CMSC 320
Introduction to Data Science

Akilesh Praveen • apraveen@cs.umd.edu
Prof. John Dickerson • Fall 2020 • University of Maryland
https://cmsc320.github.io

Last Revision: October 12, 2021

Table of Contents

1 Notes & Preface

Course notes for CMSC320, under Prof. John Dickerson. Notes collected from previous and current lectures.

2 Introduction (L1)

What is Data Science?

Data Science is the application of computation and statistical techniques to address or gain insight. It's the intersection of statistics and Computer Science. Based on what I've learned thus far, learning to do data science is like learning how to use a TI-84 in statistics class. You're simply learning how to leverage programming tools in order to perform advanced, complex, and meaningful data-related operations.

It's the use of statistics and computer science in order to find real-world insights.

CS Hacking Skills

Math + Statistics Knowledge

DATA SCIENCE

Common sense (real world substantive experience)
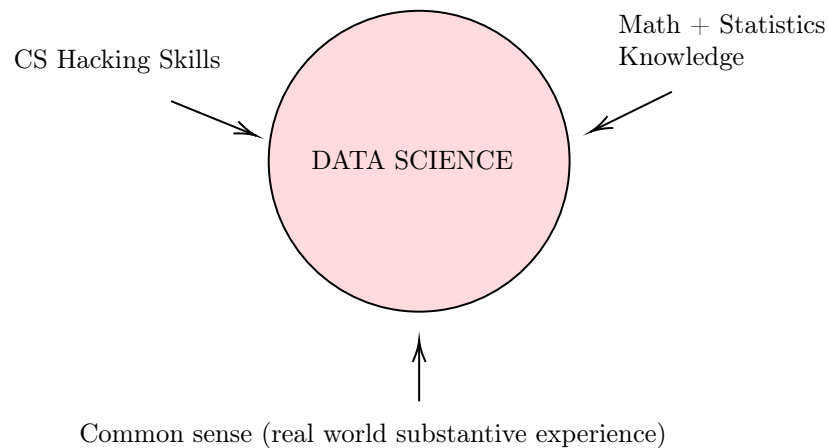
Topics

Here are the general topics that this class will cover.

- Processing data
- Visualizing data
- Understanding data
- Communicating data
- Extracting value from data

Tools

Here are some tools commonly employed by data scientists. We'll try to cover how to use most of them here.

- Python
- Scikit-Learn

- Docker

- PANDAS

- Spark

- TensorFlow

Conda

Conda is a package and environment manager for python that we can use with the command line. We can create multiple environments for us and install separate packages in each of them. This will be highly useful to us, as we sometimes want to consolidate the tools we use into separate environments.

3 Python, Jupyter, Literate Programming (L2)

> **Definition:**
>
> **Data Collection** $\rightarrow$ The process of measuring and gathering information on targeted variables.

Python Overview

Python is an interpreted, dynamically-typed, high-level, garbaged-collected, object-oriented-functional-imperative, and widely used scripted language

- **Dynamically Typed** - verifies type safety at runtime

- **High Level** - abstracted away from the raw metal and kernel

- **Garbage-Collected** Memory management is automated

- **Object Oriented Functional Imperative** - you can do bits of Object Oriented, Functional, and Imperative programming

Python is fast (developer time), intuitive, and commonly used in the industry.

Literate Programming

The idea of **literate programming** is that you have the *source code*, an *explanation* of the source code, and the *end result* of running the code all in one file. Usually, this file is identified as a *notebook*. In other words, the syntax is no different from regular code, you just get a more organized way to show off tables, plots, and other outputs generated from your code.

Basic Idea: Present code in the order that logic and flow of human thoughts demand, not the machine-needing ordering

Jupyter Notebook + Alternatives

Jupyter Notebook is a service that started off as `iPython`, but it's basically a web-based platform that we use for literate programming. Specifically, it supports Python-based literate programming. Most data scientists prefer it, and it can also apparently leverage big data tools, such as Apache Spark.

It saves files in `.ipynb` format, which most platforms (i.e. GitHub) have built in viewers for. Options to export in other readable formats are available. Basically, it's just Python with a bunch of bells and

whistles on top to make the output of your code look pretty.

Apache Zeppelin is an alternative data analysis tool, but we will stick to Jupyter for our purposes. This is because Jupyter seems to be preferred in industry.

RStudio is the equivalent, for people who prefer to use the R programming language for data science.

This course will be centered around Jupyter Notebook.

Python Coding Basics

Defining a Function

```python
# Define a Function that takes 2 arguments
def my_func(x, y):
  if x > y:
    return x
  else:
    return y
```

Note: Python is whitespace delimited Define a function that returns a tuple

```python
# Define a Function that takes 2 arguments
def my_func(x, y):
  return (x-1, y+2)
```

List Comprehensions in Python

To make lists in Python, you can use loops or the `map()` function, but a *pythonic* way of doing this would be to use a list comprehension. Below is a simple example.

Example: Make a list of all the squares of `{0,1,2,3,4,5,6,7,8,9}`

List Comprehension:
`squares = [i * i for i in range(10)]`

A good way of thinking about this is that it allows you to build sets like a mathematician. This is a common theme in data science, where we can find the intersection between a lot of math stuff and computer science stuff. It's good to know how lists are generated in a mathematical sense in Python for that reason. Here's an example where we translate mathematical notation into a Python list comprehension.

Example: Make a list of all odd natural numbers from 0 to 999

Math Notation:
$E = \{x \mid x \text{ in } \mathbb{N} \land x \text{ is odd } \land x < 1000 \}$

```
List Comprehension:
E = [x for x in range(1000) if x % 2 != 0]
```

Using Python3

We will use Python3. Since I used Python2 during my internship, I'm going to note some big changes to keep track of.

- Python3 is backwards incompatible. (Don't write in Python2!)

- Print has changed from a command to a function, so make sure to use proper function notation when invoking it.

- Division has changed. `1/2` no longer equals 0. `1/2 == 0.5` and floored division is now taken care of this way: `1//2 == 0`

Python vs. R for Data Scientists

Some arguments for both sides in terms of what to use.

- Python is a 'full' programming langauge. Also, if you've got prior experience with Python paradigms or just programming in general, that's a big plus in terms of learning curve.

- R has more mature 'pure statistics' libraries, but Python is apparently catching up.

- In terms of **processing speed**, R is certainly faster. It was designed and optimized for statistics processing.

- Python is preferable for machine learning operations, which is pretty big right now.

My personal choice will be to use Python as much as I can when I'm studying this course. Since it's more prominent in the tech industry, I should be using it more anyway.

Tabular Data

Data is an abstraction of some real world entity. It can also be referred to as: instance, example, record, object, case, individual. Each of these entities can be described by a set of *features* (i.e variables, features, attributes, etc.). Data can be processed into an nxm matrix where n is the number of entites and m is the number of attributes. This can be done through merging and processing various records.

The Classic Statistical View of Data

There are **four** main types of data: Nominal, Ordinal, Interval, and Ratio data. They can each be classified under two main subgroups, Categorical and Numerical data. Here's a visualization.
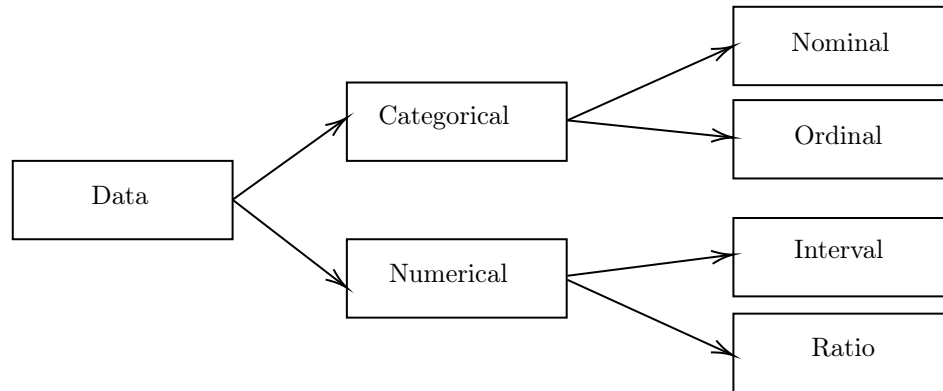
Nominal Data

A type of categorical data, **nominal data** value have **names** and describe the state of things. For example, your marriage status is nominal data because you can either be *single*, *married*, or *separated*. Another example is the type of drink you're going to have. Will it be *Milk*, *Beer*, or *Juice*?

The key here is that there can be *no quantitative values assigned to each of these categories*, as that would allow us to do math with them and would defeat the purpose of these labels. These values **cannot be easily compared**, so they have no material value. *E.g. being single is not quantitatively better than being married (objectively), and vice versa.*

> **Example:** What is your marital status?
>
> - Married
>
> - Divorced (separated)
>
> - Single

Ordinal Data

Ordinal data represents values that have names that describe the state of things, but in this case, there **is an ordering of those values**. This is what sets it apart from nominal data.

> **Example:** What did you think of the movie?
>
> - Strongly liked
>
> - Liked
>
> - Indifferent
>
> - Disliked
>
> - Strongly Disliked

Given how subjective some of these things can be, the distinction between nominal and ordinal can be **blurry** at times. For example, going back to our nominal example, some people may think that being single is quantitatively better than being married.

Interval and Ratio Data

Interval and Ratio data are pretty similar, and both can be used to measure things that can be represented by either integers or real numbers.

Interval data scales with fixed but arbitrary values. That might sound silly, but a good example is **dates**. Below is an example of two data comparisons of interval data that seem arbitrary, but indeed hold integer value.

> **Example:** The following two operations are equal.
>
> ```
> 10/1/2019 - 9/1/2019
> 10/1/2018 - 9/1/2018
> ```

The measures don't look like integer values at first, but we can quantify them by marking them with days.

Here's what sets **Interval** data apart, however. You have **no method** of computing ratios or scales with it. For example, never mind that you can try computing `(9/1/2019 × 8/25/2015)`, the unit of the answer would be totally useless to us, and neither would the actual number, even if you went ahead with the operation.

Ratio data is, in essence, the same as interval data in that it is numerical, but the scale itself **has a true zero**. While dates don't necessarily have a true zero, we can say that money counts as ratio data. For example, having zero money means that you're at the absolute zero of that scale, whereas the absolute zero for dates is disputable. Are we saying we're starting at O A.D.? The Big Bang? Even earlier?

Differentiating between the two is usually a case-by-case basis thing, which is what I'm thinking is the best way to handle any conflicts I end up running into between ratio and interval data.

> **Example:** Interval data
>
> Temperature on the scale of Celsius or Fahrenheit is interval-type data because $0°$ is set to an arbitrarily fixed point. In other words, $0°$ does NOT mean that there is 0 (or no) heat. Also, we can't scale it properly- $30°F$ isn't twice as hot as $15°F$.
>
> **Example:** Ratio data
>
> Temperature on the Kelvin scale is ratio data. $0K$ is set at legitimate absolute zero. In other words, we take $0K$ to mean that there IS 0 (or no) heat and thus a true fixed point. Also $50K$ is truly twice as cold as $100K$. **Example:** Ratio data
>
> Weight in terms of grams is considered to be ratio data. This is because 0g is a fixed point of no weight. Furthermore 4g is twice as much as 2g and so on.

OK to compute....	Nominal	Ordinal	Interval	Ratio
frequency distribution	Yes	Yes	Yes	Yes
median and percentiles	No	Yes	Yes	Yes
addition or subtraction	No	No	Yes	Yes
mean or standard deviation	No	No	Yes	Yes
ratio, or coefficient of variation	No	No	No	Yes

Data Science at a Glance

Data science is basically manipulating and computing using data. As such, we need to shift our thinking from writing **imperative** code to manipulate **data structures** to creating **sequences and pipelines** to conduct operations on **data**. That stuff is covered more in 420 and 424, for reference.

More often than not, we have to take the data that we've found and make it easily understandable for humans. This is called Data Representation.

> **Definition:**
>
> **Data Representation** → The natural way to think about data, in a human way.

Here are some ways that we think about data in this class:

- **One Dimensional Arrays** → E.g. <'red', 'blue', 'green'> or <0,3,4>. We can use functions like map, fold and filter to manipulate these.

- **N-Dimensional Arrays** → Also known as **tensors**.

    - For example, a Tensor of dimensions [6,4] is just a 6×4 matrix.
    - Similarly, a Tensor of dimensions [4,4,2] is a 3D array.
    - Here, we can start to make use of **Linear Algebra** for further data manipulation. Some example operations that we can use to mess with tensors:
        * Matrix/Tensor Multiplications
        * Transpose
        * Vector Multiplication
        * Matrix Factorization (we will explore this later)

- **Sets** of objects, or **Key/Value Pairs**

- **Tables/Relations** → This goes into relational databases, which is the basis of SQL. We'll go into this later.

- **Hierarchies/Trees/Graphs** → This sort of spills over into data structures, but they've got some additional nuances included with them.

    - They tend to make use of 'path' queries
    - Graph and Tree Algorithms will be useful here, efficiency is key
    - Example: networks are represented this way, we'll cover that later in this class

> **Definition:**
>
> **Data Processing Operations** → Take one or more datasets as input and produce one or more datasets as output.

Both Data Representation and Data Processing Operations allow one to think at a higher level of abstraction which leads to simpler and easier-to-understand scripts. It also provides independence between the abstract operations and concrete implementation as well as the ability to swithc from one implementation to another easily.

4 Getting Data to Work With (L3)

Acquiring Data

Here are some examples of how we can grab data from places. Pretty obvious, common sense stuff. We're going to explore all of these as we move forward.

- Direct download from online or loading it from local storage
- Generate the data locally via a simulation or equivalent program
- Query data from a database
- Query data from an API
- Scrape data from a website

When you pull from APIs, you're going to want to be using HTTP Requests.

RESTful APIs

This stands for REpresentational State Transfer APIs, and it's basically a standard that enforces that APIs do a few things. It says that they should support these basic operations:

- GET → Query a data entry
- POST → Create a new data entry
- PUT → Update an existing data entry
- DELETE → Delete an existing data entry

RESTful APIs are also supposed to be stateless. That is, with every API request, you send a token of who you are, and you get a current capture of the data at that time/edit the data.

> **Definition:**
>
> **stateless** → With every request, you send along a token/authentication of who you are.

A good example of a REST API is Github, where you can use REST API calls on your repositories.

There are other guiding principles and miscellaneous guidelines for RESTful APIs, which can all be found at `https:/restfulapi.net`

GraphQL → REST has been adopted by many developers and is widely regarded as the traditional way to send data over HTTP. GraphQL, on the other hand, is a revolutionary new player that's presented as a way to *replace* legacy REST APIs *(back4apps)*

Oauth

If you want to grant an app access to your identity without actually giving it your username and password, is there a way to do that? The answer is **yes**, because this is a common software engineering problem.

OAuth is the standard for *access delegation* used for internet users to grant websites access to their information on other sites. **OAuth** grants *access tokens* that give *(possibly incomplete)* access to a user or app without exposing a password.

A pretty good example of this is Google's sign in page on other websites. How do you think other websites conduct sign in without knowing your password for your Google account?

GET Requests

Assume we used Python's `requests` module to query a server with a `GET` request.

First, we'd get either a `CSV`, `JSON`, or `HTML/XML/XHTML` file back, in response. This is the data that we have to sift through. *Note:* You might also get a domain-specific file, like an **rvt** file. You're always welcome to make your own filetype for storing data, but make sure it's actually documented somewhere.

Aside: Parsing CSVs and JSON

Never write your own `CSV` or `JSON` parsers. This is another example of reinventing the wheel. We'll use Python Libraries to do this more easily. *E.g. PANDAS*

More on Data Storage Formats

Definition:

Serialization → The process of converting objects into strings.
Deserialization → The process of converting strings back into objects.

`JSON` is a pretty common format for serializing objects. Plus, serializing objects makes it easier for humans to read and perform sanity-checks on. In Python, `JSON` is built with Strings, Lists, Dictionaries, and sometimes mixes of a few of those together.

Definition:

Document Object Model → A tree-based data storage method. For example, HTML is structured this way.
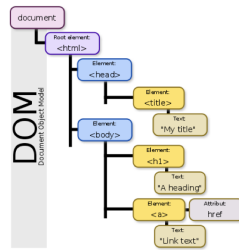
SAX

SAX (Simple API for XML) is a lightweight way to process XML. It generates a stream of events as it parses an XML file. IT helps us pay attention to individual parts of an XML file without having to process through the rest of it.

Parsing HTML

Parsing HTML is the hardest to do in this case, as I've seen many times before in hackathon projects. Although HTML's specifications are pure, the real world examples of it are pretty nightmarish, thanks to how it interacts with JavaScript and loads dynamic content. All in all, it's fairly unreliable in terms of parsing it manually.

In this case, we're best off using the Python library `BeautifulSoup`. We can also make use of Python's Regex, which is similar enough to Ruby regex that we worked with in 330. A website like Rubular-`https://pythex.org` will be useful in this case.

By combining `BeautifulSoup`, Regular expressions, and `GET` requests, we can make the process fairly streamlined. This is usually what we'll be using to scrape websites. In order to scrape more dynamic websites, we'd probably have to make use of Selenium. Check my 320 folder to find an example of a simple webscraper with `BeautifulSoup`.

Regex

Regular Expressions can be used to find strings and other necessary information. For example, this can be useful when trying to lcate strings that end in .pdf. Regex can be used to search for specific elements or groups of elements that match a pattern. This is useful for data munging and wrangling as well as to find specific patterns.

5 NumPy, Best Practices, Ethics (L4)

Available Technologies

Python has a bunch of 3rd party packages for scientific and numerical computation. Some examples are..

- **Numpy and Scipy** → Numerical and scientific function libraries.

- **NUMBA** → A Python compiler supporting 'Just in Time' compilation. That is, it supports compilation of code while code is running.

- **ALGLIB** → A cross-platform numerical analysis library

- **PANDAS** → An extensive data analysis tool with some neat built-in data structures

- **PyGSL** → GNU Scientific Library in Python

- **Scientific Python** → A collection of scientific computing modules for Python

These are a bunch of examples of what's available to developers right now, but we won't focus on all of it. Particular emphasis will be placed on Numpy and PANDAS.

NumPy Stack

The **NumPy** stack is the most commonly used out of all of these packages. It includes the following:

- Numpy - Works sort of like MatLab, just lets us handle a lot of number manipulation and mathematical operations

- Matplotlib and Seaborn - This is a plotting and graphing library

- iPython via Jupyter - interactive computing

- PANDAS - This gives us a bunch of data structures and data analysis tools to play with/keep track of our data. (Usually, you'll want to import your data into a PANDAS dataframe or something.)

- SciPy - integrates many other packages like NumPy

- SymPy - symbolic computation library

- Jupyter - This will be our medium for **literate programming**.

To see more about this stuff, search Google for the **NumPy Stack** and you'll find everything you need.

Misc About NumPy

Here are a few more notable things about Numpy:

- It contains the **n-dimensional array** object

- It contains 'sophisticated' functions that we can use

- It provides us with excellent tools to integrate C++, C, and even FORTRAN

- It has math capabilities that are highly useful to us (e.g. Linear Algebra, Fourier Transform, etc)

- Numpy also comes with a bunch of new DataTypes for us to use.

Aside: Numpy Arrays

Arrays in Numpy are homogenous data types, with many operations being performed in compiled code for performance. There are several differences between regular lists in Python and numpy.

- NumPy arrays have a fixed size. Modifying the size means creating a *new array*

- NumPy arrays must be of the same data type, but this can include Python objects - may not get performance benefits

- More efficient mathematical operations than built-in sequences types

---

NumPy Datatypes

There Wider variety of data types than are built-in to the python langauge by default. The `numpy.dtype` class includes:

- intc - same as C integener and intp - used for indexing

- int8, int16, int32, int64

- uint8, uint16, uint32, uint64

- float16, float32, float64

- complex64, complex 128

Numpy Arrays

In general, any numerical data that is stored in an array-like container can be converted to an ndarry through use of the `array()` function. The most obvious examples are sequence types like lists and tuples. Some useful functions:

- `zeros(shape)` - creates an array filled with 0 values with the specified shape. (default type is dtype is float64)

- `ones(shape)` - creates an array filled with 1 values

- `arrange()` - like Pythons's build in `range`

- `linespace()` - creates arrays with a specified number of elements, and spaced equally between the specified beginning and end values

- `random.random(shape)` - creates arrays with random floats over the interaval [0,1).

With arrays, there are a few ways to implement indexing.

- Single Dimension indexing $\rightarrow$ `x[2]` $\rightarrow$ access the second element

- Multi Dimension indexing $\rightarrow$ `x[1,3]` $\rightarrow$ access element at i = 1, j = 3 (also note that `x[1, 3] == x[1][3]` but `x[1, 3]` is more efficient to write)

- Slicing $\rightarrow$ `x[2:5]`$\rightarrow$ return all values of array from index 2 to 5-1

Aside: Array Operations

Basic operations applied to arrays are applied element wise. The result of applying these basic operation is a new array with the results in the elements.

It is important to note that multiplicate is done element wise. Therefore, to perform matrix multiplication, you must preform the dot product.

There are also some built in methods of `ndarray` objects. Universal functions which may also be applied include `exp, sqrt, add, sin, cos` etc.

An array shape can be maniupated by a number of methods such as `resize(size)` which will modify an array in place and `reshape(size)` which will return a *copy* of the array with a new shape.

Linear Algebra with NumPy

One of `NumPy`'s most common uses lies within its **Linear Algebra** module. It allows us to do regular LA stuff, like `.transpose()` and `.inverse()` to matrices stored as n-dim arrays. Here's an example.

```
1  # Note: remember, we have to use NumPy's n-dimensional array ↩
       object here
2
3  array([[1.0, 2.0],
4         [3.0, 4.0]]).transpose()
5
6  u = eye(2) #unit 2x2 matrix; "eye" represent identity matrix
7  j = arrat ([[0.0, -1.0], [1.0, 0.0]])
8  dot(j, j) #matrix dot product
9
10 trace(u) # trace (sum of elements on Diagonal)
11
12 y = array([[5.], [7.]])
13 solve(a, y) # solve linear matrix equation
14
15 eig(j) # get eigenvalues/eigenvectors of matrix
```

SciPy

SciPy includes various tools and functions for solving common problems in **scientific computing**.

We won't use it much for now, but it's supposed to be good to know. Often you'll be able to find higher-level `Scipy` functions that will work around the need to call lower-level `Scipy` functions. It's got a lot of functionality built in, so make sure not to overlook it.

An example of using scipy would be to use the `scipy.integrate` function which allows you to integrate a function on bounds.

```
1  # Integrate on the function sinx from 0 to pi
2  res = scipy.integrate.quad(np.sin, 0, np.pi)
3  #there are also functions that allow you to utilize integral ↩
       rules like trapezoidal rule.
```

The Idea of Reproducibility

Starting from the same dataset, can we reproduce your analysis and get the same results? **This is the goal that we're trying to fulfill with our analysis**- we want our stuff to be reproducible! (Otherwise, what exactly does it even mean?)

Best Practices

Honestly, most of this stuff should be common sense.

