supervised Learning (icing)
- The machine predicts a category or a few numbers for each input
- Predicting human-supplied data
- **10→10,000 bits per sample**

Unsupervised/Predictive Learning (cake)
- The machine predicts any part of its input for any observed part.
- Predicts future frames in videos
- **Millions of bits per sample**

Deep composable models

- Deep Neural Networks: we will see how modern neural networks work, how to apply then to supervised learning problems, and how to train them and monitor their performance.
- Computer Vision and Deep Convolutional Neural Networks: we will see how convolutions can be used to solve the weight explosion problem and how to apply CNNs to object recognition problems.



Better than human?

- AlphaGo Trailer
- AlphaStar

The tools of the trade

- Python: the language of choice for modern machine learning.
- Jupyter/Colab: a notebook-based system for reproducible and self-documenting science.
- Numpy: Python bindings to mathematical programming libraries.
- Scikit-learn: a toolkit with many implemented learning algorithms.
- Pandas: data management, analysis, and manipulation library par excellence for Python.
- Matplotlib/Seaborn: visualization libraries built on top of Python/Numpy/Pandas.
- Tensorflow/Keras: a graph-based, automatic differentiating framework for deep learning in Python.

Four countries and two continents

What I do

- Visual recognition: local pyramidal features, color representations for object recognition, semi-supervised and transductive approaches, action recognition.
- Person re-identification: iterative sparse ranking, semi-supervised approaches to local manifold estimation.
- Multimedia and HCI for cultural heritage: visual profiling of museum visitors, knowledge management for cultural heritage resources, personalizing cultural heritage experiences, human-computer interaction.
- Deep learning: applied and theoretical models for visual recognition, network compression, lifelong learning, reinterpretation of classical approaches in modern learning contexts.
- Other random interests: functional programming languages, operating systems that don't suck, long-distance bicycle touring, Emacs, the Grateful Dead.

Calendar

Date @ Time	Topics
09/01/2020 @ 14:00	Introduction and Python Crash Course (lab)
10/01/2020 @ 14:00	Mathematical foundations
16/01/2020 @ 14:00	Numerical programming and reproducible science (lab)
17/01/2020 @ 14:00	Supervised machine learning
23/01/2020 @ 14:00	Unsupervised machine learning
24/01/2020 @ 14:00	The bias/variance tradeoff and regularization (lab)
13/02/2020 @ 09:00	Connectionist models and neural networks
13/02/2020 @ 14:00	Deep Learning I (lab)
14/02/2020 @ 09:00	Deep model complexity and Convolutional Neural Networks
14/02/2020 @ 14:00	Deep Learning II (lab)

Notes:

- Format: 1.5 hours of lecture, break, 1.5 hours of lecture/lab, discussion.
- Most lectures have a complementary Google Colab Notebook that interactively illustrate important topics.

A note on laboratories

- The laboratories for this course have been designed using Google Colaboratory notebooks.
- To access these you must have a Google account.
- Is there anyone here that does not have a Google account?

Intermission

Questions? Comments?

Python crash course

An interactive lesson

- You may access an interactive version of this lesson here:

  `http://bit.ly/DTwin-ML1`

- Important: you must save a copy of this notebook in order to interact with it.
- Choose "Save a copy in Drive..." from the File menu to do this.

Why python?

When write/compile/test/re-compile is too slow

- In many situations the usual write/compile/test/re-compile cycle in C/C++/Java is too slow.
- This happens a lot when you are testing out ideas, or when you are iterating design ideas and need to test many ideas quickly.

Enter Python

- Python is simple to use, but it is a real programming language.
- Python also offers more error checking than C, it has high-level data types built in, such as flexible arrays and dictionaries.
- Python is an interpreted language, which can save you considerable time during program development because no compilation and linking is necessary.

Python is interactive

Python supports interactive, iterative development

- The interpreter can be used interactively, which makes it easy to experiment and to write throw-away programs
- It also allows for incremental testing during bottom-up program development.

Python is compact, readable and efficient

- Programs written in Python are typically much shorter than equivalent C, C++, or Java programs, for several reasons:
  - the high-level data types allow you to express complex operations in a single statement;
  - statement grouping is done by indentation instead of beginning and ending brackets; and
  - no variable or argument declarations are necessary.

The high-entropy version

Language details

- Python is strongly typed (i.e. types are enforced).
- It is also dynamically and implicitly typed (i.e. no variable declarations)
- It is case sensitive (i.e. var and VAR are two different variables).
- It uses automatic memory management with garbage collection.
- And it is object-oriented (i.e. everything is an object).