- Use version control to keep track of code. (e.g. `git`)

- Use unit testing. (e.g. `unittest` module in python)

- Use libraries when you can. (don't reinvent the wheel!)

The Idea of Open Data

Some data should be widely available for everyone to use as they want, without restrictions from copyright, etc.

This is probably where all of our free data comes from, so this idea is super helpful to us as data scientists.

General Process

Here's the general process for data science- just so we have an idea of what's going on.

Data Acquisition	**Algorithm / Tool Development**	**Data Acquisition**
Acquire the data, then put it in a usable format.	If new tools are required to analyze your data, create them. If new algorithms may be needed to analyze it, synthesize them.	Use the tools you've built / setup and the algorithms you've created or implemented to analyze the data.
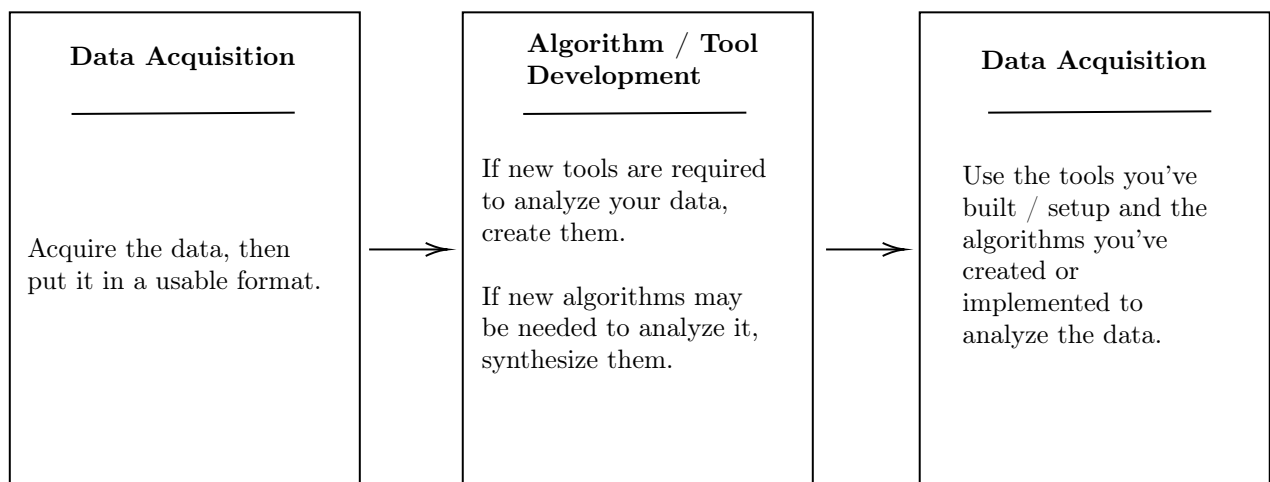
After we do that, we still technically have some programming left. In this new era of literate programming, there's one more step of processing we have to do with our results in order to make them publicly presentable and meaningful.

Communication of Results
Use the principles of **literate programming** to provide results, plots, and publication regarding your data science experiment. This gives you the ability to explain your choices and clearly interpret your results.
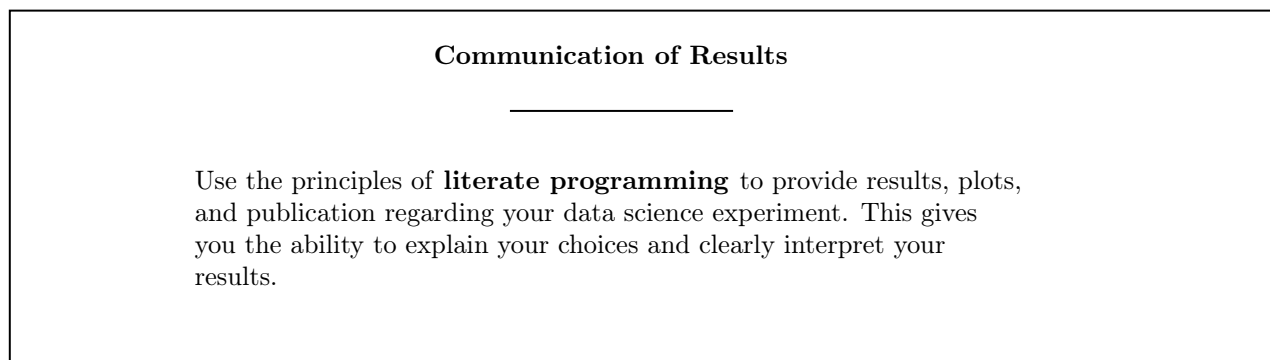
It's emphasized a lot here to think like an **algorithm developer**, as you're going to need efficiency in the data analysis that you perform. However, you also need to think like an experiment-conducting **data scientist**. We don't usually get enough training as the latter, so hopefully this course should be an introduction to that sort of stuff.

Project Organization

Make sure to organize your project in folders appropriately. Specifically, even if you have a lot of components, group with with a focus on experimental procedure.

You should certainly be isolating things like **data**, **tools**, and **experiments** into their own folders. Data could include your raw input data, along with data that you've done some processing on. Tools could include Python environment you're using, and experiments could include the meat of what your data science work will be- pipeline scripts, results, figures, plots, analysis scripts, etc.

A Little on Bias, Ethics, Responsibility

> **Aside: Fairness Through Blindness**
>
> The concept of not letting an algorithm look at protected attributes in order to keep it from forming potentially harmful biases.

A great example of fairness through blindness could be software that determines the outcome for a loan application. We want the results to be **independent** of an applicant's race, but they can be **dependent** on non-protected attributes, such as credit history and income.

> **Aside: FATML**
>
> **FATML** stands for Fairness, Accountability, and Transparency in Machine Learning.

Overall, here are some guiding principles for data ethics:

- Start with clear user need, with a focus on public benefit. (Can't go wrong with this!)

- Use data and tools that emphasize **minimum** intrusion/invasion of privacy. (Sometimes, we have no choice but to handle sensitive data)

- Create robust data science models that minimize bias and focus on objective accuracy.

- Be alert to public perceptions.

- Be open and accountable for your actions.

- Security is key- especially if working with sensitive data.

6   Tables, Relational DB, and Pandas (L5 + L6)

Tables

Here's the idea- we can abstract data into our own little data structures just like computer scientists do, and a lot of the time, in data science, tables are the optimal way to do that. (This is why software like PANDAS and Numpy have excellent support for these structures.)

Variables (Attributes/Labels/Columns)

id	age	weight	height
1	12	42.3	145.6
2	11	50.9	161.2
3	13.5	61.3	181.3
4	14	41.4	121.4
5	15	55	135.5

Observations (rows, tuples)

Known as the Index (ID) column. Usually, there are no duplicates of these allowed, and they're often ordered like this.

Here's an example table. I've highlighted and color coded the important aspects of it. Remember, don't think of this as the data structure itself- this is just an abstraction to help us keep track of our vast amounts of data. However, most table implementations do a pretty good job of representing the stuff I've color coded.

Selecting / Slicing

Selecting one or more of the rows or columns in particular to analyze. Examples:

- Select only columns ID + Age

- Select all rows with weight > 41

- We can also apply a combination of the above 2. (*You can combine select rules!*)

Example of slicing:

Aggregating / Reducing

Combining values across a column into a single value. (We don't do this across rows, because that obviously wouldn't make any sense. Think about it!). Examples:

- Find the sum of all row's columns

- Find the max of the weight column
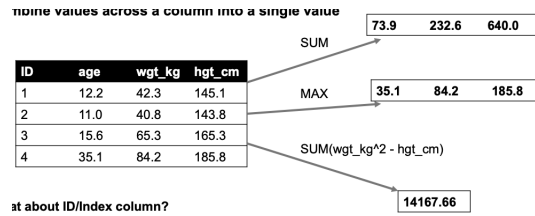
Note: It's usually never useful to aggregate/reduce the ID column, so for most cases, we ignore it when we perform such operations.

Map

Apply a function to every row, possibly creating fewer or more columns. This one's a little weird to think about without a clear example, so I'm including one below.
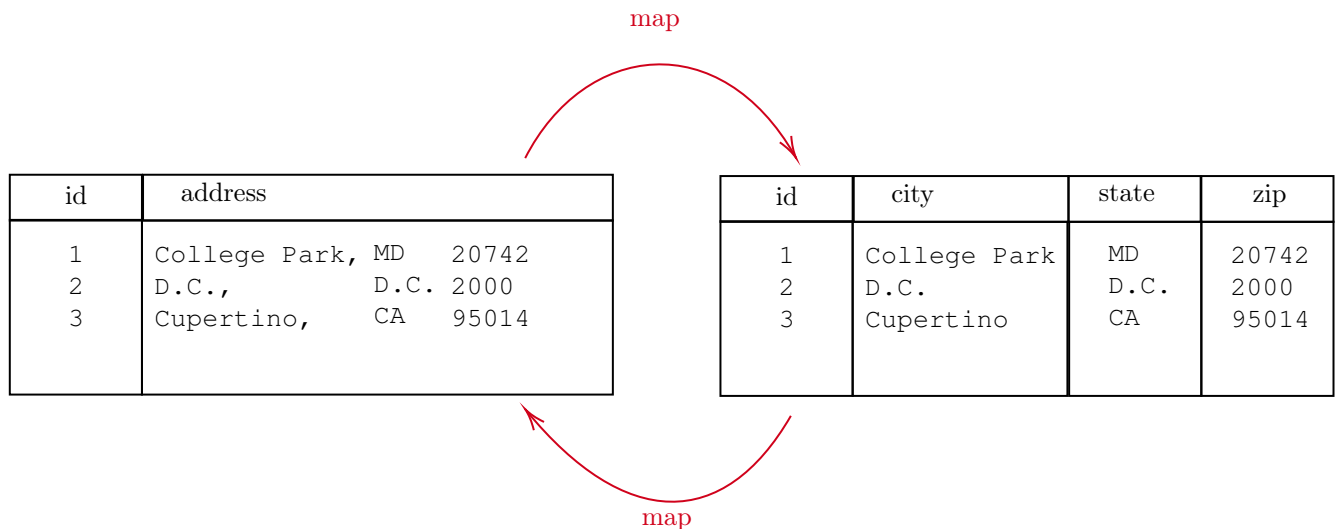
Notice how applying map to either table is valid in this case- sometimes we want to break down columns into more specific values, and sometimes we want to combine them into singular columns. Each of these operations is totally valid, and has its uses. (This is evident in the projects for this class).

Again, this is mostly about what you need. There's no necessary better or worse in this case (more columns does not always equal better data).

Group By

Group By is an operation that allows you to group tuples together based on the values in columns/dimensions. Let's say we had the following table of house addresses like earlier. This time, we'll add the number of people in each house as a column as well.

id	city	state	zip	people
1	College Park	MD	20742	3
2	Washington	D.C.	2000	4
3	Cupertino	CA	95014	3
4	College Park	MD	20742	2

Let's say we only wanted to see the data from a **single city**. In this case, let's pick College Park.

id	city	state	zip	people
1	College Park	MD	20742	3
2	Washington	D.C.	2000	4
3	Cupertino	CA	95014	3
4	College Park	MD	20742	2

This is what a 'Group By' operation would be perfect for. It'll basically just get us the rows that are from the city that we want.

Group By + Aggregate

We can combined Group By and Aggregate in pretty cool ways to get results that we want. For example, let's say we wanted to leverage the above table and get the sum total of all people who live in College Park, D.C., and Cupertino, respectively. By using a combo of Group By and Aggregate, we can totally do that. (*Group By* City, then perform summing *aggregation* operation.)

Union, Intersection, Difference

These are your usual set operations from statistics. However, this only works if the tables have identical attributes (columns). If they have identical columns, they are called **compatible tables**.

Examples: (Table A) ∪ (Table B) results in (Table C) where all three tables have the same attributes. Likewise, (Table A) ∩ (Table B) results in (Table D) where all three also have the same attributes.

ID	A	B	C
1	foo	3	6.6
2	bar	2	4.7
3	foo	4	3.1
4	foo	3	8.0

∪

ID	A	B	C
5	bar	1	1.2
6	bar	2	2.5
7	foo	4	2.3
8	foo	3	8.0

→

ID	A	B	C
1	foo	3	6.6
2	bar	2	4.7
3	foo	4	3.1
4	foo	3	8.0
5	bar	1	1.2
6	bar	2	2.5
7	foo	4	2.3
8	foo	3	8.0

ilarly Intersection and Set Difference
nipulate tables as Sets

Merge or Join

This is how you combine rows across tables, based on some distinguishing element (i.e. ID column). For example, you'd basically take the row with ID 1 in your first table and add all those elements to the row with ID 1 in your second table.

There isn't a graphical example here just because we'll be talking about this a lot more in depth in later lectures. For now, just remember it as a way to combine tables.

ID	A	B
1	foo	3
2	bar	2
3	foo	4
4	foo	3

⋈

ID	C
1	1.2
2	2.5
3	2.3
5	8.0

→

ID	A	B	C
1	foo	3	1.2
2	bar	2	2.5
3	foo	4	2.3
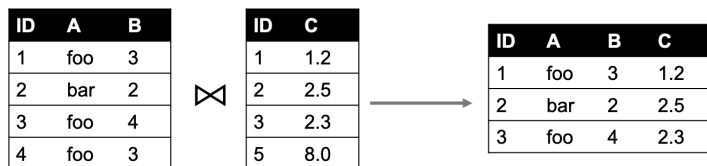
Summary

Overall, **Tables** are a simple and common abstraction. They're how we mainly keep track of data when we do most of these data science things, so it's worth learning how to employ them, and what basic operations we can use when manipulating them.

These **operations**, at a glance:

- Select

- Map

- Aggregate/Reduce

- Join/Merge

- Union/Concat

- Group By

Keep in mind that tables are an *abstraction*, after all, so these operations may be named different things in the languages you use to manipulate them (example: SQL uses 'join' but Pandas uses 'merge'). That's why its important to not just memorize the names of these operations, but what they actually do. This will prepare you for work with any data table manipulation program.

Pandas

PANDAS is a data manipulation library for Python that's highly optimized for performance. It contains two key constructs, **Dataframes** and **Series**.

Dataframes

This is PANDAS's way of representing the table abstraction we were looking at. Geeksforgeeks even calls this one a tabular data structure. Each column within a dataframe can be of a different data type. Dataframes are mutable - you can insert and delete columns. Each row and column can be access through indexing

There are a lot of PANDAS-specific commands that you'll have to learn to be proficient with these, but Geeksforgeeks is a great reference for them. (You'll find it's pretty easy to conduct all the basic table operations)

Some Things to Note:

- The use of the word `index` for what we called `key`

- Non-unique index values *ARE ALLOWED* (might raise exceptions for certain operations)

Columns - can be extracted as series

Index (ID) Column

Non-unique index values are allowed,
but they may cause issues.
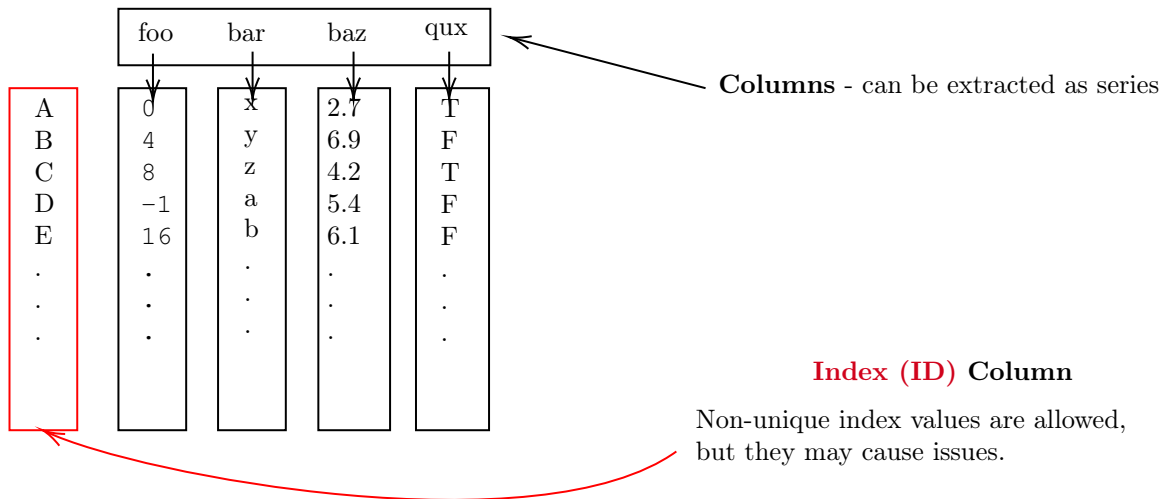
Series

Think of a `series` as a subclass of Numpy's `ndarray`.

Geeksforgeeks calls this a 'one dimensional labeled array capable of holding any data type'. Think of it as a column in an Excel spreadsheet. In fact, their most common use is when you pull a single column out of a `dataframe` and want to analyze it individually.

The object itself supports integer and label based indexing (like letters and does not need to be ordered), and allows us to perform a bunch of operations involving the index. It can be of any data type. It also allows for duplicates, but this would result in reduced functionality.

To create one of these we can grab a column from a `dataframe`, or we can create one out of a regular Python `list` or a Numpy `ndarray`.
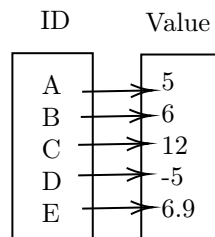
For this sort of stuff, you can probably look at **GeeksforGeeks** for more information. They have excellent documentation on `PANDAS` functionality.

Creating a Dataframe

To **create a `dataframe`**, you have a variety of options.

- Get data directly from a Python dictionary, a bunch of series, or other data structures

- `Pandas.read_csv()` → take in data from a `.csv` file

- `Pandas.read_excel()` → take in data from an excel spreadsheet

26

- `Pandas.read_html()` → take in data from a (static) website (e.g. a website with a big table of data on it)

- From a database by using SQL to make queries

- From clipboard, URL, many more options.

Tidy Data

There are **3** components of **tidy data**: **Labels**, **Variables** (values), and **Observations**. Furthermore, tidy data has 3 rules:

1. Each variable must have its own column

2. Each Observation must have its own row

3. Each value must have its own cell

Here's some elaboration.

- **Variables** → A measure or attribute, e.g. age, weight, height, sex.

- **Value** → Measurement of a *singular* attribute, e.g. `12.2 lbs` or `5.9 inches`.

- **Observation** → All measurements for an object; a *row* in the table. E.g. a single observation in the above table would be `[12.2, 42.3, 145.1]`.

Tidying and melting data basically just means that you mix data around until it's nice and usable. Usually, you are tidying in pursuit of a specific use-case, so less columns or more columns are never the 'better' option. This is one of those things that takes practice and application.

TL;DR Clean up and organize your data before you mess with it!

SQL and Relational Databases

Big Question: What is a relation?
Answer: In a databases context, they usually mean, "a tabular set of data either permanently stored in the database (a table) or derived from tables according to a mathematical description (a view or a query result)." (*Larry Lustig, Stackoverflow*)

> **Definition:**
>
> **Relation** → A relation is a data structure which consists of a heading and an unordered set of tuples which share the same type.
> **Relation Schema** → A list of all attribute names and their domains. E.g. 'The Schema for a Table'.

Indexing

> **Definition:**
>
> **Index** → An auxiliary data structure of a relation database designed to speed up the retrieval of rows.

How can we leverage **indexes** to improve search times for our relational databases? Take a look at the example below. Let's say we wanted to find all people from Canada (with a nat_id of 2).

ID	age	wgt_kg	hgt_cm	nat_id
1	12.2	42.3	145.1	1
2	11.0	40.8	143.8	1
3	15.6	65.3	165.3	2
4	35.1	84.2	185.8	1
5	18.1	62.2	176.2	3
6	19.6	82.1	180.1	1

Unfortunately, the time it takes for us to build this list every time we want to leverage the result of this search is **O(n)**. This is not so great for us. However, if we decide to build an **index** on the 'nat_id' attribute, things change.

loc	ID	age	wgt_kg	hgt_cm	nat_id
0	1	12.2	42.3	145.1	1
128	2	11.0	40.8	143.8	2
256	3	15.6	65.3	165.3	2
384	4	35.1	84.2	185.8	1
512	5	18.1	62.2	176.2	3
640	6	19.6	82.1	180.1	1

nat_id	locs
1	0, 384, 640
2	128, 256
3	512

Now, after establishing this index, which acts like a hidden sorted map of references to a specific attribute in a table, we are allowed **O(log n)** lookup with the parameter nat_id.

You can choose to build an index on a certain attribute to improve search times for it, but they aren't free. They're expensive- establish an index with caution. Not only do they take time to initially build, but now, whenever you add to the table or update it, you also need to update the index. In that sense, establishing too many indexes could lead to other operations, such as changing table values, taking a very long time. It's a delicate balance.

> **Aside: Indexes**
>
> Indexes are actually implemented with data structures like **B-trees**, which is why we are able to perform data access in O(log n) time. The worst case height of a B-tree is O(log n), and since a search is dependent on height, B-tree lookups run in O(log n) on average.

> **Definition:**
>
> **Hierarchical Indexes** → Setting more than one column name as the index.

Relationships

Primary keys and foreign keys determine interactions between different tables. These are formally known as relationships. First, let's establish definitions for primary and foreign keys.

> **Definition:**
>
> **Primary Keys** → Columns whose data can be used to uniquely identify each row in the table. Highlighted in red below. (E.g. the ID column)
> **Foreign Keys** → Columns that refer to the primary key in another table. Highlighted in blue below.

There are four main types of relationships between keys. Here they are with some examples.

- **One-to-many / Many-to-one**: A person can have one nationality, but a nationality can have many people associated with it.

- **One-to-one**: People each have one unique SSN- no conflicts.

- **One-to-one-or-none**: People can either have 1 car, or no car.

- **Many-to-many**: Cats and colors. Red can be on many cats, and many colors can be on a single cat.

Even though this system can sort of be replicated in PANDAS, PANDAS is not strictly a relational data system. Notably, it doesn't have notions of primary or foreign keys built in.

> Do heavy, rough lifting at the **relational database** level, (e.g. when you're deciding what sort of SQL queries to make) and then do the fine-grained slicing and dicing and visualization with PANDAS.

SQL and SQLite

We use **SQLite**, an on-disk relational database management system, in order to interact with our databases. Most **RDMS**s connect to a server, which provides support for better concurrency, that sort of stuff takes longer to setup.

On the other hand, `SQLite` is pretty simple to install via `conda`, so we're going to go ahead and use it.

`SQLite` provides us with two main ways to communicate with the `SQL` database that it maintains. First of all, it gives us a cool GUI-based environment where we can deal with manipulating data manually, but it also allows us to write `SQL` statements to interact with it, whether that be from the command line or from within Python.

Here's an example of how a relational database fits into our workflow.
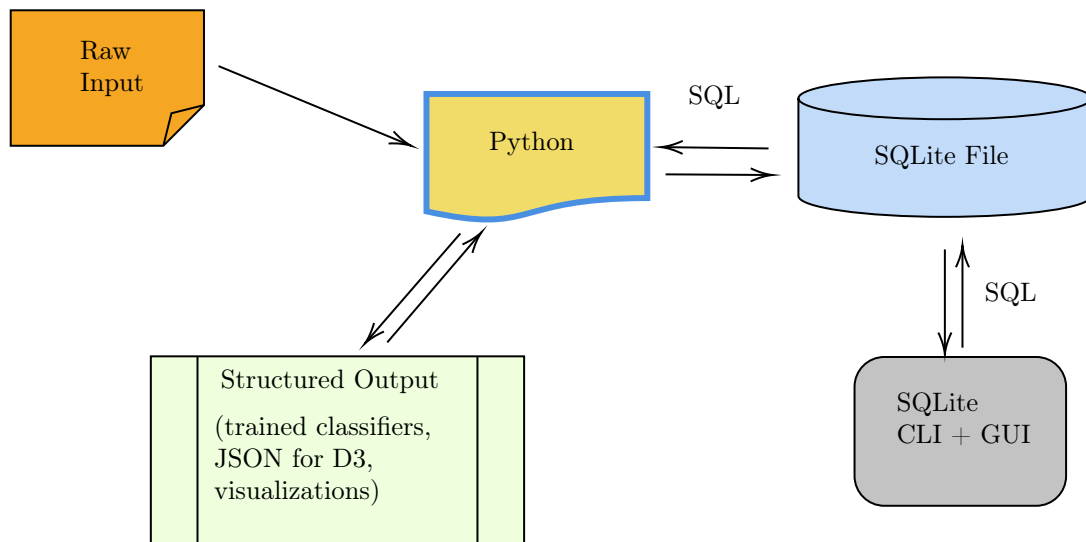
To work with `SQLite` in Python3, simply install and import the `SQLite3` package, and use that.

Joining Data

A **join** operations merges two or more tables into a single relation, based on their columns. There are four total types of join operations.

Formally, the way we format join statements is the following:

```
<type of join> join (<left table>, <right table>) on (<left table column>,
```

```
<right table column>)

Example: Right join (cats, visits) on (id, cat_id)
```

Types of Joins

- **Inner Join** → Returns merged rows that share the **same** value in the column that they are being joined on. Let's say we had the following two tables and wanted to join them.

id	name
1	Megabyte
2	Meowly Cyrus
3	Fuzz Aldrin
4	Chairman Meow
5	Anderson Pooper
6	Gigabyte

`cats`

cat_id	last_visit
1	02-16-2017
2	02-14-2017
5	02-03-2017

`visits`

In order to perform an inner join on these two tables, we need to pick a column to 'join them on'. Here, let's inner join these two tables on `id` and `cat_id`. The result would look like this:

id	name	last_visit
1	Megabyte	02-16-2017
2	Meowly Cyrus	02-14-2017
5	Anderson Pooper	02-03-2017

Inner joins are the most common type of joins; they get us the results that are shared by both tables.

- **Left Join** → A left join gets us **all** the results from the **left** table, but only **some** (the corresponding matching results) from the **right table**. So, what happens if we Left Joined `cats` and `visits` on (id, cat_id)?

id	name	last_visit
1	Megabyte	02-16-2017
2	Meowly Cyrus	02-14-2017
3	Fuzz Aldrin	NULL
4	Chairman Meow	NULL
5	Anderson Pooper	02-03-2017
6	Gigabyte	NULL

You'll notice that the fields that we couldn't fill out get populated with **NULL**.

- **Right Join** → A right join gets us **all** the results from the **right** table, but only **some** (the corresponding matching results) from the **left** table. Here's an example, with updated `cats` and `visits` tables.

id	name
1	Megabyte
2	Meowly Cyrus
3	Fuzz Aldrin
4	Chairman Meow
5	Anderson Pooper
6	Gigabyte

cats

cat_id	last_visit
1	02-16-2017
2	02-14-2017
5	02-03-2017
7	02-19-2017
12	02-21-2017

visits

If we were to perform a right join on these two tables, here's what would happen. It's basically just a flipped version of the left join.

id	name	last_visit
1	Megabyte	02-16-2017
2	Meowly Cyrus	02-14-2017
5	Anderson Pooper	02-03-2017
7	NULL	02-19-2017
12	NULL	02-21-2017

Again, this time, notice how the row entries missing from the left table are now set to **NULL**

- **Full Outer Join** → The full outer join combines the left and right join. It's analogous to a **union** operation. Here's an example of a full outer join of `cats` and `visits` on `id` and `cat_id`.

id	name	last_visit
1	Megabyte	02-16-2017
2	Meowly Cyrus	02-14-2017
3	Fuzz Aldrin	NULL
4	Chairman Meow	NULL
5	Anderson Pooper	02-03-2017
6	Gigabyte	NULL
7	NULL	02-19-2017
12	NULL	02-21-2017

Syntax in PANDAS

Here's how to write some basic join syntax in PANDAS. This should be easy once you learn how to phrase join statements- you're basically just translating it into code.

First, this is how you'd read from SQLite (or any other database you're hooked up to) using Pandas and generate the appropriate dataframes to work with.

```
1  # establish dataframes from SQL
2  df_cats = pd.read_sql_query("SELECT * from cats", conn)
3  df_visits = pd.read_sql_query("SELECT * from visits", conn)
```

Now, here's how to do the joins.

```
1  # inner join
2  df_cats.merge(df_visits, how = "inner",
3                left_on = "id", right_on = "cat_id")
4
5  # left join
6  df_cats.merge(df_visits, how = "left",
7                left_on = "id", right_on = "cat_id")
8
9  # right join
10 df_cats.merge(df_visits, how = "right",
11               left_on = "id", right_on = "cat_id")
12
13 # full outer join
14 df_cats.merge(df_visits, how = "outer",
15               left_on = "id", right_on = "cat_id")
```

There are also ways you can perform most of these joins via SQL (save for the right join), but I would prefer doing them from Python. As such, I won't include the SQL syntax here.

Aside: Raw SQL with Pandas

If you want to use raw SQL queries to interact with PANDAS dataframes, you are free to do so when you use the PandaSQL library.

Visual Example

Here's a neat way to visualize joins using Venn Diagrams.

INNER JOIN

FULL OUTER JOIN

LEFT JOIN

RIGHT JOIN
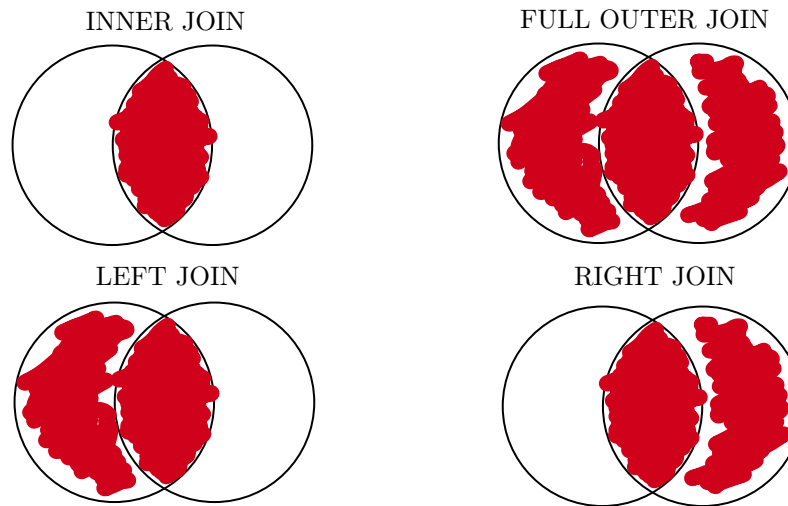
Prediction vs Inference

Drawing useful conclusions from data is an important skill. This can be done by identifying patterns in information, using visualizations, and bringing data together.

> **Aside: Prediction and Inference**
>
> **Prediction:** Given what I've seen, what is the most likely value I'll see in the future?
>
> - Predictions forecast the most likely values of the data cominng from the data gnerating process.
>
> - Making Informed Guesses
>
> - Uses Machine Learning and Optimization
>
> **Inference:** How likely is what I observed representative of the broader picture?
>
> - Statistical inference draws conclusions (with confidence) about the structure of the data generating process (population).
>
> - Quantifying whether those patterns are realizableUses randomization
>
> Notes: **Inferences** learn about the data generation process while **Predictions** predict what will come next.

Introduction to Models

A **Model** is a theory or explanation of the data generationn process (relationship). A **True Model / Population** is a phenomemon under investigation. Models can be useful when it comes to analyzing and visualizing data.

Fit Model: An instance of a model that is likely to explain a fixed dataset.

- Finds the most likely parameters that explain the observed data under the given model.

- Parameters are found by minimizing a loss function - typically some notion of 'error' or 'cost'.

Note: Model is effective if it captures the general understanding of how Data processing behaves and describes the data well.

Statistical Model: An instance of a model that is likely to explain a fixed dataset.

- Function that measures properties of X

- Example: is there a linear relationship between the heights of children and the ehigh of their buological mother?

Prediction Model Regression: attempts to predict most likely quantitave value associated to an observation (set of input features).

- Regression results in having a model that we can use to predict a numerical value for data that we have not seen yet

- Example - Give the heights of a child's parents, what is the height of their child?

Prediction Model Classification: attempts to predict most likely class associated to an observation (set of input features)

- Example - Given product purchase attributes (item, price, age, state), can one preduct whether the person was satisfied with their purchase?

- Classification results in a model that we can use to preduct labels for data we have not seen

7 Version Control Software (L7)

This lecture focuses mainly on **version control** and `Git`. Since I know the basics of this stuff already, this will just be a smaller review of `Git` plus the stuff I didn't already know very well coming into this course.

> **Big Idea:** Teams needed good ways to maintain central repositories for their code projects without having conflicting versions of their code, so **version control software** was created.
>
> Eventually, this software carried the secondary purpose of tracking and managing bugs. These days, however, dedicated enterprise tools like **JIRA** also exist to handle bugs. It's mainly based on what your company decides to use.

Version Control Software is used to do the following:

- Search revision history and get **any version** of the project you're working on

- Share code changes with your collaborators

- Confidently make changes to large files

Centralized VCS

People used **Centralized VCS** to have multiple users all pushing towards a central repository. I've seen examples of this in use at HPE (`SVN`- Subversion), and during my classes at UMD- `CVS` in CMSC132.
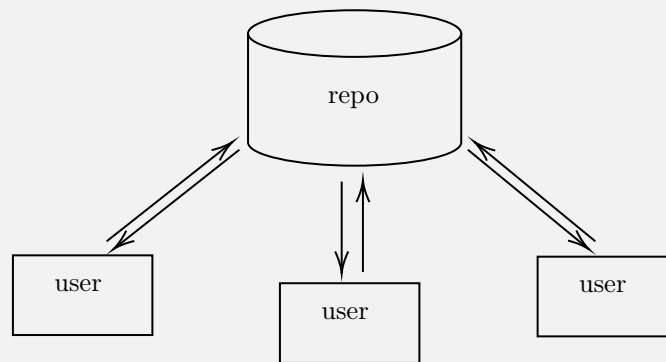
In this case, there's a singular centralized codebase that users will all be contributing to at the same time.

Distributed VCS

Distributed VCS has no central repository, and every repository has their own commits and history. Examples of this would be **Git** and **Mercurial**.

> **Aside: `Git`'s popularity**
>
> `Git` is currently the most widely used code management tool (VCS). The next two in line are `SVN` and `Mercurial`.

`Git` is also more efficient and secure than `SVN`, but a lot of old legacy codebases and companies still make use of `SVN`. For that reason, it may be good to pick up some of the basics of `SVN` so I'm not totally unfamiliar, but `Git` generally seems to be preferable.

Branching

`Git` also allows for branches. **Merging** also ties in closely with branching on git, and has a lot more sophistication in `Git`.

When Should I Branch?

Anything in the master branch is considered to be deployable. If you're adding a new feature, working on an experiment, or trying to implement a new fix, make a branch. You can always merge it back to master once it's considered 'deployable' again.

Git Basics

As far as `git` is concerned, a file has 4 states:

- **Modified** → File has been changed, but those changes have not been committed

- **Staged** → Marked to go to next commit snapshot

- **Committed** → Safely stored in local database as part of a 'commit'

- **Untracked** → News added or removed files

This idea can be complemented by a visual guide, where you can see the three main 'places' that a file can be within the `git` system.
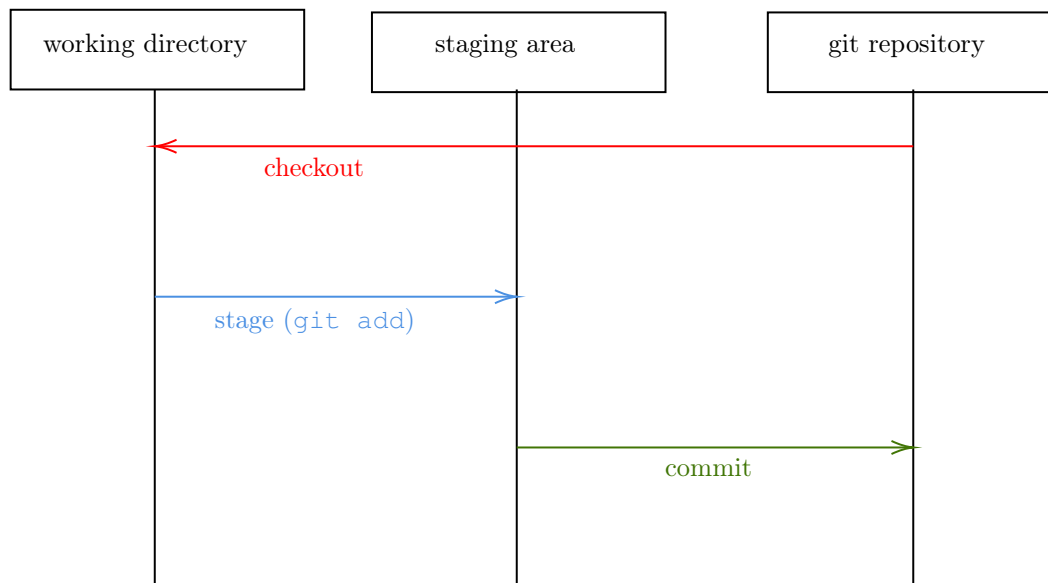
Online Hosting

Github, **Bitbucket**, and **Gitlab** are all popular sources that will host your git repositories. Think of this as just another place for your git stuff to exist, except whenever you want to update the main online repository, perform a `git push`.

It's sort of like a hybrid of a centralized and distributed VCS. According to a website online, "GitHub and similar services bring all of the benefits of a decentralized VCS to a centralized service."

8   Missing Data and Imputation (L8)

Missing Data

Missing data is information that we want to know, but don't know. It can come in many forms. Here are some examples.

- People omitting answers on surveys
- Inaccurate measurements that we need to discard- we're mainly talking about easily detectable outliers here
- Canceled trials of an experiment

To do something about this, however, we need to figure out the following.

- What contributes to the *probability* of a data point being absent?
- Can this missing data be interpolated using the data we already have? Or not?

Just Deleting It

The easiest way to deal with this is just to delete all the tuples with any missing values, so we don't have any 'incomplete observations'. All we have to do in this case is just use `df.dropna()` to trop the appropriate row.

Be warned that a loss of a sample could lead to a variance greater than what's reflected by the size of our data. This could cause bias. Overall, despite this being the easiest way to get rid of 'problem' data, we should avoid doing this if we can, and intelligently account for it being missing if possible. Obviously, if we want to remain the most accurate that we can possibly be, the goal is not to throw in the towel and just toss out missing data right off the bat.

Types of Missing Data

Let's start by classifying missing data into a variety of different types. Missing data can fall into one of three categories. They also have commonly used abbreviations, included here.

- Missing Completely at Random (**MCAR**)
- Missing at Random (**MAR**)
- Missing Not at Random (**MNAR**)

Missing Completely At Random

Missing completely at random means exactly what it says- the data that has gone missing has gone missing completely and entirely at random- there is no rhyme or reason behind what's gone and why it went.

> **Example of MCAR:** Imagine you are doing an experiment on plants grown in pots, when you have a nervous breakdown and destroy some of the pots. You didn't have any bias in how you picked the pots to break, so this data is now **MCAR**.

However, this just isn't realistic. Data is usually missing for a reason. For example, if you're standing outside CSIC polling people for a survey and you suddenly ask for their grades, students who are failing may be less likely to reveal their grades than students who are doing well.

Missing At Random

For data that is missing at random, the probability of the missing data is dependent on the observed data, but not the unobserved data.

There is a **probabilistic mechanism** associated with whether the data is missing, and that mechanism takes the observed data as input.

> **Example:** If a child does not attend an educational assessment because the child is (genuinely) ill, this might be predictable from other data we have about the child's health, but it would not be related to what we would have measured had the child not been ill.
>
> Since we could predict the "missing-ness" of the student's score from the student's health, this is considered **Missing At Random**. *(Taken from Martin Bland's textbook: An*

We can model a parameter's "missing-ness" on other properties of the data we have already.

Missing Not At Random

The "missing-ness" of a variable has something to do with the variable itself.

> **Example:** If men failed to fill out a survey because of their level of depression, their depression affects the measurement that is being taken about their depression, making this data **Missing Not at Random**.

Line of Best Fit

In order to aid in this, let's again revisit the idea of a 'line of best fit' from statistics. Here's a quick review.

> **Definition:**
>
> **Line of Best Fit** $\rightarrow$ When the difference between the true y-values and the line that you're using the estimate y-values is minimized.

This is accomplished by using the least squares method. Below is a visualization of the process. Given the following plotted points (light green), we're looking to create the line of best fit (blue).
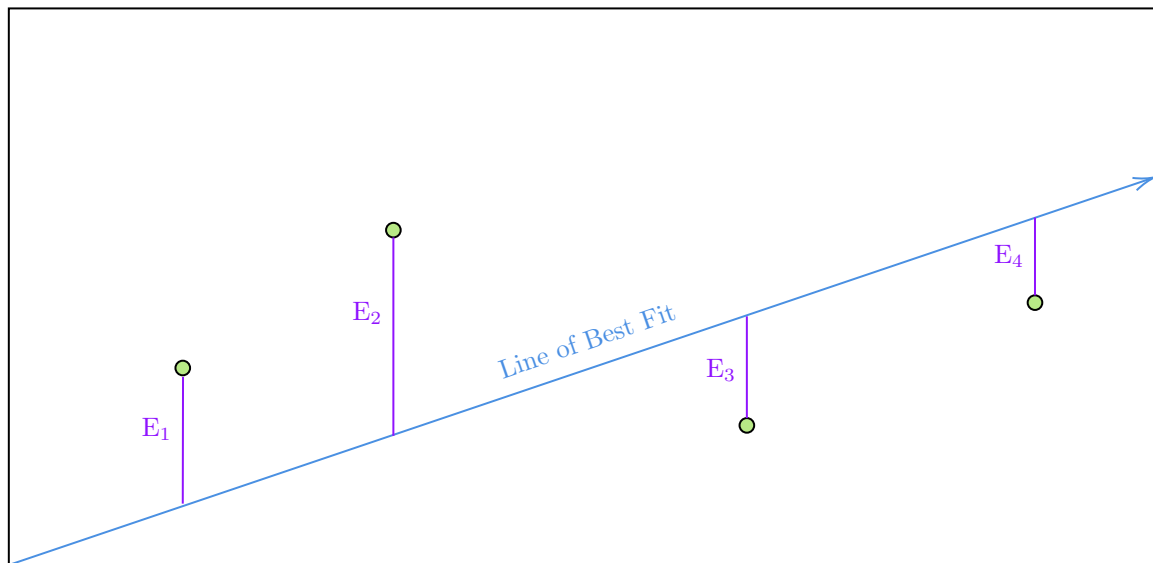
Here, using the least squares method, we try to minimize the equation yielded by the following summation.

$$\sum_{i=1}^{n} (E_i)^2 = (E_1)^2 + (E_2)^2 + ...(E_n)^2$$

Imputation

How do we handle missing data? One excellent way to do that is **imputation**.

Definition:

Imputation $\rightarrow$ Replacing the missing data with substituted values.

Imputation is basically just a fancy way of saying that you'll replace whatever's missing with other data that you generate or collect yourself.

Types Of Imputation

- **Mean Imputation** $\rightarrow$ Imputing the average from observed cases for all missing values

- **Hot-Deck Imputation** $\rightarrow$ Imputing a value from another subject, or *donor*, that is most like the subject.

- **Cold-Deck Imputation** $\rightarrow$ Bringing in other datasets

- **Other Methods** $\rightarrow$ Regression imputation, stochastic regression imputation, interpolation and extrapolation.

Single vs. Multiple Imputation

Single Imputation is supposedly 'old and busted', while **Multiple Imputation** is the 'new hotness'. Multiple imputation is basically when you impute on the original data n times, then you perform analysis on each of the resulting n datasets, then pool all those results into one, distilled result.
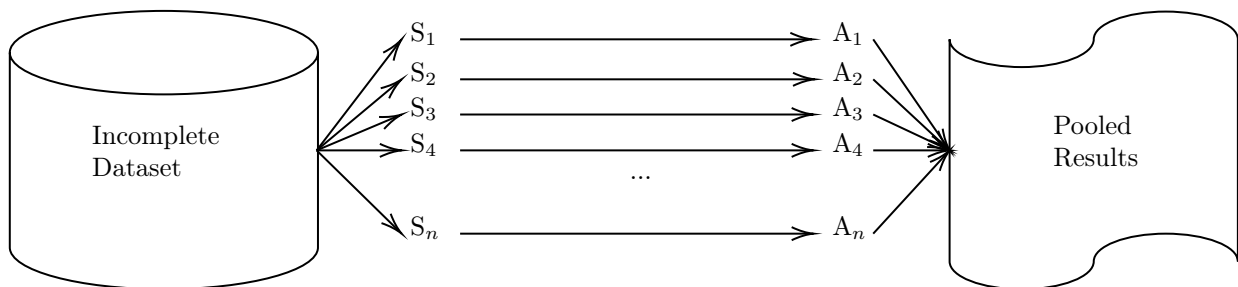
In this case, pooling the data is just taking an average. We can see that this 'multiple imputation' model relies on the results of each imputation being distinct.

Now, the question arises: **How big should n be** when we're talking about multiple imputation? This ultimately depends on the **size of your dataset** and the **amount of data missing**.

Recent research suggests, however, that a 'good' n value is generally higher. General counsel here is to choose a pretty sizable value for n.

9 Data Wrangling, Integration, Cleaning (L9)

Now that we have imputed and made some new datasets, how do we make use of them? To clarify from the previous example, we can simply perform analysis on each **individual dataset**, then pool the

results of that analysis (take the average of the analyses on the imputed datasets).

Bayesian Imputation is another way to handle imputation, this time using Bayes' theorem. A google search doesn't yield much on this, but I assume it's got something to do with a Bayesian attempt at imputing data.

Data Wrangling

> **Definition:**
>
> **Data Wrangling** → Getting data into a structured form suitable for analysis.
>
> A.k.a. **data preparation**, **data mining**, or the **Extract → Transfer → Load (ETL) Process**.

Supposedly, 80 to 90% of your time is spent on data wrangling.

Key Steps of Data Wrangling

- **Scraping** → Taking info from sources. The process of getting your data from its unusable format into Python data structures.