Python is interactive and helpful

- We run the Python console using the 'ipython' command.
- This runs an interactive console (think Matlab) that can be used to run scripts, load definitions, and inspect the definitions already made.
- We can also run IPython in notebook mode (now called Jupyter) which is a nice way of experimenting with Python.
- [RUN NOTEBOOK NOW]

Python is reflective and self-documenting

How to get information

- You can use the `help()` function to get information about anything.
- You can use the `dir()` function to get a list of object members.
- Use "scratch" notebook cells experiment and as an interactive system for figuring out how things work.
- All objects can be inspected and interrogated, poked and prodded to see how they respond.
- Jupyter notebooks (and the IPython console) are excellent ways to experiment and explore possibilities.

Versions and distributions

Python versions

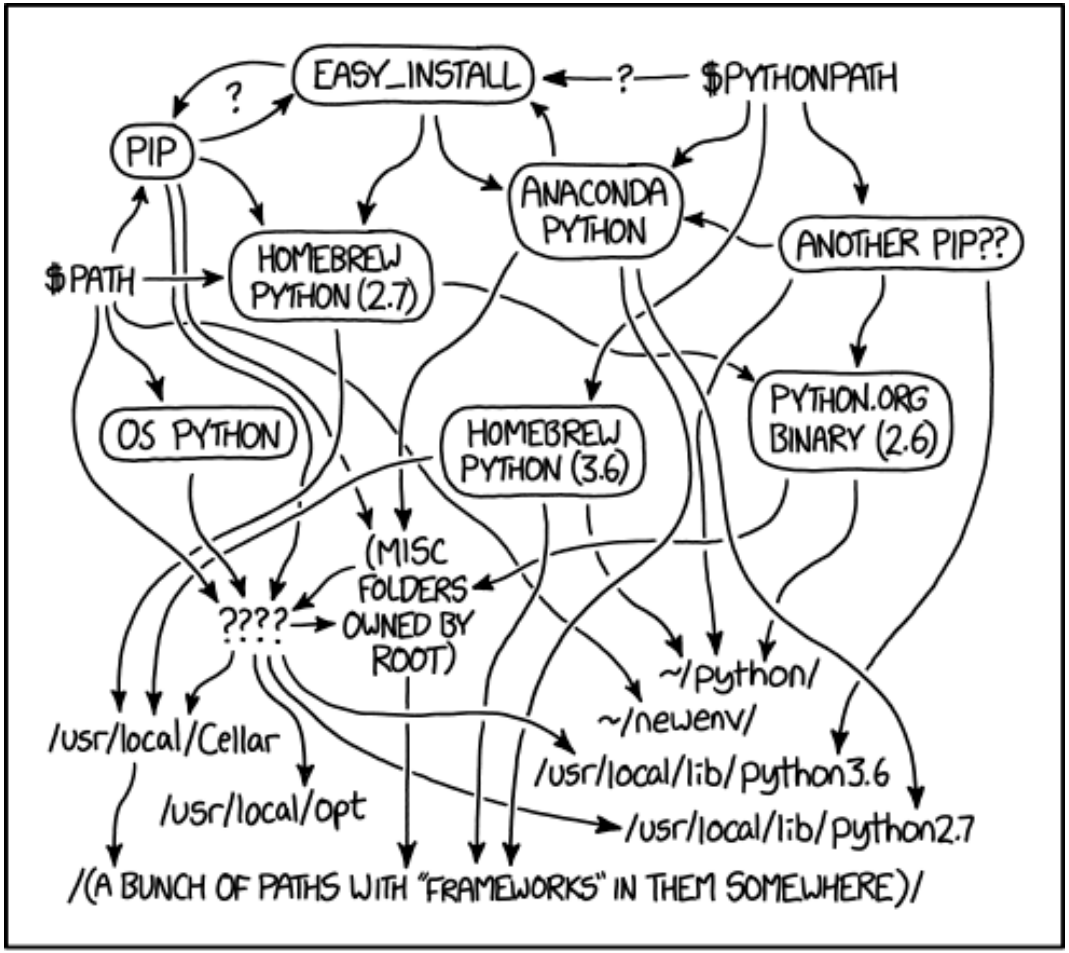
- In this course we will use Python 3.7
- Python 3 brings a number of improvements to the syntax and feature set of the language.

Python distributions

- For the examples, laboratories, and exercises in this course we will be using Google Colaboratory (*Colab* for short)
- Colab is based on the Jupyter notebook system.
- The Jupyter Notebook is an open-source web application that allows you to create and share documents containing live code, equations, visualizations and narrative text
- Uses include: data cleaning and transformation, numerical simulation, statistical modeling, data visualization, and machine learning.
- To access Colab you *must* have a Google account.