- **Transformation** → Tidying it into the right data structure. (e.g. converting `string`s that represent the date into `datetime` objects)

- **Information Extraction** → Extracting structured information from unstructured text and/or sources.

- **Data Integration** → (If applicable) combine info from multiple sources so that it all plays nice together.

- **Cleaning** → Removing inconsistencies + errors.

Usually, there are a 'mishmash' of tools that are used for all these processes, but there are generally accepted tools that are used to 'pipeline' all of these operations so that they work smoothly.

For example, some popular ETL services are **Talend** and **AWS Glue**.

Data Integration

> **Definition:**
>
> **Data Integration** → Combining data from different sources into one unified view, and allowing for structured querying and analysis on that data.

Each data has a **local schema**. We want to bring it all under one, unified, **global schema**. As it happens, there are actually two main ways that we can accomplish this.

1. Bring all the data into one single data structure or repository. This is known as **data warehousing**.

2. Keep the data where it is, in separate repositories, and send **queries** to our separate data when we need to.

Option 1 is pretty self explanatory- you're just mushing all the data together, and with a little effort, you can ensure that the format of your data is all compatible, and combine it. Option 2 is a little tougher to think about at first, so here's a diagram.
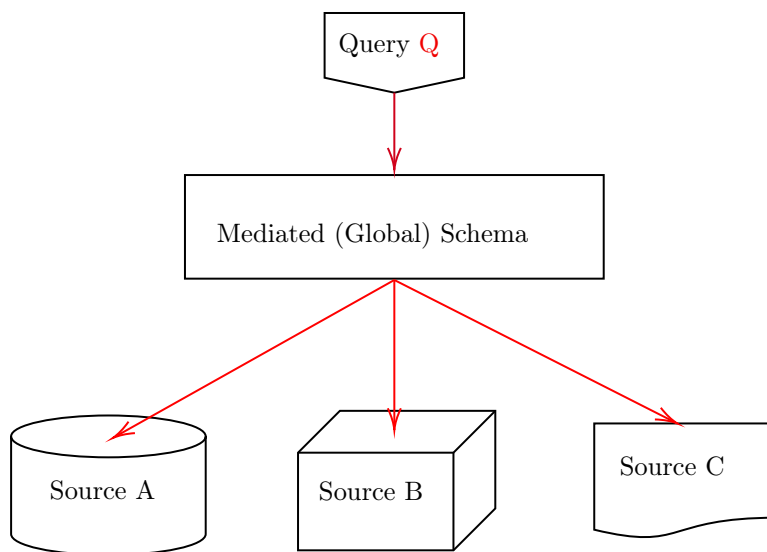
Though it is **less efficient**, this setup is preferable when the data is **dynamic**.

For example, if you needed to make a query on the sum total of the data in Sources A, B and C, and data source B was constantly updating, there's no real way you'd be able to warehouse that data.

We usually move to keeping our data separate and maintaining only a global schema when data warehousing is unfeasible.

Data integration has key challenges associated with it. For example integration, cleaning, setting up a global schema, and limits on how many time you can access a source (e.g. API request limits)

Definition:

Schema Matching → Constructing a global schema that allows all the data you're accessing to 'mix'. (Different sources label their data differently!)

Big Idea: Data integration is still a pretty ad-hoc and manual process as of right now, but it is a growing area of focus in this relatively new industry.

Data Cleaning

Data cleaning is the process in which we deal with data quality issues.

Types of Data Quality Issues

- **Single Source Problems** → These are problems you run into when you work with a single source. Databases tend to force constraints on their data, so we can trust them sometimes. There is **no Quality Control** for data scraped from files, spreadsheets, or webpages. As such, we're susceptible to things like **ill-formatted or missing data**, **contradicting data**, **illegal entries**, and the like.

- **Multi Source Problems** → More problems arise when you try to integrate multiple sources. Different sources are maintained by different people, so we can already see a basis for issues.

  1. Mapping Info
     * Naming conflicts: The same name is used for different objects.
     * Structural conflicts: different representations across sources.
  2. Entity Resolution
     * Matching entries across sources may prove to be difficult.
  3. Data Quality Issues
     * Contradicting information, mismatched information, etc.

Outlier Detection

Univariate outliers are detected by using the **MAD**.

> **Definition:**
>
> **Median Absolute Deviation (MAD)** → Median distance of all the values from the median value.

If your data is **normally distributed**, we can assume that any and all data farther than `1.4826` MAD from the median are considered outliers.

> **Aside: Curse of Dimensionality**
>
> The "Curse of Dimensionality" states that methods to find outliers break down as the dimensionality of the data increases. We always have the option of **projecting** this data down to lower dimensions and working with that, but that's the best we can do. Usually, it isn't that straightforward.

Entity Resolution

> **Definition:**
>
> **Entity Resolution** → Determining when references to data are equivalent. (E.g. Bob Spence, B. Spence, and Robert Spence *could* all be referencing the same person)

When it comes to working with entity resolution in practice, there are three main issues to get over. First, let's informally define a **mention** as when some object gets referenced.

- **Deduplication** → Cluster records/mentions to the same entity and create a **cluster representative**.

- **Record Linkage** → Matching records across different databases. (e.g. finding the same person on FB and Google Plus)

- **Reference Matching** → Matching mentions of an entity to the actual entity being mentioned.

Actually finding similarity between two references/mentions can be done in many ways. For example, one can use 'edit distance' (string dissimilarity) functions, set similarity, vector similarity, etc. Generally, this must be optimized for each dataset.

For more advanced examples, the keyword to search the web for is **entity resolution algorithms**. The examples in the slides look to be fairly basic and straightforward.

10 Statistics Review (L10)

This lecture is a summary of the 'important stuff' from STAT400, and then some. Worth going over, it's been a long time since statistics.

Exploratory Data Analysis

Exploratory Data Analysis is fancy talk for when you're getting a feel for what's going on with the data. We can usually spot certain nuances in the data (skews, trends, interactions of variables) that will help suggest what assumptions to make and what approaches to take when performing actual analysis.

Summary Statistics - Overview

Part of **descriptive statistics**, summary statistics is basically exactly what you think it is- it's used to summarize data.

The big idea here is to convey information with extreme simplicity. This includes the following:

- Measures of Location

- Measures of Dispersion

- Correlations

Measures of Location

There exists a certain distribution of values, and we are interested in the center of said distribution. In other words, we are interested in **measures of central tendency.**

> **Definition:**
>
> **Measure of Central Tendency** → A measure of **central tendency** is a single value that attempts to describe a set of data by identifying the central position within that set of data.

There are two measurements that we use for this business: **sample mean** and **sample median**. They look similar, and they measure the same thing.

We also know that these two measures **differ predictably** when the data **aren't symmetric**.

The idea of **weighted average** is when you give more power to larger contributions. For example, bombing a midterm will hurt your grade average more than bombing a homework assignment.

It's interesting to note that **extreme observations** will skew the mean, but not the median. For example:

> The **median** of [1,2,4,6,8,9,17000] is 6
> The **mean** of [1,2,4,6,8,9,17000] is 2432.8

Skewed Data occurs with the data is not symmetric in which case the mean does not equal the median.

- Left Skewed $\rightarrow$ Mean < Median

- Symmetrict $\rightarrow$ Mean = Median

- Right Skewed $\leftarrow$ Mean > Median

The mean will exceed the median when the distribution is skewed to the right. Skewness is in the direction of the long tail.

Measures of Dispersion

So how do we measure **dispersion**? We know we can't use the range to do so, as shown by the below example:
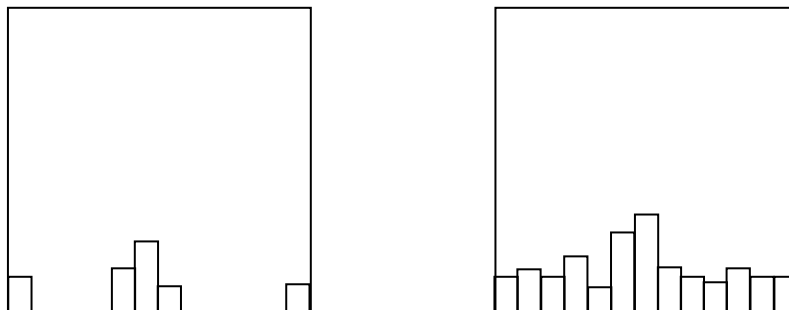
Ex) These two datasets have the same range, but highly different dispersions

As such, **Variance** and **Standard Deviation** are univariate measures of dispersion in a dataset.

Variance measures how far a set of numbers is spread out from their average value. It can be calculated using the following formula:

$$\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

In this case, x_i represents each data point, while $\bar{x}$ represents the sample mean.

However, the units get messed up when we calculate variance. Thus, **standard deviation** is an **interpretable** unit of measurement for dispersion.

> ### Aside: Variance
>
> Dividing by n-1 in the *sample variance* leads to unbiased estimate of the population variance. Sampled values are on average closer tot he sample mean than to the true population, so we will underestimate the true variance slightly. The n-1 makes the variance calculation slightly bigger. This is **embiggening** which impacts smaller n more than a larger n. *If sample IS the population then just divide by n*

We calculate the standard deviation by simply finding the square root of variance. The formula is as follows:

$$\sqrt{\frac{1}{n} \sum_{i=1}^{n} (x_i - \bar{x})^2}$$

> ### Aside: Bessel's Correction
>
> **Bessel's Correction** is the formal name for when we use $(n-1)$ instead of n when calculating sample variance and sample standard deviation, where n is the number of observations in the sample. This method apparently corrects bias in the estimation of the population variance.

Standard deviation can also be used as a unit. For example, 68% of the distribution will be within 1 standard deviation from the mean, 95% will be within 2 standard deviations from the mean, 99.73% in 3 standard deviations, and so on. (Just like STAT400)

Correlation

> ### Definition:
>
> **Correlation** $\rightarrow$ The idea that variables x and y vary together.

Causality vs. Correlation: Can we figure out if movement in x induces some sort of movement in y?

> **Example:** If the following scatter plot suggests correlation, **linear regression** can actually measure that correlation.
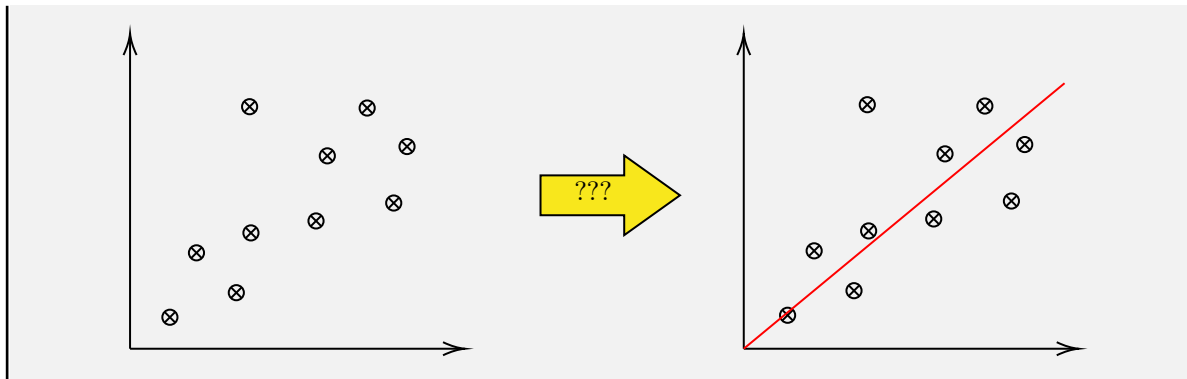
However, be warned. **Correlation does not imply causation!** Just because two things vary at similar rates does not imply that one affects the other!

Standardization

Say we find out our variables are skewed and have different ranges. We should **transform** the variables to ease data analysis and allow the use of other statistics and/or machine learning models.

This can be done with **Standardization** (3 types).

- **P-Standardization** (Percentiles)

- **Z-Standardization** (Z-score)

- **D-Standardization** (Dichotomization)

You should **always** standardize.

> **Example:** when averaging multiple variables, like creating a 'socioeconomic status' variable out of a combination of age, income and education.

P-Standardization is sometimes called a 'percentile score'. Every observation is assigned to a number between 0 and 100, indicating the % of observations 'beneath it'.

Special examples of this: median, quartiles, quintiles.

This process is known as transforming the variable into an **ordinal variable**.

You can also group data into 'standard deviations from the mean', like we do with exam scores.

It's important to note which of your datasets uses **discrete** or **continuous** variables, because some models only work on one type of dataset or another, so we should keep that in mind.

11 Networks and Graphs (L11)

Definition:

Network → a system of interconnected objects, often represented by a graph.

Review on Graphs

A **graph** is a data structure describing a set of vertices and edges. It can either be **weighted** or **unweighted**, **directed** or **non-directed**.

You can store graphs in a few different ways. They also are mathematical models used to represent networks. We will apply algorithms on graphs to answer questions about real-world networks.

Adjacency List
(For each vertex, store a list of the vertices it connects to)

Pros:

- Easy to add edges

Cons:

- Deleting is hard

- Checking for edges is `O(|v|)`
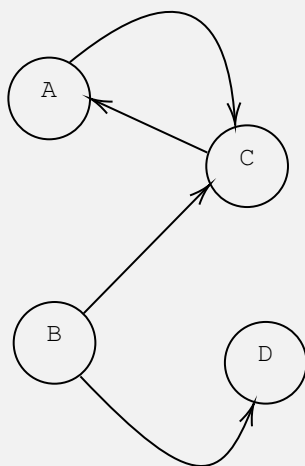
Vertex	Neighbors
A	[C]
B	[C, D]
C	[A]
D	[]

Adjacency Dictionary
(For each vertex, store a dictionary of vertices that it connects to)

Pros:

- O(1) to add, remove and query edges

Cons:

- Memory + Caching overhead (heavy use of Python data structure)

Neighbors for B → `{C:1.0, D:2.0}`

The previous two are pretty much similar, it's just that the Adjacency Dictionary takes advantage of features provided to us in Python, while the Adjacency List is the more classic way to represent a graph. However, the reason we talk specifically about the adjacency dictionary will be explored below.

Adjacency Matrix
(Store the connectivity of the graph in a matrix- almost always in a sparse matrix)

Pros:

- `O(1)` time for query, add edge, remove edge

Cons:

- `O(`v^2`)` space complexity, regardless of the actual number of edges

NetworkX

`NetworkX` is how data scientists work with graphs in **Python**. It makes use of an **Adjacency Dictionary** representation, which is essentially just a beefed up Adjacency List with better access times and more overhead (see above).

Graph Databases

Graph databases exist and are an alternative to relational databases.

Example: A very simple graph database. Note the similarities to a relational database. Here, the different relations like '`is_friend`' and '`read`' can be represented with, for example, separate adjacency dictionaries.

NAME: Lizzy

NAME: John P.

is_friend

is_friend

read

read

TITLE: Bob's Adventure
AUTHORS: Tim Lane, Jack Hart
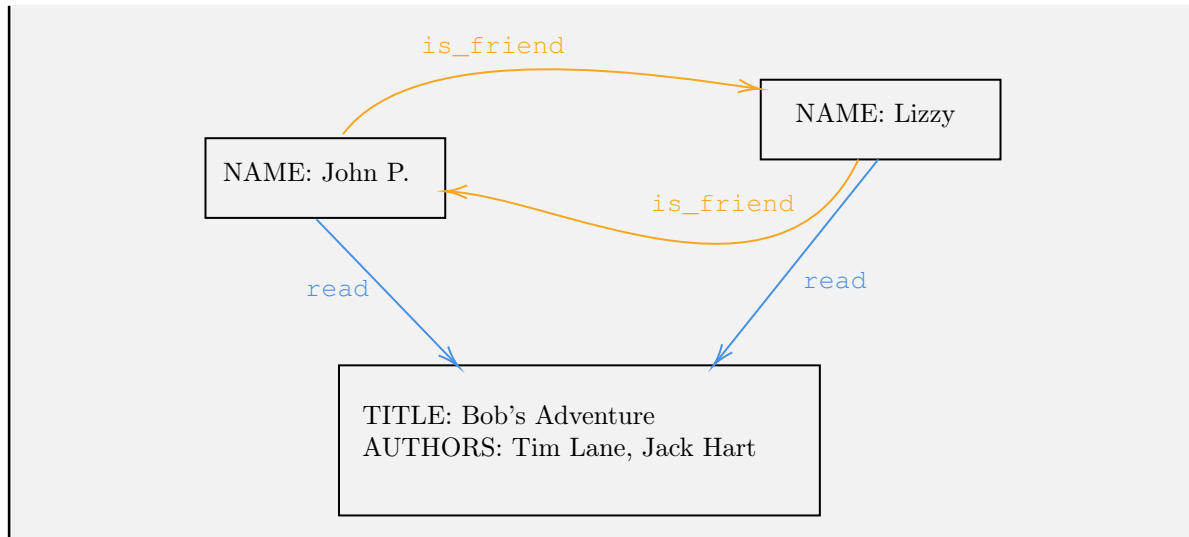
To **query** a graph database, we need to use a language other than SQL.

Neo4J and **Titan** are popular graph DB solutions, and can be accessed and leveraged using **Bulbflow**, a Python library.

Centrality Analysis

Centrality Analysis is the idea of finding out the most important node(s) in one network- not all nodes are created equal. This is important in visualization and classification. There are four distinct types of centrality analyses.

- **Degree Centrality** $\rightarrow$ The importance of a vertex is determined by the number of vertices adjacent to it. In other words: the larger the *degree*, the more important the vertex. To find normalized degree centrality, just divide (degree of node) by (total nodes - 1).

- **Closeness Centrality** $\rightarrow$ The importance of a vertex is measured by how close a vertex is to other vertices. For example, to determine this, we can calculate the average length of all the shortest paths from that one node to every other node in the network. If needed, we can factor in edge weight.

- **Betweenness Centrality** $\rightarrow$ How "in the way" is this node in terms of shortest paths between nodes?
$$\frac{\sigma_{st} v_i}{\sigma_{st}}$$
  To find this, we just need to divide the shortest paths passing through v_i by the total shortest paths in the graph (as shown above).

- **Eigenvector Centrality** $\rightarrow$ A vertex's importance should be determined by the importance of the friends of that vertex- i.e. *If one has many important friends, they too, are important.* A variant of eigenvector centrality is Google's PageRank algorithm's scoring system.

All of this and more are implemented in NetworkX.

Network Topology

We want to learn from the topology of a network- which is basically its broad, overarching structure.

<div>

Definition:

Bridge → An edge is a bridge if its removal results in total disconnection of its terminal vertices. Bridges connecting two different communities are weak ties.

Community → A community is a tightly knit region of a network. We can use bridges to help define how communities are separated.

</div>

The **Girvan-Newman Method** is a way to flesh out tightly-knit communities in networks. You basically remove the edges of a higher betweenness, then keep repeating this process. (This functionality is found in `NetworkX`).

You can also use `GraphViz` (like in CMSC330 with the regex project) to display your graphs and networks.

12 Natural Language Processing (L14 + 15)

<div>

Definition:

Natural Language Processing → Natural language processing is a subfield of linguistics, computer science, information engineering, and artificial intelligence concerned with the interactions between computers and human languages, in particular how to program computers to process and analyze large amounts of natural language data. *(Wikipedia)*

</div>

It appears that this field is the intersection of **linguistics** and **computer science**. It's worth picking up further linguistics knowledge if pursuing NLP.

There are many uses for Natural Language Processing (NLP) software, but here are some good examples.

- **Review analysis** is something that a lot of marketing and product companies have to do- they don't want to manually sift through every review of their product and figure out if the people like it or not, so they assign software to do it!

- **Language translation** is another excellent example. Language translation isn't as simple as swapping out words for one another, extra processing has to be done. This falls under the realm of NLP.

- **Question and answer**, like *Jeopardy*, requires a fair amount of natural language processing. When IBM's Watson appeared on *Jeopardy*, programmers mentioned that he had to distinguish between simple 'factoid questions', which could just be answered via a simple fact lookup, and 'thinking questions', which usually required the answer-er to follow some sort of narrative and do some critical thinking.

Background

Before the 1980s, Natural Language Processing was widely based on sets of hand-tuned rules.

After the 1980s, however, machine learned seeped into the realm of natural language processing.

This started with decision trees then went on to leveraging **hidden markov models**. Hidden markov models are based on augmenting **Markov Chain**s.

Aside: Markov Chains

A Markov chain is a mathematical system that experiences transitions from one state to another according to certain probabilistic rules. The defining characteristic of a Markov chain is that no matter how the process arrived at its present state, the possible future states are fixed. It looks kind of like an NFA.

Excellent further information and an example found at:
https://setosa.io/ev/markov-chains/

Then, people shifted to using statistical model for language. Now, people have finally come to using mostly unsupervised or semi-supervised learning for models.

In data science, lots of data comes in the form of **unstructured text**. We're basically defining that as text that cannot be predictably processed. The following are examples of chunks of text that could carry unstructured, yet valuable data.

- Facebook posts
- Amazon reviews
- Wikileaks dumps

Understanding the English language is hard, as it's fairly hard to predict which bits of sentences mean what. Specifically, we can say that **structure can sometimes be ambigious**.

Aside: Windgrad Schema Challenge

Proposed as a complement to the turing test, the Windgrad Schema Challenge involved asking people (or computers) to pick out the antecedent of an ambiguous pronoun.

> **Example:** The city **councilmen** refused the **demonstrators** a permit because they [*feared/advocated*] violence.

Here, we know that **feared** corresponds to **councilmen**, while **advocated** corresponds to the **demonstrators**. It's fairly easy for a human to deduce this, byt the guy who hosted this challenge argued that understanding this requires 'more than NLP', and that such a task required some type of common sense and **deep, contextual reasoning**.

NLP is a pretty broad field- we're starting this with sentiment analysis.

It can be argued that we don't need to fully understand the text to determine the general sentiment. For example, if you were to scroll through Amazon interviews, you can sort of tell which ones are glowing ones and which ones are very negative.

The main idea is to look for **signals** in the text that tell us the sentimental value of that text.

Natural Language Processing Terminology

Some key terms used in industry.

- **Documents** → Groups of free text- remember that this definition is sort of distinct from the idea of 'word documents' (common confusion)

- **Corpus** → A collection of documents

- **Terms** → Individual words (usually delimited by whitespace or some equivalent)

- **Syntax** → Refers to the grammatical structure of language

- **Semantics** → The study of meaning of language

- **Tokenization** → Splitting sentences into tokens (similar to how compilers 'lex' stuff). There's almost certainly a library for this process.

- **Stemming** → Finding roots of words, e.g. turning *organizer* into *organiz*. Also written as: 'The crude chopping of 'affixes'.

- **Lemmatization** → Reducing inflections on variant forms back to 'base form', e.g. *are, am*, and *is* become *be*.

- **Morphological Segmentation** → A process that aims to break words into meaning-bearing morphemes. (Morphemes are the smallest meaningful units in a language.) How words are formed & relationships of different parts of the words.

- **Parts of Speech (POS) tagging** → Finding parts of speech of a word. This is a key process in producing parse trees for sentences.

- **Parsing** → Making a parse tree for a sentence, based on parts of speech. Again, just like 330s compiler project.

- **Information Extraction** → Turning unstructured text into 'structured sequences', or usable information.

- **Named Identity Recognition** → Is when you identify key entities in text, like locations, times, people, dates, etc.

- **Sentiment Analysis** → Deciding if opinions are good or bad. Lots of ad and marketing companies use software like this.

Spoken Dialogue System (Example)

Below is an example of a spoken dialogue system.

Phonology	Morphology	Syntax	Semantics	Reasoning
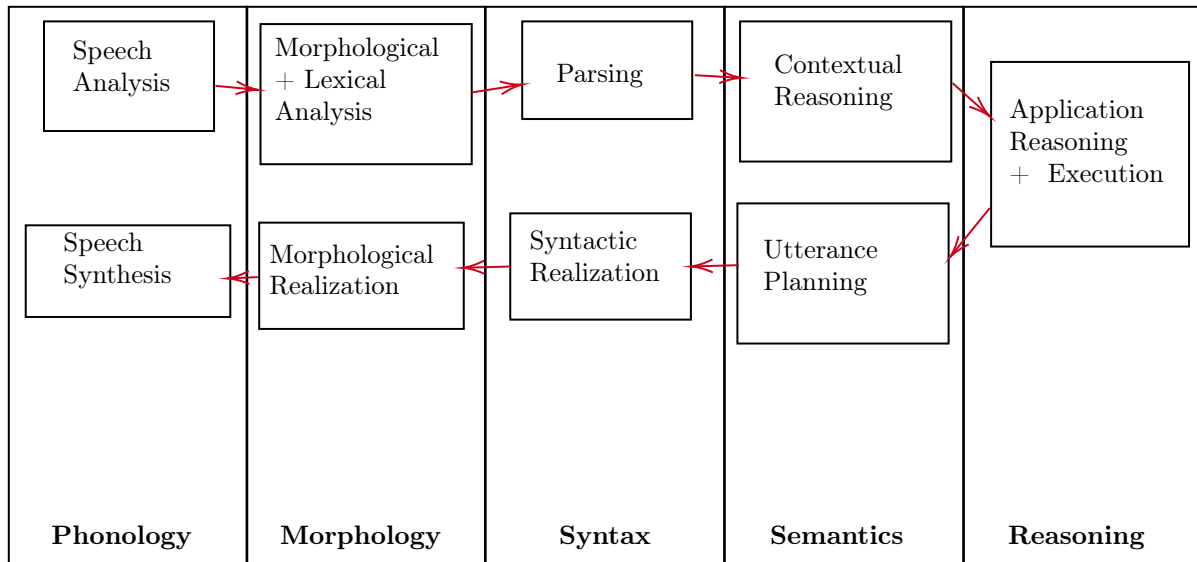
Notice the **complexity** and **number of components** needed for a process like this, even though the system is seemingly simple.

The issues involved in translating human language to objective data are immense, and so our systems built to tackle these tasks are fairly involved.

General Industry

These days, people in industry are working on a bunch of NLP-related interesting problems.

- Speech recognition

- Caption generation

- Natural Language Generation

- Optical Character Recognition

- Word Sense Disambiguation

Text classification is a common subset of problems in this field, for example. For example, we can take a look at an email spam classifier.
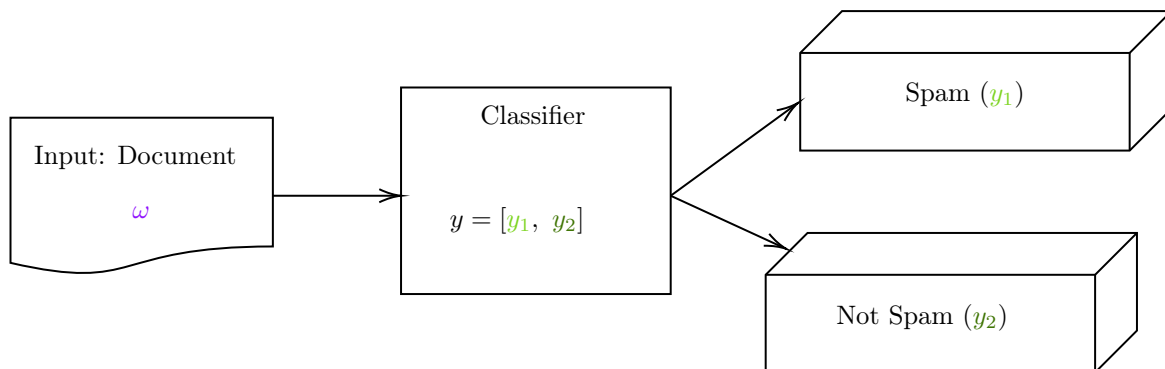
In the diagram above, general ML naming schemes are also used. More on this in ML + NLP classes (and later lectures).

Aside: Hardcoded Rules

It was, at a time, the only option to use hard-coded rules to accomplish the tasks that we allocate to modern NLP techniques. We have some pros and cons for this approach.

- **Pros** → Plenty of domain-expertise and customizability, human readable (we can easily understand the motivation behind decision making)

- **Cons** → Brittle, expensive to maintain, and extremely hard to generalize. (Most customization is very domain-specific)

Supervised Learning

Supervised learning is one such common method to escape the issues that we find in the hardcoded rules mentioned in the Aside above. Although we can get more generalized solution capability, we trade that for transparency. The algorithm that decides where an object should be classified essentially becomes a black box.

Let's say we have the following as **input**:

- A **document**, ω

- A set of **classes**, $y = \{y_1, y_2, y_3\}$

- A 'training' set of **manually classified documents** (hand-labeled), $k = \{(\omega_1, y_1), (\omega_2, y_3), (\omega_3, y_1)\}$

As output, our **supervised learning** setup would produce a **learned classifier** that was able to classify documents, $\omega \to y$

In order to figure out how to approach this problem, we need to think like a computer would need to. First and foremost, we need a way to keep track of what the document *is*. We can do this by keeping track of **features** that we produce based on what we observe about the document.

Definition:

Feature *(machine learning)* → A feature is an individual measurable property or characteristic of a phenomenon being observed. *(Wikipedia)*

Although we may be able to extract a lot of meaning from a clump of words, a computer cannot readily do the same. As such, we will first try and represent this **document** as **math**, loosely speaking. How can we do that?

Idea: Bag of Words
One idea is to represent each document as a **vector** of **word frequencies**. A good visual

example of something like this is a word cloud, but could also be as simple as a bar graph with frequencies.

In this case, *term frequency* represents the number of times that a term appears in a document.

Although this isn't our whole solution, it's a part of it. We can use this as a **feature** with which we can train our classifier, $\omega \to y$.

Similarly to how we identified just one feature above, we can deduce many other features from our documents. Then, we can assign **weights** to these features to tell our algorithms just how important each feature when our classifier renders its final decision.

However, not all is as simple as it seems- there's a clear **issue** with the **term frequency** we discussed.

When we approach this problem realistically, we can see that term frequency gets **overloaded** with **common words**, like *a*, *to*, *I*, and *the*. How can we intelligently address something like this?

Solution: Inverse Document Frequency

Idea: In order to address the commonality issue, we weight individual words negatively by how frequently they appear in the corpus.

$$IDF_j = log(\frac{d_{total}}{d_j})$$

In the above example, d_{total} represents the total number of documents and d_j represents the number of documents that contain word j. Note that IDF is defined across all documents, not a specific word-document pair.

In practice, we end up combining term frequency and inverse document frequency to get a more functional result.

NLP In Python

In order to work with NLP in Python, there are two major libraries we can make use of- **NLTK** and **SPACY**.

- **NLTK** $\to$ Has **more "stuff" implemented** and is more customizable. Started as a research tool, now has found lots of use in industry as well.

- **SPACY** $\to$ Younger and more sparse than **NLTK**, but can be much **faster than NLTK**. Much newer, more streamlined implementation.

Distributional Models of Meaning

Distributional models of meaning revolve around the idea that you know a word by the company that it keeps.

> **Definition:**
>
> **Distributional models of meaning** → quantitative; express the semantic relation between terms but offering no immediately obvious way of modelling the contribution of sentence structure to meaning

The idea is that two words are **similar** if they have **similar word contexts**.

Document Matrices

A **document matrix** is a way to quickly compare features of documents.
Here's an example that records 'term frequency'- how often each of these given words appear in the selected texts. We can determine two main things from the following document matrix.

	As You Like it	12th Night	Julius Caesar	Henry V
Battle	1	1	8	15
Soldier	2	2	12	36
Fool	37	58	1	5
Clown	6	117	0	0

Two words are similar if their vectors are similar.

Two documents are similar if their vectors are similar.

Measuring Similarity

We've seen a visual way to represent document matrices, but how do we actually, **quantitatively** tell if vectors are similar?

Using The Dot Product

One such method is to use the **vector dot product**. Given vectors v and w, recall that we can calculate dot product by multiplying their components as follows.

$$\sum_{i=1}^{n} v_i \cdot w_i$$

We can use this to develop a general idea of when 2 vectors have **large values** in the **same dimensions**. The dot product will also be lower for two vectors with different dimensional structures.

It will be zero for completely orthogonal vectors (representing wholly different objects), i.e. orthogonal vectors represent objects that cannot be any more different.

However, the dot product is affected by term frequency as a whole. Higher word frequencies overall lead to a larger dot product overall. This is **bad**- we don't want a similarity metric to be sensitive to word frequency in this way. For this reason, we need some way to **normalize** the dot product.

Solution: normalize the dot product by dividing by the lengths of both vectors.

$$\frac{\vec{a} \cdot \vec{b}}{|\vec{a}||\vec{b}|}$$

Now, take advantage of a property of the dot product to help us define this process.

$$\frac{\vec{a} \cdot \vec{b}}{|\vec{a}||\vec{b}|} = cos\theta$$

Thus, we can say that the **cosine of the angle between the vectors** provides us with a **normalized** measure of 'similarity' between both vectors. This process is formalized below.

Normalizing Dot Product: Cosine Similarity

Given two documents X and Y, represented here as vectors in matrix form, the **cosine similarity** of the two can be represented by the following equation:

$$similarity_{cos}(X, Y) = \frac{X^T Y}{|X||Y|}$$

Technically, we're measuring the **cosine of the angle between vectors X and Y**. This is how we determine similarity while ignoring term frequency.

- **Similar** documents have **high** cosine similarity.

- **Dissimilar** documents have **low** cosine similarity.

Example: Calculating Cosine Similarity

Given the following document matrix, here are a few examples of calculated cosine similarity. Assume the rows represent **documents** and the columns represent **words**.

	large	data	computer
apricot	2	0	0
digital	0	1	2
information	1	6	1

Also, note the formula for calculating cosine similarity given vectors v and w amounts to the following:

$$similarity_{cos}(v, w) = \frac{\vec{v} \cdot \vec{w}}{|v||w|} = \frac{\sum_{i=1}^{n} (v_i \cdot w_i)}{\sqrt{\sum_{1}^{n} v_i^2} \sqrt{\sum_{1}^{n} w_i^2}}$$

Examples:

$$similarity_{cos}(\textbf{apricot}, \textbf{information}) = \frac{2 + 0 + 0}{\sqrt{2}\sqrt{38}} = 0.23$$

$$similarity_{cos}(\textbf{digital}, \textbf{information}) = \frac{0 + 6 + 2}{\sqrt{38}\sqrt{5}} = 0.58$$

$$similarity_{cos}(\textbf{apricot}, \textbf{digital}) = \frac{0 + 0 + 0}{\sqrt{1}\sqrt{5}} = 0.00$$

Minimum Edit Distance

A metric used to calculate how similar 2 strings are to one another is **minimum edit distance**.

Definition:

Minimum Edit Distance $\rightarrow$ The minimum edit distance between two strings is the number of edit operations to turn one string into another.

This seemingly simple tool is used widely in NLP, namely for **spelling correction**, **information extraction**, **speech recognition**, and plenty of other things.

Aside: Information Extraction Systems

Miscellaneous further info on information extraction systems. Here are some general goals for an information extraction system:

- To extract **clear**, **factual** information

- To find and understand texts

- To gather information from **many** pieces of text

- To provide a structured representation of relevant information (e.g. in **relations** or **data structures**)

- To organize information to make it useful to humans

Representing Text with N-Grams

Idea: We can represent text as contiguous sequences (arrays) of N tokens/words.

Definition:

N-Gram $\rightarrow$ An n-gram is a contiguous sequence of N tokens/words. In layman's terms, it's just a group of words. Latin prefixes can denote how big they are, i.e. unigram is one word, bigram is two words, trigram is 3 words, etc.

Probabilities

We can use probabilities as the basis of solving a lot of problems in a language. Examples:

- **Machine Translation** → P(High winds tonight) > P(Large winds tonight)

- **Correctness Check/Grammar** → P(15 Minutes) > P(15 Minuets)

- **Speech Recognition** → P(I saw a van) > P(Ice of a ban)

In that regard, our goal should be to be able to calculate the probability of a sentence of words. (Generalized version of the above examples)

$$P(w) = P(w_1, w_2, w_3...w_n)$$

We can use **n-grams** for this purpose. An **n-gram model** is a type of probabilistic model for predicting the next item in such a sequence in the form of a (n-1) order Markov chain.

N-gram models are popular due to their **simplicity** and **scalability**. The given structure allows small experiments to scale up efficiently.

13 Machine Learning (L16 + 19)

Supervised machine learning gives us a way to automatically **infer** a predictive model, given some amount of pre-labeled data.

Overview

This diagram provides a broad overview of the supervised learning process.

Training Data	ML Algorithm	Predictions
(x^1, y^1) (x^2, y^2) (x^3, y^3) (x^4, y^4) . . . (x^n, y^n)	$y^i = h(x^i)$ Hypothesis function formed based on the given training data	Given a new input x_{new} The algorithm will predict what it ends up as $y_{new} = h(x_{new})$
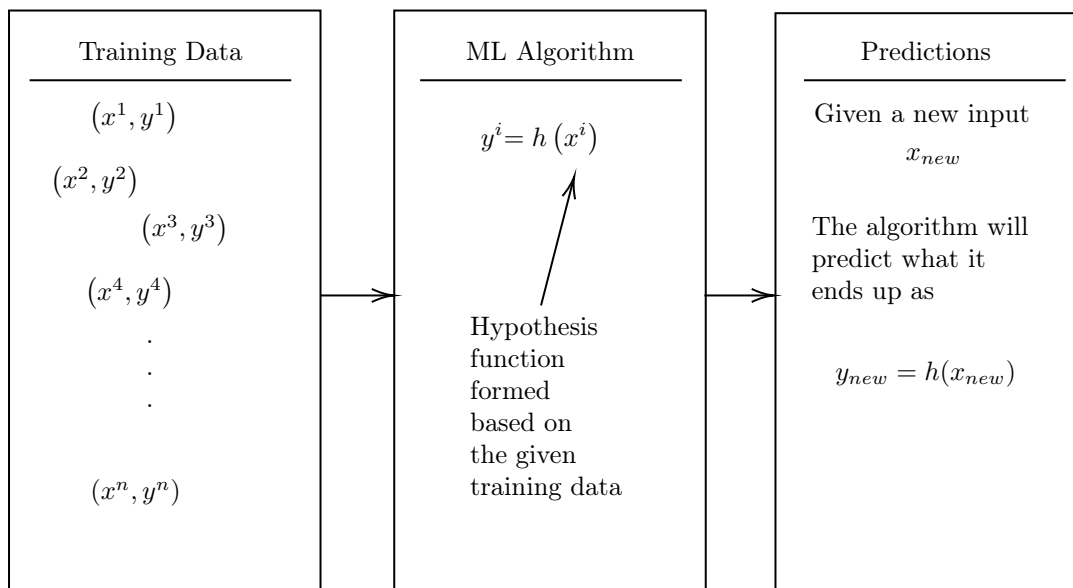
Labeled data (usually done manually) is provided to the machine learning algorithm, the algorithm runs and selects the best hypothesis function from the given hypothesis space, and then is able to render its own predictions on unlabeled data.

Hypothesis

> **Definition:**
>
> **Hypothesis** → A function that describes a relation between the input and the target in supervised learning.

The hypothesis can be affected by **data**, **restrictions**, **bias**, and other factors. These are the nuances of the ML algorithm.

The hypothesis itself must be selected from a given **hypothesis space**. Think of it as a starting point for the algorithm.

Providing the algorithm with a hypothesis space and asking it to design a hypothesis is like saying, "Here's the domain of all possible functions you can specify your hypothesis of choice from, now decide which one is the most optimal".

Hypothesis Function

At the end of the day, our algorithm is just trying to figure out the best hypothesis function. The end goal is the following: a function that can (supposedly) successfully classify all given inputs.

$$h_\theta : R^n \to y$$

(Takes all possible inputs and classifies them)

Our hypothesis function depends on domain knowledge. For example, a hypothesis function could be a line on a Naviance college acceptances graph that determines where acceptances and rejections will lie given an x coordinate of SAT scores and a y coordinate of GPA.

Hypothesis Space

> **Definition:**
>
> **Hypothesis Space** → Set of all possible legal hypotheses. The set from which the ML algorithm will decide the best.

Overall, the **hypothesis space** is simply the set of **all possible hypotheses**, and a **hypothesis** is a single potential solution. (The algorithm then chooses the **best** hypothesis.)

The Loss Function

The loss function (defined below) is a component of a machine learning algorithm that measures the difference between a prediction and the true output (what it's supposed to be).

$$l : y \times y \to R_+$$

There are **two types** of **loss functions**.

- **Classification Loss** → Classification functions predict a **label**. Examples of this type of loss function would be logarithmic loss, focal loss, KL divergence, relative entropy, exponential loss, hinge loss.

- **Regression Loss** → Regression functions predict a quantity. Examples of this type of loss function would be Mean square error (quadratic loss), Mean absolute error, Huber loss, Smooth mean absolute error, log cosh loss, quantile loss.

There is no single loss function that works on all sets of data. Your choice of loss function depends on factors like the following:

- Outliers

- Choice of ML Algorithm

- Time Efficiency of Gradient Descent

- Confidence of Predictions

It looks like, at the end of the day, you should just pick the right loss function for your situation.

The Canonical Machine Learning Problem

At the end of the day, we want to **learn** a hypothesis function that **accurately** predicts outputs given inputs.

As such, we can say that our **goal** is to **minimize loss** by choosing the parameterization that minimizes loss over all possible parameters.

Here's where the hypothesis function and the loss function fit into the big picture:

$$\underbrace{\sum_{i=1}^{m} l((h_\theta(x^i)), y^i)}_{\text{minimize}}$$

We want to **find a hypothesis** function that **minimizes the loss** across all the training data inputs that we have. In this case, m is our training data.

Minimizing this is not as easy as it seems- it's a high order differential equation, and we need an efficient way to optimize it. Different optimization methods work differently for different machine learning algorithms.

Overarching Idea

Big Idea: **Take in a hypothesis space, loss function and an optimizer, then decide a hypothesis function given a hypothesis space.**

Our **hypothesis function** will be domain-specific. It could be anything from drawing a line on a graph to separate pass from failure, or designing a function to classify different types of fruits.

Our **loss function** can be chosen based on our problem type, as we explored above. Again, this is case-specific.

Our **optimization method** is usually gradient descent or stochastic gradient descent- this is basically how you easily find minima of a high order differential equation.

Examples of ML Algorithms

Name	Hypothesis Function	Loss Function	Optimization Approach
Least Squares	Linear	Squared	Algo / Gradient Descent
Linear Regression	Linear	Squared	Algo / Gradient Descent
Support Vector Machine	Linear/Kernel	Hinge	Algo / Gradient Descent
Perceptron	Linear	Perceptron Criterion	Perceptron Algorithm
Neural Networks	Compound Nonlinear	Squared, Hinge	Stochastic Gradient Descent
Decision Trees	Hierarchical Half-planes	Many	Greedy
Naive Bayes	Linear	Joint Probability	#SAT

We can see that ML Algorithms are just the sum of their components. Its not so much our job to discover these methods, but instead to decide where to apply which method, using scientific insight.

Example: Linear Regression As Machine Learning

Consider linear regression that minimizes the sum of squared error, i.e. makes use of least squares. (First row in example table above).

In technical terms:

- **Hypothesis Function** $\to$ Linear $(h_\theta(x) = \theta^T x)$

- **Loss Function** $\to$ Squared error loss $(l(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2)$

- **Optimization** $\to$ Minimize $\sum_{i=1}^{m}(\theta^T x^i - y^i)^2$

Now, our task is to minimize the function we wish to optimize. Rewrite inputs in technical form:

$$
X = \begin{pmatrix} (x_1)^T \\ (x_2)^T \\ . \\ . \\ . \\ (x_m)^T \end{pmatrix} \in R^{m \times n}
$$

Feature Vectors a.k.a. Training Data Inputs

$$
y = \begin{pmatrix} y_1 \\ y_2 \\ . \\ . \\ . \\ y \end{pmatrix} \in R^m
$$

Labels a.k.a. Training Data Outputs

Now we'll review the gradient, then get into how we're applying gradient descent. Recall the following from multivariate calculus:

Definition:

Gradient $\rightarrow$ The multivariate generalization of the derivative

For a function $f : \mathbb{R}^n \rightarrow \mathbb{R}$, the gradient is a vector of all n partial derivatives of that function:

$$\nabla_\theta f(\theta) = \begin{pmatrix} \frac{\partial f(\theta)}{\partial f(\theta)_1} \\ . \\ . \\ . \\ \frac{\partial f(\theta)}{\partial f(\theta)_n} \end{pmatrix} \in \mathbb{R}^n$$

As such, minimizing a multivariate function involves finding a point where the gradient is zero.

$$\nabla_\theta f(\theta) = \begin{pmatrix} 0 \\ 0 \\ . \\ . \\ . \\ 0 \end{pmatrix}$$

That is, we find when the gradient is a zero vector of order n. However, note that points where the gradient is zero are mainly local minima. Only in convex functions can we be sure that this point is the global minimum.

Given our optimization problem can written as the following, we can look to find the gradient from there.

$$\text{minimize}_\theta \frac{1}{2}||X\theta - y||_2^2$$

By applying a few rules, we can arrive at the following gradient.

$$X^T(X\theta - y)$$

Now, solve for the model parameters θ, because we already have defined X and y above.

$$X^T(X\theta - y) = 0$$
$$X^T X\theta - X^T y = 0$$
$$X^T X\theta = X^T y$$
$$\underbrace{(X^T X)^{-1} \cdot X^T X}_{cancels} \theta = (X^T X)^{-1} \cdot X^T y$$
$$\theta = (X^T X)^{-1} \cdot X^T y$$

Thus, we have found a systematic way to find the optimal model parameters for linear regression, given the training data.

Machine Learning in Python

Python provides us with lots of options in terms of machine learning libraries.

Scikit-learn (`sklearn`) is the most well-known library. It provides us with a lot of functionality, namely:

- Classification (SVN, K-NN, Random Forest)
- Regression (SVR, Ridge, Lasso)
- Clustering (K-means, spectral, mean-shift)
- Dimensionality Reduction (PCA, matrix factorization)
- Model selection (Grid search)
- Preprocessing (Cleaning, EDA)

`sklearn` also works well with `numpy` and `matplotlib`.

(Stochastic) Gradient Descent

First, let's establish this definition.