Python package management

- Using Colab allows us to avoid *Python package management hell*.



MY PYTHON ENVIRONMENT HAS BECOME SO DEGRADED THAT MY LAPTOP HAS BEEN DECLARED A SUPERFUND SITE.

Basic data types

Numbers

- Numbers: the integer numbers (e.g. 2, 4, 20) have type `int`, the ones with a fractional part (e.g. 5.0, 1.6) have type `float`.
- Expression syntax is straightforward: the operators +, -, * and / work just like in most other languages; parentheses (()) can be used for grouping.
- The equal sign (=) is used to assign a value to a variable.

Strings

- Besides numbers, Python can also manipulate strings, which can be expressed in several ways.
- They can be enclosed in single quotes ('...') or double quotes ("...") with the same result.
- String literals can span multiple lines. One way is using triple-quotes: """..."""" or '''...'''.

Basic datatype examples

Numbers

```
>> 50 - 5*6
20
>> (50 - 5.0*6) / 4
5.0
>> 8 / 5.0
1.6
```

Strings

```
>> 'spam eggs' # single quotes
'spam eggs'
>> 'doesn\'t'  # use \' to escape the single quote...
"doesn't"
>> "doesn't"   # ...or use double quotes instead
"doesn't"
>> '"Yes," he said.'
'"Yes," he said.'
```

Basic datatype examples

Multi-line strings

```
print("""
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
""")
```

Lists

Lists are the most commonly used compound datatype

- Python supports many compound data types that group together other values.
- The most versatile is the list, which can be written as a list of comma-separated values (items) between square brackets.
- They can contain items of different types, but usually all have the same type.

Examples

```
>> squares = [1, 4, 9, 16, 25]
>> squares
[1, 4, 9, 16, 25]
>> squares[0]    # indexing returns the item
1
>> squares[-1]
25
>> squares[-3:]  # slicing returns a new list
[9, 16, 25]
```

Lists (continued)

Lists are mutable (strings are not)

- Lists are mutable data structures (you can change their values).
- Strings can be indexed like lists, but are immutable.

Examples

```
>> cubes = [1, 8, 27, 65, 125]  # something's wrong here
>> cubes[3] = 64  # replace the wrong value
>> cubes
[1, 8, 27, 64, 125]
>> foo = '1234567'
>> foo[3]
'4'
>> foo[3] = '9'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Lists (continued)

Like arrays, but more flexible

- Lists are kind of like arrays (random-access), but can expand and change size.

List examples

```
>> cubes.append(7 ** 3)  # and the cube of 7
>> cubes
[1, 8, 27, 64, 125, 216, 343]
>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>> letters
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>> # replace some values
>> letters[2:5] = ['C', 'D', 'E']
>> letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>> # now remove them
>> letters[2:5] = []
>> ...
```

Tuples and sequences

Tuples

- Tuples are like lists, but are used in different situations and for different things.
- Tuples are immutable, and usually contain a heterogeneous sequence of elements that are accessed via unpacking (see later example) or indexing.

Example

```
>> t = 12345, 54321, 'hello!'
>> t[0]
12345
>> t
(12345, 54321, 'hello!')
>> # they can contain mutable objects:
... v = ([1, 2, 3], [3, 2, 1])
>> v
([1, 2, 3], [3, 2, 1])
```

Tuples packing and unpacking

Handling multiple values

- Tuple construction is referred to as "tuple packing" since you are packing elements together into a single compound data structure.
- The reverse is also possible, by which tuple values are unpacked into a sequence of variables.

Example

```
>> t = 12345, 54321, 'hello!'
>> (x, y, z) = t
>> x
12345
>> y
54321
>> z
'hello!'
```

Dictionaries

A most useful data structure

- Python relies heavily on the use of dictionaries to organize key/value pairs.
- Dictionaries are sometimes called hashes, associative arrays, or HashMap in Java and std::Map in C++.

Example

```
>> tel = {'jack': 4098, 'sape': 4139}
>> tel['guido'] = 4127
>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>> tel['jack']
4098
>> tel.keys()
['guido', 'irv', 'jack']
>> 'guido' in tel
True
```

If statements

Familiar, yet unfamiliar

- The `if` statement is probably the most common control-flow tool in any programming language.
- It is also a good example of one of the most controversial syntactic features of Python, the fact that indentation matters.
- Recall how in most programming languages braces ({ and })are used to indicate scope and logically sequential blocks.
- In Python, code that is indented at the same level is considered to be in the same block – exactly as if there were enclosing braces.
- This is used extensively in function definitions, to indicate the logical blocks for `if ... then ... else` constructions, and in class definitions.

If statements (continued)

Example (if)