> **Definition:**
>
> **Stochastic** $\rightarrow$ Randomly determined; having a random probability distribution or pattern that may be analyzed statistically, but may not be predicted precisely.

Recall that the **gradient** always points in the *local* direction of steepest ascent.

As such, we need to get to the 'lowest' point on our multivariate n-dimensional plane by taking minute steps *against* the gradient.

Gradient of $f(x,y) = xe^{-(x^2 + y^2)}$
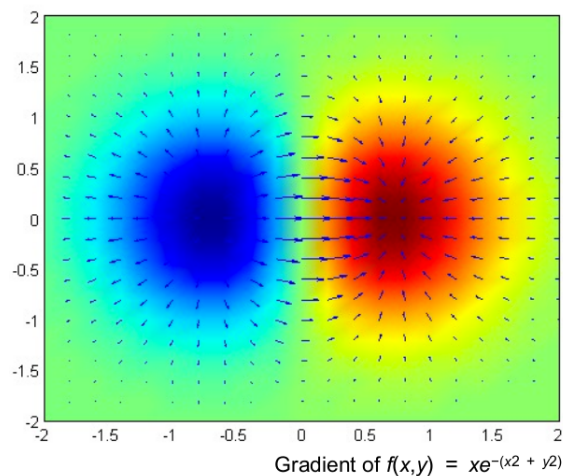
For example, in the above graph, the gradient points us towards higher points on the 3-dimensional plane. Our objective is to step against the gradient and find the minimum point.

Notice that you won't always end up at the global minimum (the blue)- it depends on where you start. If you start to the left of the red portion, you're fine, but if you start to the right of the red portion, you're in trouble. This is the nuance of gradient descent.

Procedure

Given the following initial elements:

- a hypothesis function: $h_\theta : R^n \to y$

- a loss function: $l : y \times y \to R_+$

- a step size: γ

- an initial parameter vector, θ of all zeroes

We have all we need to perform gradient descent. From there, repeat the following:

- Compute gradient:

$$g \leftarrow \sum_{i=1}^{m} \nabla_\theta l(h_\theta(x^i), y^i)$$

  (We want to see how our multidimensional optimization function is behaving- let's change up our parameters and make sure we're stepping in the right direction)

- Update parameters:

$$\theta \leftarrow \theta - \gamma \cdot g$$

Step size is also an important parameter to consider. If it's too big, you can end up oscillating around the minimum. If it's too small, you can take a really long time to converge on the minimum.

Batch vs. Stochastic Gradient Descent

- **Batch Gradient Descent** calculates gradient descent using the whole dataset. It's good for convex or relatively smooth error manifolds. That is, cases where you're pretty clearly moving towards an optimal solution or goal.

- **Stochastic Gradient Descent** computes using one sample. It works well for examples with lots of local minima and maxima. This seems to be a more realistic case.

14 Decision Trees (L20)

Let's look at the big picture of learning first, then move on to decision trees specifically.

Big Picture of Learning

Learning can be seen as **fitting a function** to the data. We can consider **different target functions** (which basically means that we can consider different hypothesis spaces).

A learning problem is **realizable** if its hypothesis space contains the **true function**. (The most accurate function)

There is always a **tradeoff** between the **expressiveness of a hypothesis space** and the **complexity of finding simple, consistent hypotheses** within the space.

Examples of different target functions:

- Propositional if-then rules

- Decision Trees

- Linear functions

- Neural networks

Decision Trees Intro

We'll be analyzing **Decision Trees**. Specifically, decision trees for boolean classification, where each example is classified as positive or negative.

First, what is a decision tree?

<div style="border:1px solid red">

Definition:

Decision Tree → A function that makes consecutive, sequential decisions based on the nature of that input, and produces a final result.

- **Input:** An object or situation described by a set of attributes (features)

- **Output:** A "decision" that predicts the corresponding output value.

Structured sort of like a flowchart or a how-to guide, the decision tree is very human-readable.

It is composed of two types of components: **Decision Nodes** and **Leaf Nodes**.

Decision Node → Specifies a choice or test of some attribute with 2 or more alternatives. Every decision node is a part of a path to a leaf node.

Leaf Node → Indicates classification of an example- function terminates.

</div>

A decision tree is a tree-like graph with nodes representing the place where we pick an attribute and ask a question; edges represent the answers the to the question; and the leaves represent the actual output or class label. They are used in non-linear decision making with simple linear decision surface. (*hackerearth*)

Example: Election Decision Tree

The following is an example of a decision tree. Note that the order of the decisions taken matters, and the results are all based in likelihood, not certainty.

Decision Tree: The Obama-Clinton Divide

In the nominating contests so far, Senator Barack Obama has won the vast majority of counties with large black or highly educated populations. Senator Hillary Rodham Clinton has a commanding lead in less-educated counties dominated by whites. Follow the arrows for a more detailed split.

Is a county more than 20 percent black?

NO There are not many African-Americans in this county.

YES This county has a large African-American population.

And is the high school graduation rate higher than 78 percent?

NO This is a county with less-educated voters.

YES This is a county with more educated voters.

Clinton wins these counties 704 to 89.

Obama wins these counties 383 to 70.

And is the high school graduation rate higher than 87 percent?

NO 78 to 87 percent have a diploma.

YES This is a highly educated county.

And where is the county?

Northeast or South | West or Midwest

Obama wins these counties 185 to 36.

Clinton wins these counties 182 to 79.

In 2000, were many households poor?

YES At least 47% earned less than $30,000.

NO At least 53% earned more than $30,000.

Given:
A collection of examples $(x, f(x))$

Return:
A function h (*hypothesis*) that approximates f, where h is a **decision tree**.

Decision Tree Learning Algorithm (ID3)

Decision trees can express **any Boolean function**.

Our goal is to find a decision tree that '**agrees**' with our training data set. If it agrees with the training data, it should theoretically produce correct output for the actual inputs.

One idea is to **construct a specific path for each example**; this would guarantee us accuracy with the training data. Unfortunately, this just won't work. This approach would just memorize the example, and **doesn't generalize** at all.

Therefore, we want the smallest tree, to minimize chances of this sort of 'memorization' from happening. One would think there's some systematic way to minimize this, but that problem ends up being **NP-hard**.

Therefore, our **overall goal** should be to get the best classification with the smallest number of tests-minimize decision nodes while maximizing accuracy.

Badminton Example: Hackerearth

Let's assume we had the following data as a table:

Day	Weather	Temperature	Humidity	Wind	Play?
1	Sunny	Hot	High	Weak	No
2	Cloudy	Hot	High	Weak	Yes
3	Sunny	Mild	Normal	Strong	Yes
4	Cloudy	Mild	High	Strong	Yes
5	Rainy	Mild	High	Strong	No
6	Rainy	Cool	Normal	Strong	No
7	Rainy	Mild	High	Weak	Yes
8	Sunny	Hot	High	Strong	No
9	Cloudy	Hot	Normal	Weak	Yes
10	Rainy	Mild	High	Strong	No

It indicates our human observations as to whether or not we'd decide to play badminton on a given day based on environmental factors, for the last 10 days.

We may use this table to help us decide whether we want to play or not in the future, but let's say that on day n, we find that the environmental factors are different than any of the previously recorded days.

We can try and form a **decision tree** based on this data that will help us make our decision. Below is an example decision tree for this data.

69
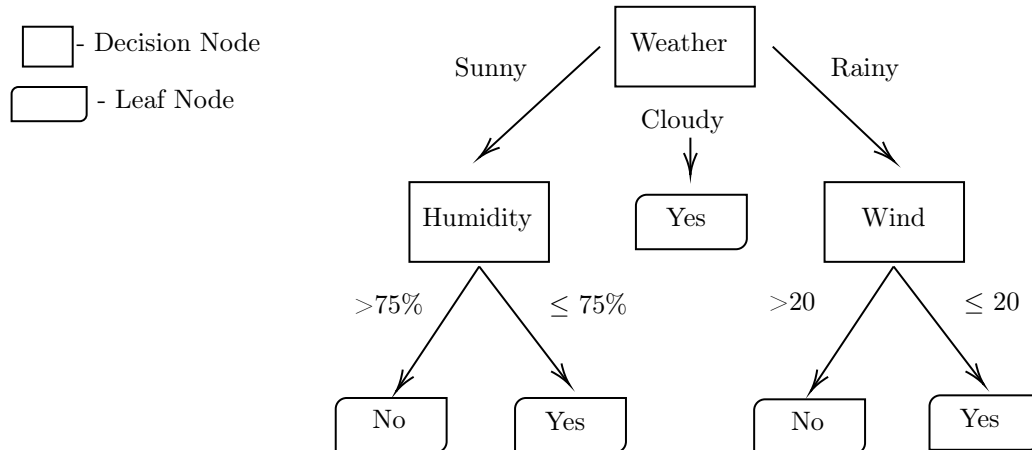
ID3 Algorithm

The following is a basic algorithm used to construct decision trees. It's known as the *Iterative Dichotomiser 3*, or *ID3* for short. It builds decision trees using a **top-down**, **greedy** approach.

1. Select the **best** attribute/feature, γ, from the pool of given features. The "best" attribute is the attribute which best splits or separates the data.

2. Assign γ as the decision attribute for a new node, β.

3. For each value of γ, create a new descendant of β.

4. If the training data is now perfectly classified, stop. Otherwise, recursively start from step 1 on each of the descendants of β.

This approach is fairly straightforward; let's investigate further into how we can end up choosing the 'best' attribute from the pool of given attributes. For this, we need to establish the concept of **information gain**.

Information Gain + Entropy

> **Definition:**
>
> **Information Gain** $\rightarrow$ A measure that expresses how well an attribute splits our data into groups based on classification.

Information gain is a statistical property that measures how well a given attribute separates the training examples according to target classification. If the presence of a certain attribute lead to way more of one outcome than another, we consider that attribute to have a **high information gain**.

However, we still need a way to define this property precisely. To aid in this definition, we will define the idea of entropy.

Definition:

Entropy $\rightarrow$ A property that measures the level of *impurity* in a group of examples.

In the case of our *ID3* algorithm, which only provides results in the form of **positive** or **negative**, we can define entropy with the following formula:

$$Entropy(S) = -p_+ log_2 p_+ - p_- log_2 p_-$$

Where:

- S is a sample from our training data.

- p_+ is the proportion of positive examples in S.

- p_- is the proportion of negative examples in S.

This formula essentially tells us how 'split' the data is on a scale from 0 to 1.0- for example:

Example: S is a sample containing 14 boolean examples, with 9 positive and 5 negative examples. The Entropy of S relative to this classification is:

$$Entropy([9+, 5-]) = -\frac{9}{14} \cdot log_2 \frac{9}{14} - \frac{5}{14} \cdot log_2 \frac{5}{14} = 0.940$$

Note that entropy is 0, or optimal, if all members of S belong to the same class, and entropy is 1.0 if exactly half the members belong to one class, and the other half belong to the other.

At this point, our intuition tells us that we need to **pick the attribute that reduces the entropy the most.**

Using this definition of entropy, we can now properly define information gain as a measure of the effectiveness of an attribute in classifying the training data.

Information Gain, $Gain(S, \gamma)$, of an attribute γ relative to a sample of examples S is defined as the following:

$$Gain(S, \gamma) = \underbrace{Entropy(S)}_{\text{Entropy of original sample S}} - \underbrace{\sum_{v \in Values(\gamma)} \frac{|S_v|}{|S|} Entropy(S_v)}_{\text{Expected value of entropy after S is partitioned using } \gamma}$$

Where:

- $Values(\gamma)$ is the set of all possible values for attribute γ

- S_v is the subset of S for which attribute γ has value v

Overall, we can see that $Gain(S, \gamma)$ is therefore the expected reduction in entropy caused by knowing the value of attribute γ. We're basically trying to say this:

$$\text{Information Gain} = Entropy(\text{Parent Node}) - AvgEntropy(\text{Children Nodes})$$

Example: Information Gain + Entropy

Suppose a sample S has 30 instances, 14 positive and 16 negative.

Also suppose that an attribute γ divides the samples into two subsamples of 17 instances (4 negative and 13 positive) and 13 instances (1 positive and 12 negative).

30 instances

(14 positive, 16 negative)

Divide on γ

17 instances

(13 positive, 4 negative)

13 instances

(1 positive, 12 negative)
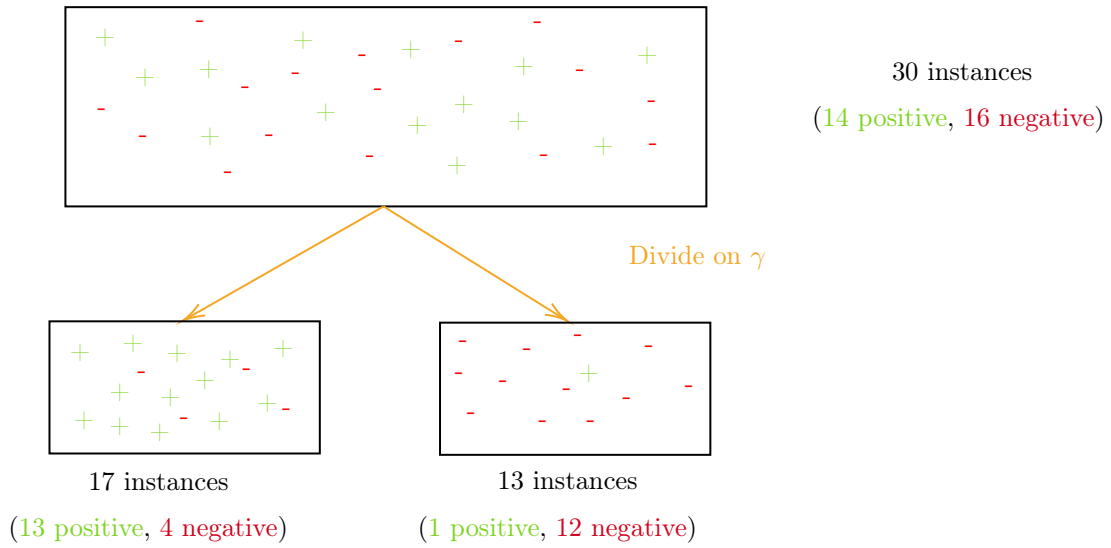
Given these conditions, we can calculate the **information gain** of attribute γ.

We know that the following formula calculates information gain of attribute γ from a sample S:

$$Gain(S, \gamma) = \underbrace{Entropy(S)}_{\text{Entropy of original sample S}} - \underbrace{\sum_{v \in Values(\gamma)} \frac{|S_v|}{|S|} Entropy(S_v)}_{\text{Expected value of entropy after S is partitioned using } \gamma}$$

We also know that the entropy of sample S can be calculated using:

$$Entropy(S) = -p_+ log_2 p_+ - p_- log_2 p_-$$

Using this formula, we can deduce the following:

- Entropy of parent (30 instances): $\frac{14}{30} \cdot log_2 \frac{14}{30} - \frac{16}{30} \cdot log_2 \frac{16}{30} = 0.996$

- Entropy of child (17 instances): $\frac{13}{17} \cdot log_2 \frac{13}{17} - \frac{4}{17} \cdot log_2 \frac{4}{17} = 0.787$

72

- Entropy of child (13 instances): $\frac{1}{13} \cdot log_2 \frac{1}{13} - \frac{12}{13} \cdot log_2 \frac{12}{13} = 0.391$

Now, we take the weighted average of the entropy of the children (expected value of entropy after S is partitioned using γ).

$$\frac{17}{30} \cdot 0.787 + \frac{13}{30} \cdot 0.391 = 0.615$$

Using all this information, we can calculate the information gain for attribute γ:

$$G(S, \gamma) = 0.996 - 0.615 = 0.38$$

Keeping this in mind, we are free to calculate information gain for **each attribute** and, based on this number, decide which attribute that we should split the data on for our ID3 algorithm.

Noisy Data

Lots of 'noise' could potentially occur in your training data- **training data is not necessarily perfect**. Examples:

- Two examples have the same attribute/value pairs, but different classifications

- Some values of attributes are incorrect because of errors in the data acquisition or preprocessing phase.

- The classification itself is wrong due to some error

Noisy data is just one of the reasons you don't want to **overfit** your data.

Overfitting

> **Definition:**
>
> **Overfitting** $\rightarrow$ Fitting the training set "too well" such that performance on the test set degrades.

At the end of the day, the training data is just that- training data. You don't want your algorithm to learn so much from your training data such that it adheres to it so perfectly that it fails to properly classify new training data that it hasn't yet seen. In other words, you want your algorithm to learn, but not totally and entirely meticulously.

Example: Overfitting

> You are trying to predict the roll of a die. The experiment data includes:
>
> - Day of the week
>
> - Month of the week
>
> - Color of the die

Decision Tree Learning may find a hypothesis that fits the data, but with irrelevant attributes.

Some attributes are irrelevant to the decision making process, such as the color of the die, but the algorithm may use them to differentiate examples all the same.

If the hypothesis space has **many dimensions** because of a **large number of attributes**, we may find a **meaningless regularity** in the data that is irrelevant to the true, important distinguishing features of the dataset.

Overfitting is a key problem in learning there are formal results on the number of examples needed to properly train a hypothesis of a certain complexity- the more parameters, the more data is needed.

Defining Overfitting

Consider D, the entire distribution of data, and T, the training set. We can say that hypothesis $h \in H$ overfits D if $\exists h' \neq h \in H$ such that $error_T(h) < error_T(h')$ but $error_D(h) > error_D(h')$.

That's basically saying that hypothesis h overfits the data if there exists another hypothesis that's less accurate for the training data but more accurate on the validation data.

Aside: Fitting Training Data vs. Validation Data

It's hard to think of a better to the training data as a 'worse' result. It's often difficult to fit training data well, so it seems that a better fit to the training data means a good result. This is an easy way to fall victim to overfitting.

Here's an example of overfitting as decision tree size increases.

Note: with larger and larger trees, we just do better and better on the training set!
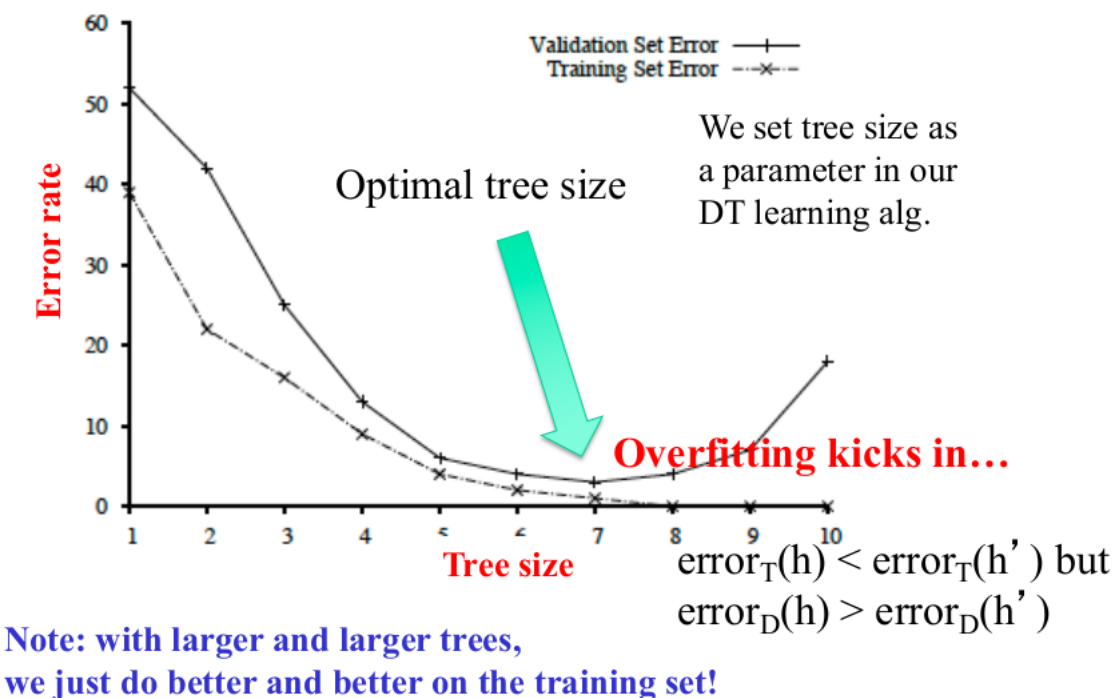
Evaluation Methodology

How do we evaluate the quality of the hypotheses produced by learning algorithms? We generally do this by seeing how good they are at classifying unseen examples.

Standard Methodology: Holdout Cross-Validation

1. Collect a large set of examples.

2. Randomly divide the collection into two disjoint sets, training set and validation set

3. Apply a learning algorithm to the training set and generate hypothesis h

4. Measure performance of h with regards to the test set

It's important to keep the testing sets and training sets disjoint as well- that is, make sure there is no **peeking**.

> **Peeking Example**:
> Suppose generate four different hypotheses- for example by using different criteria to pick the next attribute to branch on. We test the performance of the four different hypotheses on the test set and then select the best hypothesis.
>
> This is considered **peeking**, as the hypothesis was selected **based on** its performance on the **test set**, so information about the test set leaked into the learning algorithm.
>
> In this case, a new, separate test set would be required- one that our algorithm has not yet seen.

Learning Curve Graph

The learning curve graph is the average prediction quality as a function of the size of the training set. Usually, as the training set increases, so does the quality of the prediction.

Precision vs. Recall

> **Definitions:**
>
> **Precision** $\rightarrow \frac{\text{number of true positives}}{\text{number of true positives} + \text{number of false positives}}$
>
> **Recall** $\rightarrow \frac{\text{number of true positives}}{\text{number of true positives} + \text{number of false negatives}}$

A precise classifier is selective, while a classifier with high recall is inclusive. There are situations where we'd err to either one side due to caution, precision or recall. It's a case-by-case thing.

Decision Trees in Scikit

As expected, it's very easy to load up a decision tree in `sklearn`. Here's a python snippet to train a decision tree using default parameters.

```
1  from sklearn.datasets import load_iris
2  from sklearn import tree
3
4  # Load a common dataset, fit a decision tree to it
5  iris = load_iris()
6  clf = tree.DEcisionTreeClassifier()
7  clf = clf.fit(iris.data, iris.target)
8
9  # Predict the most likely class
10 clf.predit([[2., 2.]])
```

You can also use `graphviz` and `pydotplus` to visualize a decision tree using Python.

Random Forests

Decision trees are very interpret-able, but may be brittle to changes in the training data, as well as noise.

Random forests are an ensemble method that:

- Resamples the training data

- Builds many decision trees

- Averages predictions of trees in order to classify

This is done via bagging and random feature selection.

Bagging

Bagging is short for **B**ootstrap **Ag**gregation.

Essentially what you're doing when you're bagging is resampling a training set of size n via the bootstrap, a sample of n elements taken *with replacement* from the original dataset. For example:
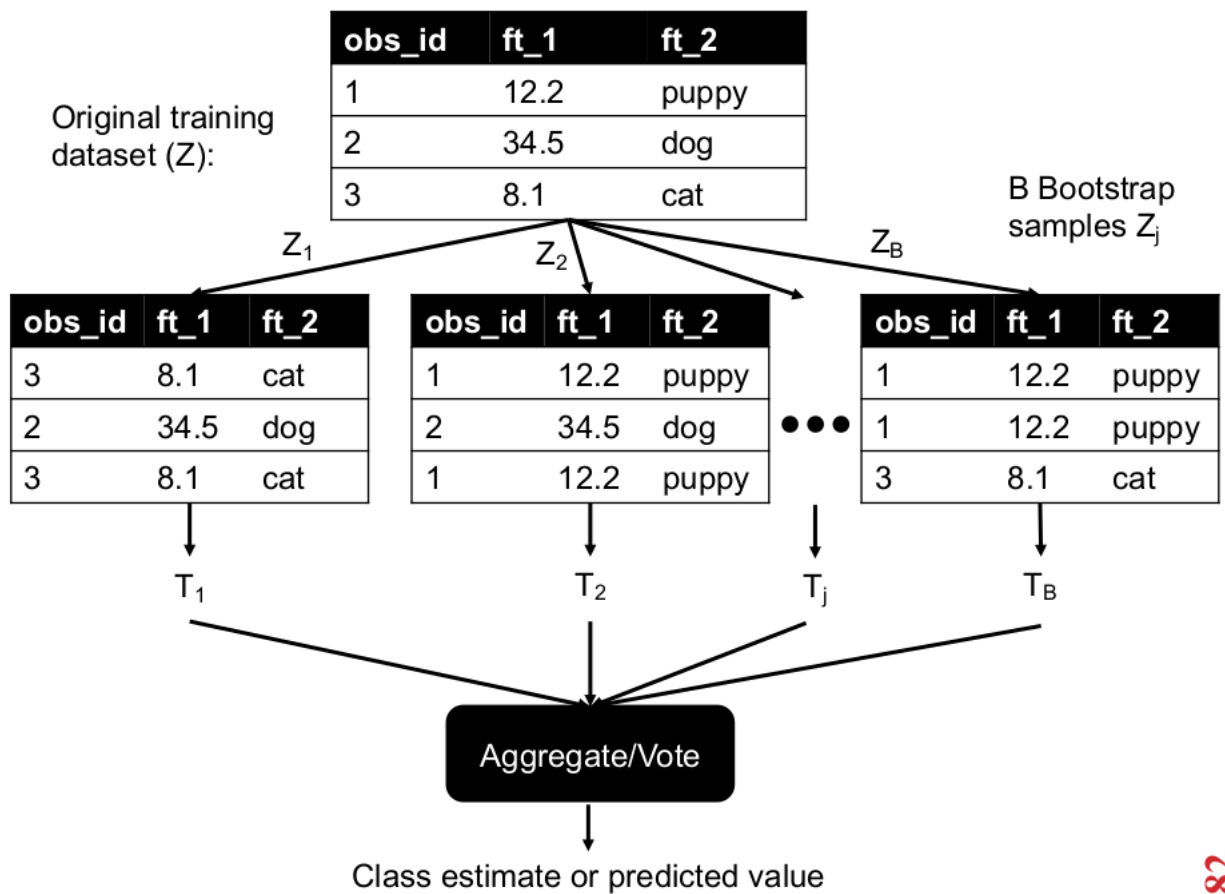
Original dataset: `1,2,3,4,5,6,7,8,9,10`

Resampled training set 1: `2,3,3,5,6,1,8,10,9,1`
Resampled training set 2: `1,1,5,6,3,8,9,10,2,7`
Resampled training set 3: `1,5,8,9,2,10,9,7,5,4`

Below is a visual example of bagging:

Original training dataset (Z):

obs_id	ft_1	ft_2
1	12.2	puppy
2	34.5	dog
3	8.1	cat

B Bootstrap samples Z_j

Z_1

obs_id	ft_1	ft_2
3	8.1	cat
2	34.5	dog
3	8.1	cat

Z_2

obs_id	ft_1	ft_2
1	12.2	puppy
2	34.5	dog
1	12.2	puppy

Z_B

obs_id	ft_1	ft_2
1	12.2	puppy
1	12.2	puppy
3	8.1	cat

T_1 T_2 T_j T_B

Aggregate/Vote

Class estimate or predicted value

Since bagging resamples the original training data *with replacement*, some instances (or even entire resampled datasets) may be present multiple times, while others are left out.

Once we perform analysis on all the separate datasets, we can have those analyses 'vote' and decide what the final classification should be.

Random forest classification is available in `sklearn`, simply import `RandomForestClassifier` from `sklearn.ensemble`.

15 K-Nearest Neighbors & Support Vector Machines (L21)

Overview

Basic idea: If it walks like a duck, quacks like a duck, then it's probably a duck.

It's a classifier that decides the value of a data point by analyzing that data point's k nearest neighbors.

The **K-NN Classifier** requires three things:

- The set of stored records
- A distance metric, to compute distance between records

- The value of k, the number of nearest neighbors to retrieve

In order to classify an unknown record given the above three things:

- You compute the distance from your unknown record to other training records

- You identify the list of k nearest neighbors

- Use the class labels of those k nearest neighbors to determine the class label of this record (e.g. by taking a majority vote)
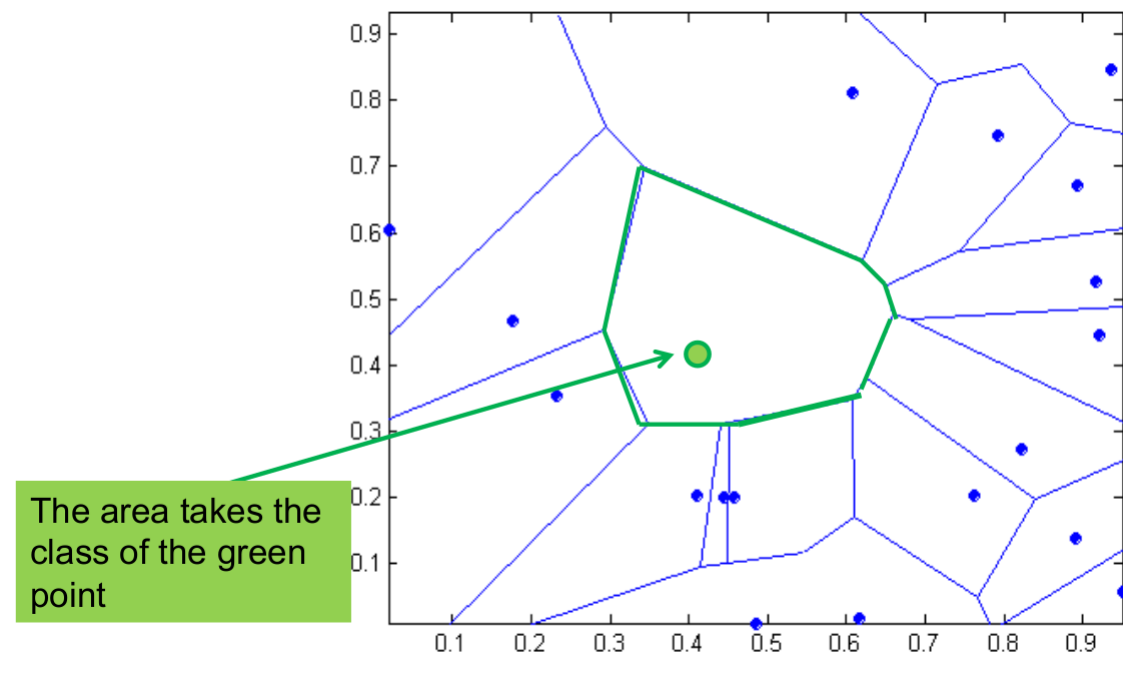

The K-NN classifier will mark this unknown point x as a +, due to its neighbors
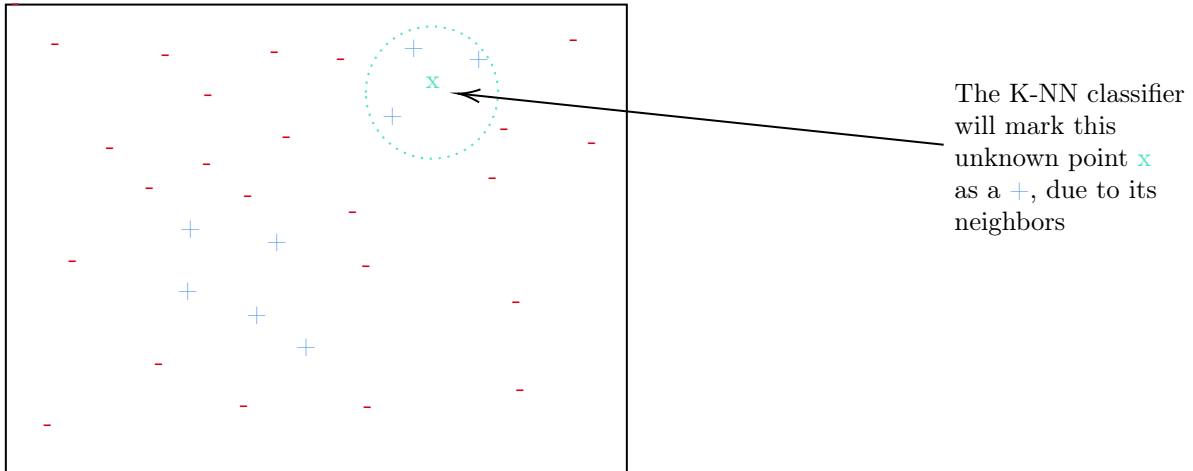
You can have 1-nearest neighbor, 2-nearest neighbors, 3-nearest neighbor, etc. The bottom line is that K-nearest neighbors of a record x are data points that have the k smallest distances to x.

Using Nearest Neighbors

We can compute the distance between our unknown point x and all the other points to develop a nearest neighbor list. We can then take the majority vote of class labels among the k-nearest neighbors. We can also weight the vote according to distance.

Choosing a Value for k

- If k is too small, it's sensitive to noise points (i.e. one or two 'mistake' points may throw it off)

- If k is too large, the neighborhood may include points from other classes; it may end up being useless.

Scaling

Additionally, attributes have to scaled to prevent distance measured from being dominated by one of the attributes. For example. if we are comparing people:

- height of a person may vary from 1.5m to 1.8m

- weight of a person may vary from 90lb to 300lb

- income of a person may vary from \$10k to \$1M

We can see that some of these variables vary greatly, while others vary minimally. As such, we have to **standardize** the variables in order to prevent this problem.

Curse of Dimensionality

There is another problem with using a euclidean measure, if we have data of a high dimension, we know from the *Curse of Dimensionality* that data becomes sparse relative to the total volume of the space, so the distance metrics that we're using effectively lose meaning. This can produce counter-intuitive results- for example:
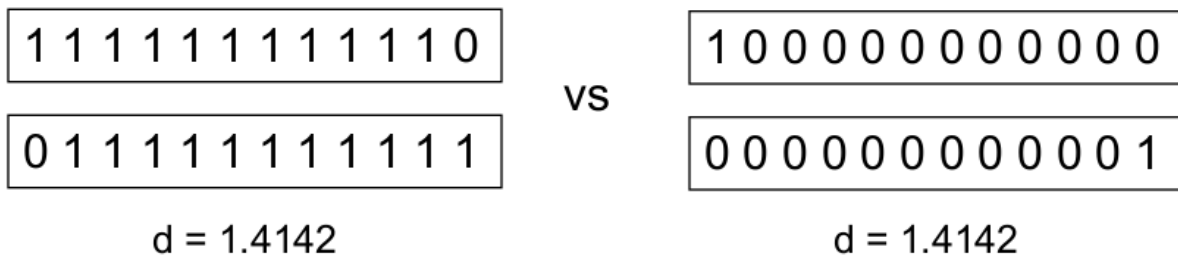
These two data points have the same *distance*, which is the only measure that the K-NN classifier uses, but we can see with the human eye that they are extremely different comparisons.

Think of each 1 or 0 as a value for a variable, these are just multivariable vectors.

We can combat this by **normalizing vectors to unit length**.

K-NN is Lazy

K-NN classifiers are known as lazy learners. It doesn't build a model explicitly, such as eager learners like the decision-tree algorithm.

Aside: Lazy vs. Eager Learners

Learning algorithms can either be called lazy or eager learners. Here are the differences.

Lazy Learners:

- Store the dataset without initially **learning** from it.

- Start classifying data when it receives test/validation data.

- Takes less time to 'learn' and more time to 'classify'.

Eager Learners:

- When it receives data, it immediately starts classifying them.

- Does not wait for test data to proceed with learning.

- Takes a long time to learn, but less time to classify data.

Examples of both in supervised learning:

- **Lazy:**

  - K-NN
  - Case-based reasoning

- **Eager:**

  - Decision Tree
  - Naive Bayes
  - Artifical Neural Networks

As such, **classifying unknown records is relatively expensive**.

The Naive algorithm is done in linear time, `O(n)`. We are in need of data structure that help us retrieve nearest neighbors quickly. Spatial data structures that can help us with this will be covered more in CMSC420.

Nearest Neighbor Search

The nearest neighbor search problem is an **optimization problem** of finding the point in a given set that is closest or most similar to a given point.

Formally, the problem is defined as follows:

> Given a set S of points in a space M and a query point $q \in M$, find the closest point in S to q.

Knuth called it the 'post-office' problem, referring to an application of assigning to a residence the nearest post-office.

Most commonly, M is a metric space, and dissimilarity is expressed as a distance metric (e.g. Euclidean Distance or Manhattan Distance).

Exact Methods

Aside: Branch and Bound Paradigm

Branch and bound is an algorithm design paradigm which is generally used for solving combinatorial **optimization problems**. These problems are typically exponential in terms of time complexity and may require exploring all possible permutations in worst case. The Branch and Bound Algorithm technique solves these problems relatively quickly.

More is covered in Algorithms.

Aside: R-tree

R-trees are tree data structures used for spatial access methods, i.e., for indexing multi-dimensional information such as geographical coordinates, rectangles or polygons. In other words, the best way to think about an R-tree is like for a **2-d map space**.

A common real-world usage for an R-tree might be to store spatial objects such as restaurant locations or the polygons that typical maps are made of: streets, buildings, outlines of lakes, coastlines, etc. and then find answers quickly to queries such as "Find all museums within 2 km of my current location", "retrieve all road segments within 2 km of my location" (to display them in a navigation system) or "find the nearest gas station" (although not taking roads into account).

R-trees are similar to B+ Trees in that **no data is actually stored in the internal nodes, only in the leaf nodes**.
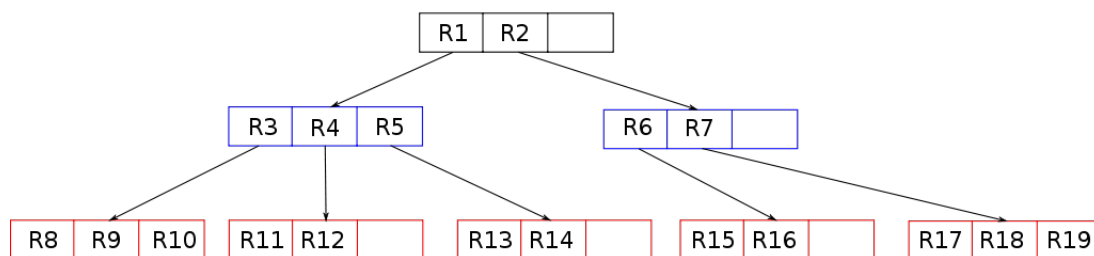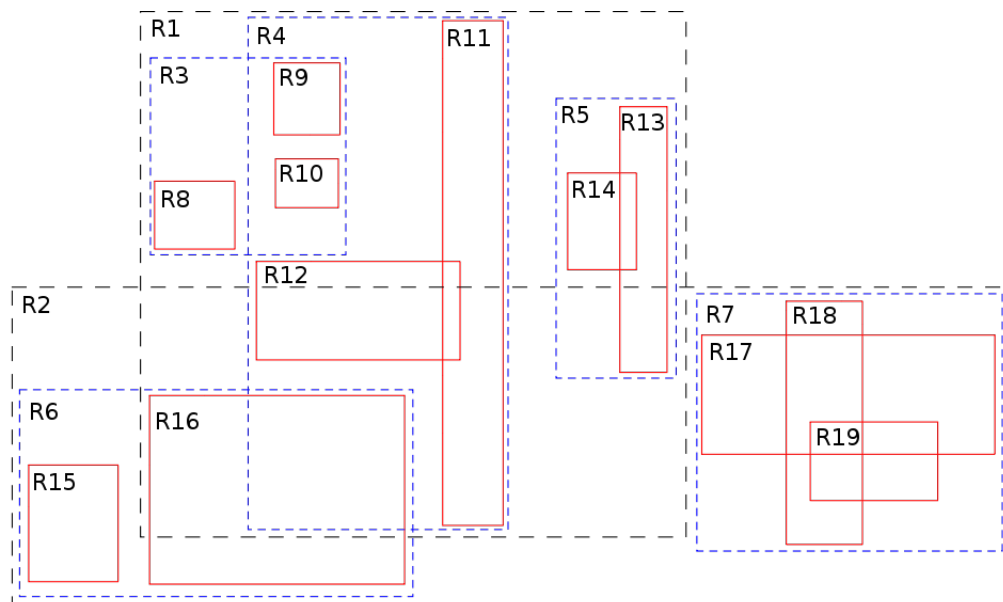
An R-tree consists of:

- Root node - holds a pointer reference to largest region - **(grey wide dashed line)**

- Internal nodes - hierarchically store pointers to child nodes, which either store more internal nodes, or leaf nodes. In any case, this will be **data bounded by this internal node - (smaller dashed lines)**

- Leaf nodes - store actual data within respective Minimum Bounding Regions (sub-regions within the entire space that group data in as few regions as possible) - **(solid lines)**

The R-tree can also be generalized to higher dimensions than just the two we see here, (which is where we can make use for it with k-NN classifiers).
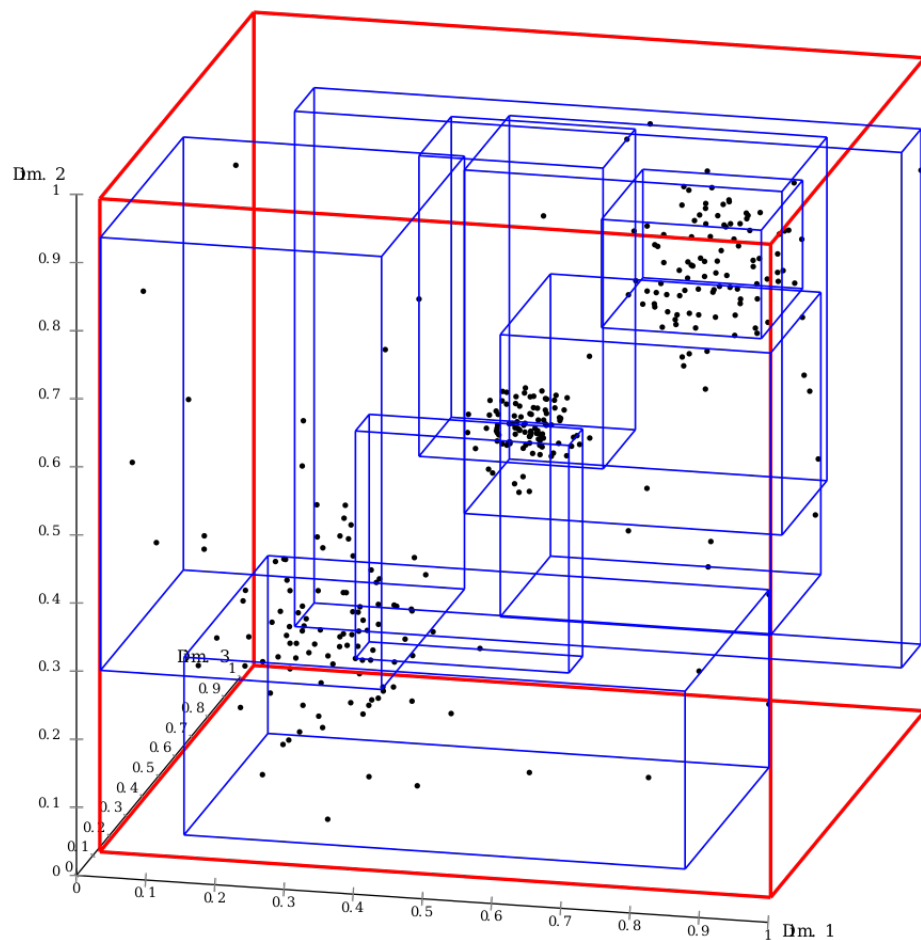
Above is a 3 dimensional generalization of the an R-tree.

- **Linear Search**

    - The simplest solution to NNS is to compute the distance from the query point to every other point in the database, while keeping track of the current minimum. (Naive approach)

- **Space Partitioning**

    - The 'branch and bound' methodology has been applied to this problem since the 1970s. The simplest of these approaches is the **kd-tree**, which bisects the search spaces into two regions containing half the points of the parent region, over and over again.

        * E.g. a **2d kd-tree** is a data structure for answering nearest neighbor queries in R^2.

· Select the x or y dimension (alternating between the two

· Partition the space into two with a line passing from the median point

· Repeat recursively in the two partitions as long as there are enough points to do so

· Now that you've built the tree, you can **quickly query it** for nearest neighbors by finding an incumbent best, and pruning large chunks of the tree away as you go

It's worth nothing that this algorithm can be generalized to higher dimensions, for the sake of our advanced problems.

– The **r-tree** was also developed to support nearest neighbor search in a dynamic context.

Approximation Methods

An approximate nearest neighbor search algorithm is allowed to return points, whose distance from the query is at most c times the distance from the query to its nearest points. The appeal of this approach is that, in many cases, an approximate nearest neighbor is almost as good as the exact one.

Proximity Graph Methods, such as HNSW, are considered the current state-of-the-art for the approximate nearest neighbors search.

Aside: HNSW

The industry standard for the approximate K-nearest neighbor search is known as the method leveraging **Hierarchical Navigable Small World** graphs.

The solution is graph-based, and doesn't depend on any other additional search structures.

Hierarchical NSW incrementally builds a multi-layer structure consisting from hierarchical set of proximity graphs (layers) for nested subsets of the stored elements. The maximum layer in which an element is present is selected randomly with an exponentially decaying probability distribution. This allows producing graphs similar to the previously studied Navigable Small World (NSW) structures while additionally having the links separated by their characteristic distance scales.

Starting search from the upper layer together with utilizing the scale separation boosts the performance compared to NSW and allows a logarithmic complexity scaling. Additional employment of a heuristic for selecting proximity graph neighbors significantly increases performance at high recall and in case of highly clustered data.

Other methods include:

• Locality sensitive hashing

• NNS in spaces with small intrinsic dimension using cover trees

• Projected radial search

- Vector approximation files

- Compression/clustering based search

Advantages of K-NN

- **Simple** technique that is easily implemented

- Building a model is **cheap**

- Extremely **flexible** classification scheme

K-NN is well-suited for multimodal classes and records with multiple class labels.

Additionally, it's been shown that it sometimes is the best method.

Disadvantages of K-NN

Classifying unknown records are relatively expensive, as you have to perform the distance computation of k-nearest neighbors. This is computationally intensive, especially as the size of the training set grows.