```
>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
More
```

For statements

Iteration over *sequences*

- The `for` statement in Python also uses indented blocks.
- An important difference is that in Python, `for` loops are generalized iterators over sequences (like Java iteration over collections).
- In most languages, `for` loops iterate over ranges of integers.
- In Python, iteration is always over a sequence data type like a list.

Example

```
>>> # Measure some strings:
... words = ['cat', 'window', 'defenestrate']
>>> for w in words:
...     print(w, len(w))
cat 3
window 6
defenestrate 12
```

range()

When you need to iterate over integers

- Use the `range()` function to iterate over a sequence of numbers; it generates lists containing arithmetic progressions.
- The given end point is never part of the generated list; range(10) generates a list of 10 values, the legal indices for items of a sequence of length 10.
- The range can start at another number, or use a different increment.

Example

```
>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>> range(5, 10)
[5, 6, 7, 8, 9]
>> range(0, 10, 3)
[0, 3, 6, 9]
>> range(-10, -100, -30)
[-10, -40, -70]
```

Iteration over sequences

Enumerating list items

- Sometimes you need list items and their indices – this is called enumerating list items.
- You can use `len()` and `range()` for this.
- Or, you can use the `enumerate()` function and unpack the pairs.

Example

```
>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>> for i in range(len(a)):
...     print(i, a[i])
>> enumerate(a)
<enumerate object at 0x7fbe30ee56e0>
>> list(enumerate(a))
[(0, 'Mary'), (1, 'had'), (2, 'a'), (3, 'little'), (4, 'lamb')]
>> for (i, name) in list(enumerate(a)):
...     print(i, name)
```

Generalized iteration

The map paradigm

- List comprehensions provide a concise way to create lists.
- Common applications are to make new lists where each element is the result operations applied to each member of another sequence (this is called a map).
- Or to create a subsequence of those elements that satisfy a certain condition (this is called a filter).

Example

```
>> squares = []
>> for x in range(10):
...     squares.append(x**2)

>> # Much better:
... squares = [x**2 for x in range(10)]
>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Generalized iteration (continued)

Filtering

- A list comprehension consists of square brackets containing an expression followed by a `for` clause.
- Then zero or more `for` or `if` clauses.
- The result is a new list resulting from evaluating the expression in the context of the for and if clauses which follow it.

Example

```
>> [x ** 2 for x in range(10) if (x % 2) == 0]
[0, 4, 16, 36, 64]

>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]

from math import pi
>> [round(pi, i) for i in range(1, 6)]
[3.1, 3.14, 3.142, 3.1416, 3.14159]
```

Function definitions

The basic definition

- The keyword `def` introduces a function definition; it must be followed by the function name and the parenthesized list of formal parameters.
- The statements that form the body of the function start at the next line, and must be indented.

Example

```
# Print Fibonacci numbers up to n.
def fib(n):
    """Print a Fibonacci series up to n."""
    a, b = 0, 1
    while a < n:
        print(a, ' ')
        a, b = b, a+b

# Now call the function we just defined:
fib(2000)
```

Function documentation

Functions are self documenting.

- The first statement of the function body can optionally be a string literal
- This string literal is the function's documentation string, or docstring.

Example

```
def sq(n):
    """Return the square of n, accepting all numeric types:

    >> sq(10)
    100

    >> sq(10.434)
    108.86835599999999
    ...
    """
    return n*n
```

Default parameter values

Only pass what you need

- Default values define functions that can be called with fewer arguments.

Example

```
def ask_ok(prompt, retries=4, complaint='Yes or no, please!'):
    while True:
        ok = raw_input(prompt)
        if ok in ('y', 'ye', 'yes'):
            return True
        if ok in ('n', 'no', 'nop', 'nope'):
            return False
        retries = retries - 1
        if retries < 0:
            raise IOError('refusenik user')
        print(complaint)
```

Named parameters

Unambiguous argument passing

- Functions can be called with keyword arguments: `kwarg=value`.

Example

```
def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue')
    print("- This parrot wouldn't", action,
    print "if you put", voltage, "volts through it."
    print "- Lovely plumage, the", type
    print "- It's", state, "!"

parrot(1000)                                          # 1 positional
parrot(voltage=1000)                                  # 1 keyword
parrot(voltage=1000000, action='VOOOOOM')             # 2 keyword
parrot(action='VOOOOOM', voltage=1000000)             # 2 keyword
parrot('a million', 'bereft of life', 'jump')         # 3 positional
parrot('a thousand', state='pushing up the daisies') # 1 positional and
                                                      # 1 keyword
```

Lambda expressions

Throwaway function definitions

- Sometimes you need to provide a simple function as an argument (e.g. the comparison for a sort).
- The `lambda` keyword allows you to create small anonymous functions.
- They can be used wherever function objects are required, but are syntactically restricted to a single expression.

Example

```
pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
pairs.sort(key=lambda pair: pair[1])
pairs
```

Defining your own modules

Rolling your own

- It's very useful to group your code up into reusable packages.
- You can access code in saved files using the `import` directive.

Example

```
# Fibonacci numbers module (in file fibos.py).
def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print(b),
        a, b = b, a+b

def fib2(n): # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Importing modules

The basic import form

- You can use `import` to import a namespace.
- Then you must use fully-qualified names to access module members.
- You can also directly import module members into the global namespace.
- And you can alias imported modules for convenience.

Example

```
# We already put our definition in 'fibos.py'
import fibos
fibos.fib(10)
fibos.fib2(20)

from fibos import fib, fib2
fib2(10)    # Now no 'fibos' prefix.

import fibos as f
f.fib(10)   # Namespace alias.
```

Overview

- Some Python code makes extensive use of object-oriented programming techniques.
- Classes have been added to the Python language with a minimum of new syntax to learn and remember.
- In this first crash course we will only introduce the basics.
- In particular, we will not discuss advanced topics like inheritance and constructor chaining.

Basic class definition

- The simplest form of class definition looks like this:

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

- Class definitions, like function definitions (def statements) must be executed before they have any effect.
- The statements inside a class definition will usually be method definitions.
- Definitions inside a class normally have a peculiar form of argument list, dictated by the calling conventions for methods.
- When a class definition is entered, a new namespace is created, and used as the local scope - thus, all assignments to local variables go into this new namespace.
- When a class definition is exited normally (via the end), a class object is created.

A simple example

Example

```
class MyClass:
    a = None
    b = None

    def __init__(self, a):
        print(self.a)
        self.a = a
        self._x = 123
        self.__y = 123
        b = 'meow'
```

- `print(self.a)` : doesn't find an instance variable and thus returns the class variable.
- `self.a = a` : a new instance variable a is created.
- `self._x = 123` : creates an instance variable "not considered part of the public API".

Instantiating and accessing members

- Class objects support two kinds of operations: attribute references and instantiation.
- Attribute references use the standard syntax used for all attribute references in Python: `obj.name`.
- Valid attribute names are all the names that were in the class's namespace when the class object was created.
- Advice: don't use class attributes, which are kind of like static class members, instead assign attributes in the constructor.
- Class instantiation uses function notation. Just pretend that the class object is a parameterless function that returns a new instance of the class.

A more complex example

- Here is a complete class definition.
- Note the `self` calling convention in the constructor.
- Let's add a method to add two complex numbers...

Example

```
class Complex:
    def __init__(self, realpart, imagpart):
        self.r = realpart
        self.i = imagpart

x = Complex(3.0, -4.5)
x.r, x.i
```

Summary

- Python is a flexible, modern language with a great ecosystem.
- It is also very easy to learn by doing things with it.
- By this, I mean that you don't need to spend a long time studying it's complexities before you can do useful things with it.
- General advice:
  - Import only what you need from modules.
  - Use import aliases to make your life easier.
  - But use import aliases sparingly to make life easier on others.
  - Basic skeleton is: imports, top-level variable definitions (constants for parameters), class and function definitions, executable code.
- Use IPython consoles to experiment, and notebooks to organize ideas and snippets.
- Most importantly: learn by do

Reflections

The way forward

- Tomorrow we will see some of the mathematical foundations we will need to go forward.
- You do not need to become expert Python programmers, but you should make sure you grasp the basic concepts.
- Everything – including the mathematical fundamentals – will be illustrated using Python examples.
- Please take the time to review the Colab notebook from today and ensure you are comfortable working in the notebook interface.

Some links

- Here is the Official Python 3 Tutorial if you want to take a deeper look at the Python programming language.
- Here is a Harvard Article on the History of AI (with many links).

Some references

- The canonical reference for Artificial Intelligence:

  Stuart Russell and Peter Norvig, Artificial Intelligence: A Modern Approach. Pearson Education Limited, 2016.

- The canonical reference for Machine Learning:

  Christopher Bishop, Pattern Recognition and Machine Learning. Springer, 2006.

- A recent book on Deep Learning:

  Ian Goodfellow, Yoshua Bengio, Aaron Courville, Deep learning. MIT press, 2016.

- An excellent general book on Artificial General Intelligence:

  Nick Bostrom, Superintelligence. Dunod, 2017.