The accuracy of the classification can also be severely degraded by the presence of:

- Noisy or irrelevant features

- High-dimensional space (curse of dimensionality)

- Choosing the wrong distance metric

- Picking a bad value for k

## K-NN Classification in Scikit-Learn

As expected, `sklearn` supports K-NN classification.

```
1 from sklearn import neighbors, datasets
2
3 # Load a common dataset, fit a 15-NN classifier to it
4 iris = datasets.load_iris()
5 X = iris.data[:, :2] # take the first two features
6 y = iris.target
7
8 clf = neighbors.KNeighborsClassifier(15, weights='uniform')
9 clf.fit(X,y)
```

K-NN Regression

Here's the idea- to predict a target value y for a data point x, we simply apply interpolation and regression to the neighborhood of x.

The most naive approach to this would be to simply connect the dots in your graph, yielding the following regression:
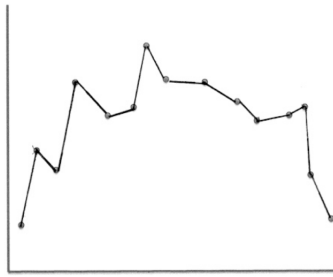
We can also, however, apply K-NN methodology to this problem. For example, we can fit a line using linear regression (when k is some number, or we could predict the average target value of the k points.
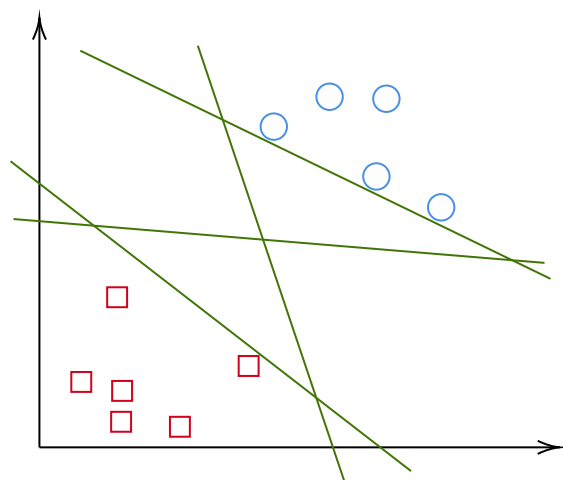
Here is what those two graphs look like.
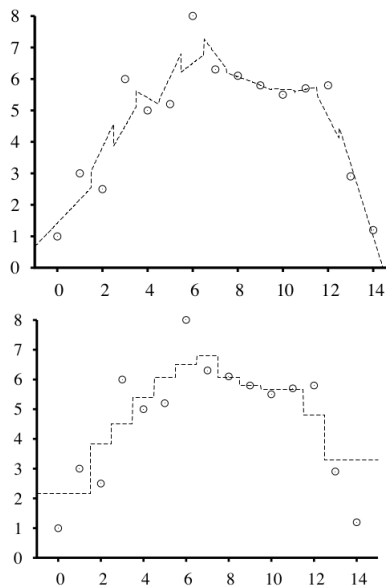

Support Vector Machines

The objective of a Support Vector Machine algorithm is to find a hyperplane in an n-dimensional space (where n is the number of features) that directly classifies the data points.

Below are examples of such planes in a simple 2D space.

These are all valid 'hyperplanes' that appropriately separate the data

Not only this, but the hyperplane is supposed to be **optimal**. This means that we maximize the margin between the hyperplane and the given data.
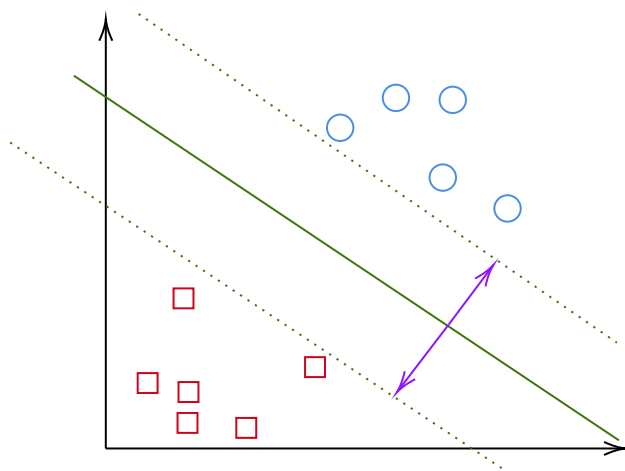
The optimal hyperplane maximizes the margin from the hyperplane.

Definition:

Hyperplane $\rightarrow$ Decision boundaries that exist in n-dimensional space which help us classify data points. Data points falling on either side of a hyperplane can be attributed to different classes.

It's important to note that although the given examples lie in $\mathbb{R}^2$, this learning method generalizes hyperplanes to $\mathbb{R}^N$.

If the problem is not linearly separable but should still be described by a linear hyperplane, we introduce **slack variables**.

We can introduce some sort of penalty for hyperplanes that deduct accuracy points for how far wrongly classified points lie from the hyperplane. Example:
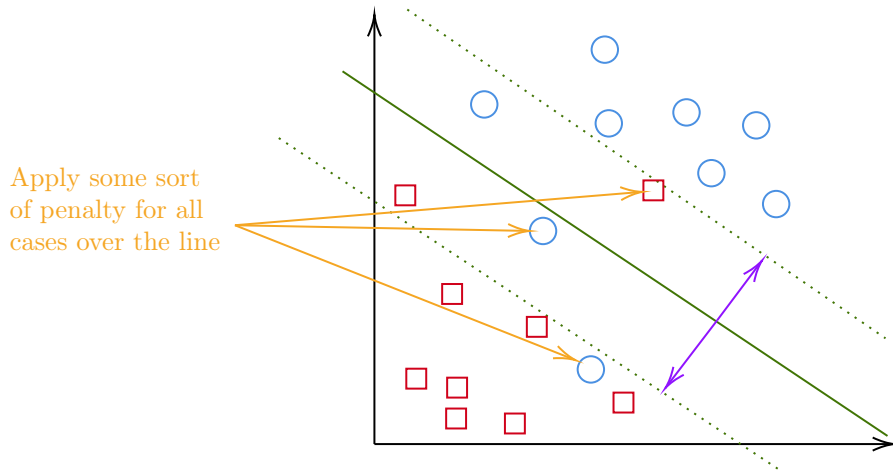
If the Decision Boundary Is Not Linear

If the decision boundary itself is not linear, e.g. if we have something like this, it's not conventional to then establish some sort of bendy-looking hyperplane, like below.
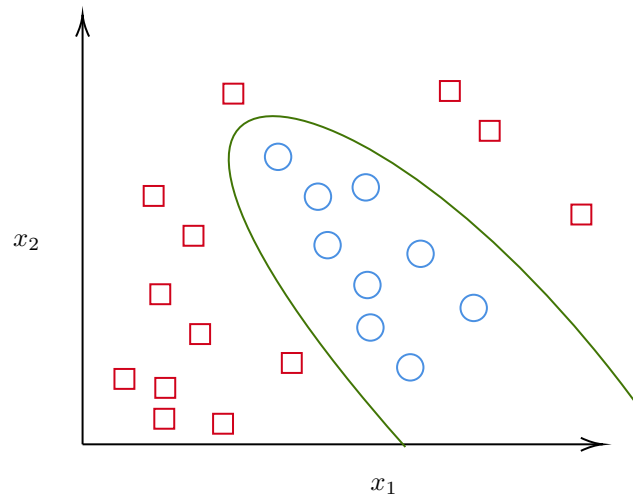
Instead, we can simply transform the data into higher dimensional space, and attempt to establish a hyperplane there.

SVMs in Scikit-Learn

`sklearn` also has the ability to produce SVM classifiers.

```
1  from sklearn import svm
2
3  # Fit a default SVM classifier to fake data
4  X = [[0, 0], [1, 1]]
5  y = [0, 1]
6
7  clf = svm.SVC()
8  clf.fit(X, y)
```
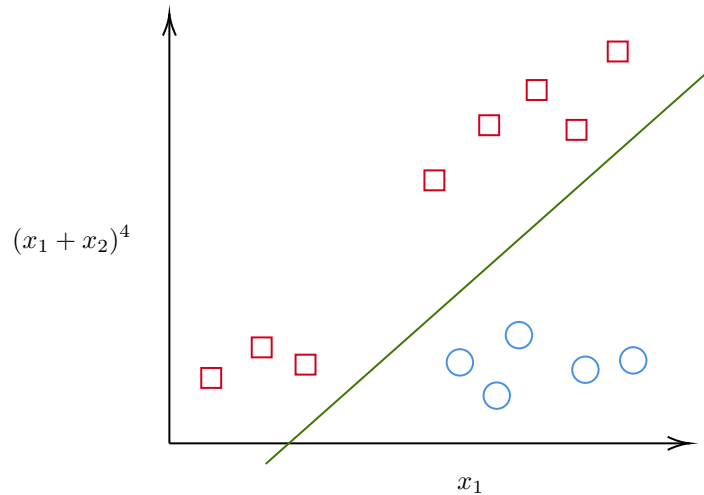
16   Nonlinear Regression & Regularization (L22 + 23)

So far, we have known linear regression as just 'drawing a line through our data', but it was in actuality a function of our input features.

It may seem limited, but when you apply linear regression algorithms to non-linear features of input data, you can essentially create a non-linear prediction.

Our **linear regression algorithm can be generalized to higher degree, non-linear functions as well**.

Generalization Error

The problem with the canonical machine learning problem (discussed earlier, is the following:

$$\underbrace{\sum_{i=1}^{m} l((h_\theta(x^i)), y^i)}_{\text{minimize}}$$

The whole point is to minimize the error on the *training* data set, not on the *validation set*. I.e. we're optimizing for the given training data by definition, not the new data that will be provided.

Training vs. Generalization Loss

Here, we should make an important distinction between training loss and generalization loss.

As an example, as we increase the degree of our regression function, from 1 to 2 to n, we have to understand that **at a certain point although we become more and more accurate in fitting a function to the training data, we start losing accuracy with new data that we need to classify.**
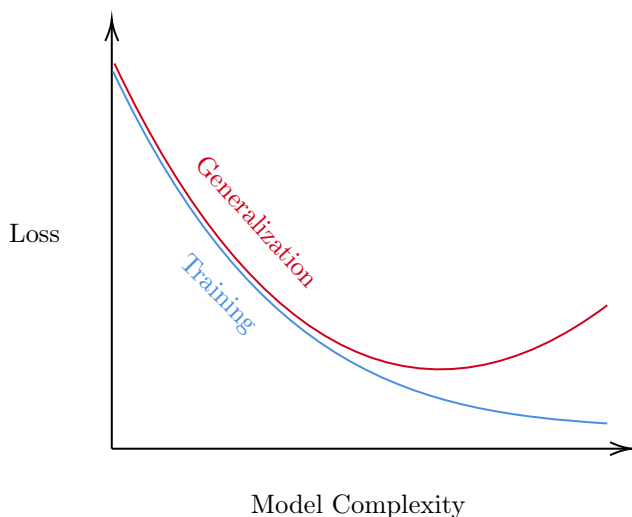
Cross-Validation

Although it is difficult to quantify the true generalization error, we can approximate it using **holdout cross-validation**.

We want to split the dataset into a training set and a 'holdout' set- we train our algorithm on the training set and evaluate on the holdout set.
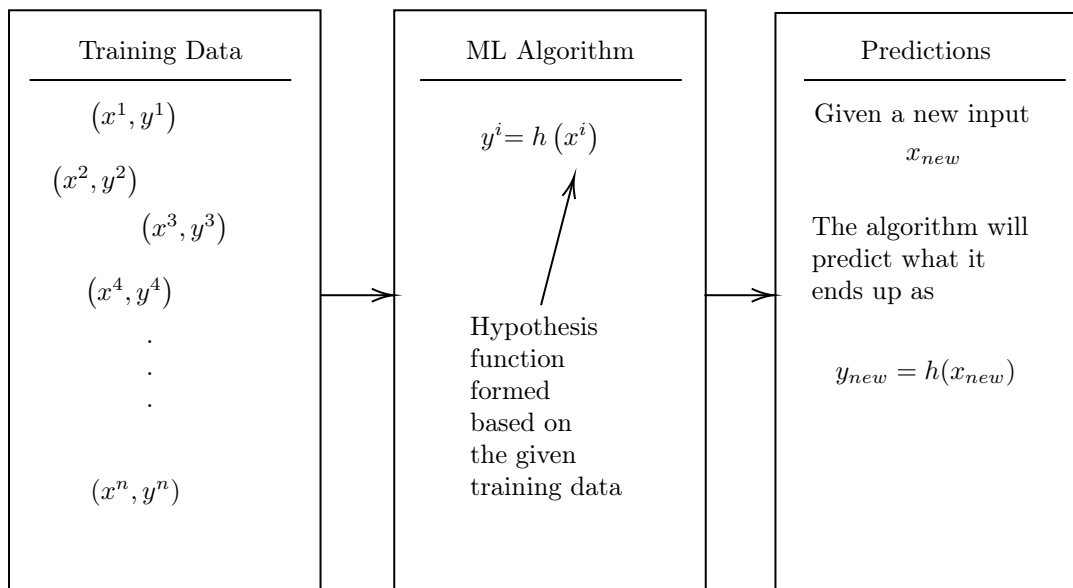

Regularization

We have previously seen that the degree of a polynomial acts as a natural measure of the 'complexity' of a model: **higher degree polynomials are more complex**.

However, we can notice that fitting these models also requires **extremely large coefficients on these very high degree polynomials**. Thus, we can look at an alternative way to control model complexity:
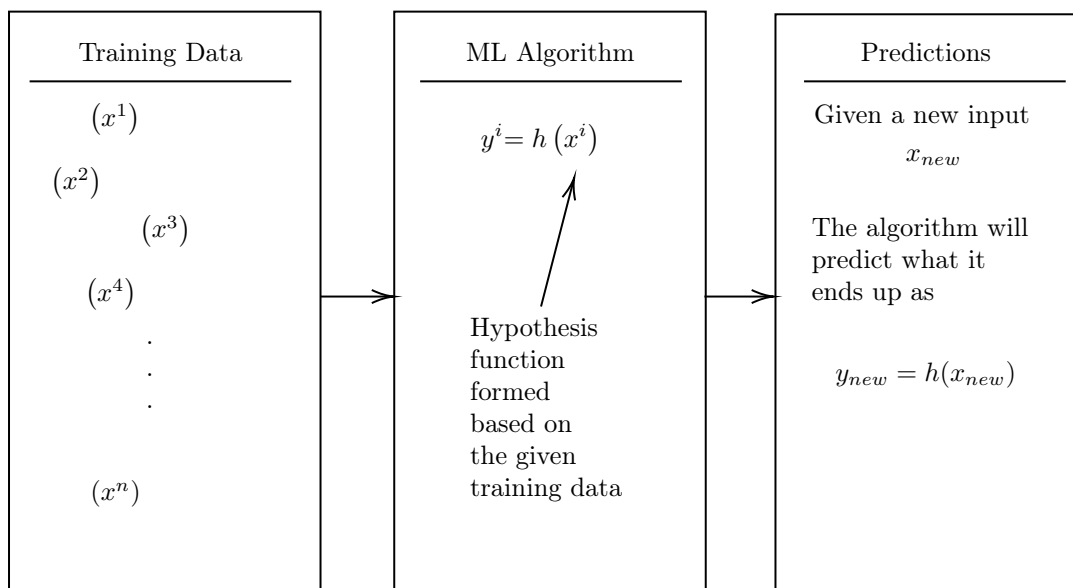
by keeping weights small.

Unsupervised Learning

Previously, we learned about the Supervised Learning paradigm. Here it is again.

Training Data	ML Algorithm	Predictions
(x^1, y^1) (x^2, y^2) (x^3, y^3) (x^4, y^4) . . . (x^n, y^n)	$y^i = h\left(x^i\right)$ Hypothesis function formed based on the given training data	Given a new input x_{new} The algorithm will predict what it ends up as $y_{new} = h(x_{new})$

Now, we can introduce the Unsupervised Learning paradigm. It's basically **equivalent to the Supervised Learning paradigm, but our training data no longer has valid answers mapped to it**.

Training Data	ML Algorithm	Predictions
(x^1) (x^2) (x^3) (x^4) . . . (x^n)	$y^i = h\left(x^i\right)$ Hypothesis function formed based on the given training data	Given a new input x_{new} The algorithm will predict what it ends up as $y_{new} = h(x_{new})$

In a less formal sense, virtually all unsupervised learning algorithms can be considered in the same manner as supervised learning.

Define a hypothesis function, define a loss function, and define how to optimize the loss function. The ques-

tion is: **What do the hypothesis function and loss function signify in unsupervised learning?**

The goal of unsupervised learning is to pick some restricted class of hypothesis functions that extract some kind of **structure** from the data that isn't explicitly given to us. Examples of this type of algorithm are K-means and Principal Component Analysis.

K-Means Clustering

The K-Means Algorithm is easy to visualize. If we are given some collection of data points, we want to find k 'center' points such that all points are close to at least one of these 'centers'.
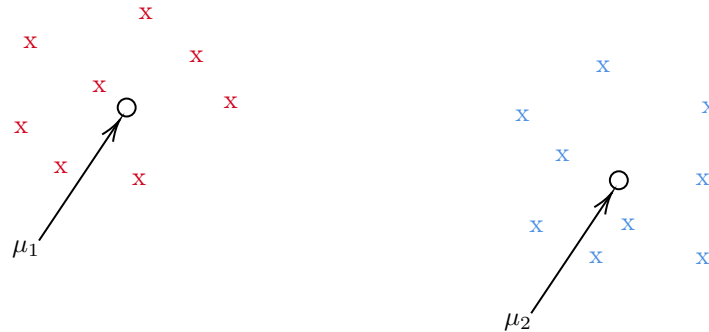
Here, The centers that we have identified in this 2-means clustering variant are labeled with μ. he 'centers' are known as centroids. We can formally define clusters and centroids here.

Definition:

Cluster $\rightarrow$ A collection of data points aggregated together because of certain similarities.
Centroid $\rightarrow$ The imaginary or real location representing the center of the cluster.

How K-Means fits Into the Unsupervised Framework Paradigm

- **Parameters** of K-Means are the choice of centers $\theta = \{\mu_1, ...\mu_k\}$ where $\mu_i \in \mathbb{R}^n$.

- **Hypothesis** function is a function that decides the center closest to a point x. Formally:

$$h_\theta(x) = \underbrace{argmin}_{\mu \in \{\mu_1, ...\mu_k\}} ||\mu - x||_2^2$$

- **Loss function** is squared error between input and hypothesis. Formally:

$$l(h_\theta(x), x) = ||h_\theta(x) - x||_2^2$$

- **Optimization problem** will therefore be:

$$\underbrace{minimize}_{\text{for } \mu_1, ...\mu_k} \sum_{i=1}^{m} ||h_\theta(x_i) - x_i||_2^2$$

93

K-Means Clustering Algorithm

To process learning data, the K-Means algorithm starts with a group of randomly selected centroids, which are used as the beginning points for every cluster, then performs iterative, repetetive calculations to optimize the positions of the centroids.

It **halts** when either the **centroids have 'stabilized'**, meaning that there is (basically) no change in their values because the clustering has been successful, or **when the defined number of iterations has been completed**.

How Do We Select k?

There is no *right* way to select k. Note that **a larger k virtually always will have lower loss than smaller k**, even on a hold out/validation set.

Insetad, it's common to look at the loss function as a function of increasing k, and then you just stop when things look "good".

There are other heuristic ways to do this, but they actually don't convincingly outperform this simple method.

Example of K-Means Clustering

K-Means Clustering is just an example of Unsupervised learning at the end of the day, so this is also a great example for unsupervised learning as well.

The following example is K-Means Clustering performed on the MNIST digit classification dataset. Here are some examples from the actual dataset:
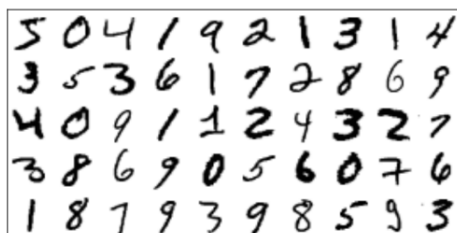
K-Means can be run with $k = 50$ on this given dataset, and here's the results it produces.
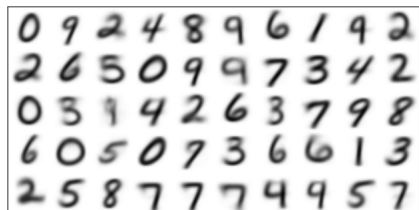
It is evident that this data was just categorized, and not 'classified' under any label. Such is the nuance of unsupervised learning.

Principal Component Analysis

Large datasets are increasingly common and are often **difficult to interpret**. More on this excellent example (along with interactive components) can be found at the following website:

`https://setosa.io/ev/principal-component-analysis/`

Definition:

Principal Component Analysis → technique for reducing the dimensionality of large and complex datasets.

Principal Component Analysis, or **PCA**, accomplishes this by creating new (uncorrelated) variables that successively maximize variance.

It provides us the answer to the question: **Can we combine raw features into new features that yield a simpler description of the same system?**

3-Dimensional Example

PCA really reveals its usefulness in 3 dimensions, because it's hard to see through a cloud of data. In the following example, the original data are plotted in 3d.
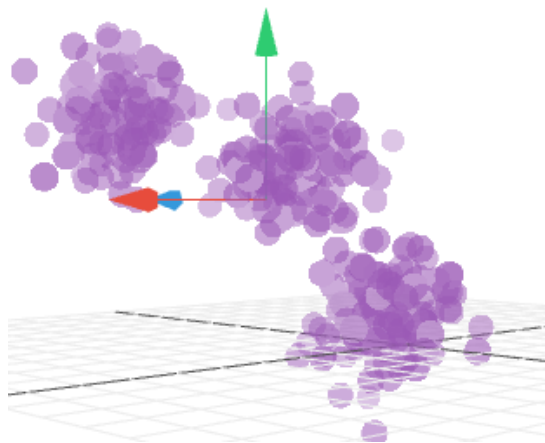
You can project this data into 2 dimensions using a transformation. (This particular transformation is no different than finding a different camera angle: rotate the axes to find a better angle.)

So what's this 'better' angle? Usually, the one with the most variance. Here's a visual explanation. Here's the above dataset projected into 2 dimensions (unchanged).
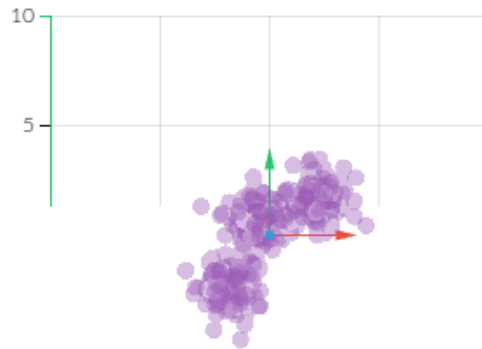
Our goal is the apply a transformation (adjust the camera angle) such that the 3 dimensions that are available to us are arranged as such:

- The horizontal axis, PC1, has the most variation

- The vertical axis, PC2, has the second most

- The third axis, PC3, has the least

Once we've found this optimal camera angle that ensures these three conditions are met with the most variance in PC1 and the least in PC3, we have something like this:
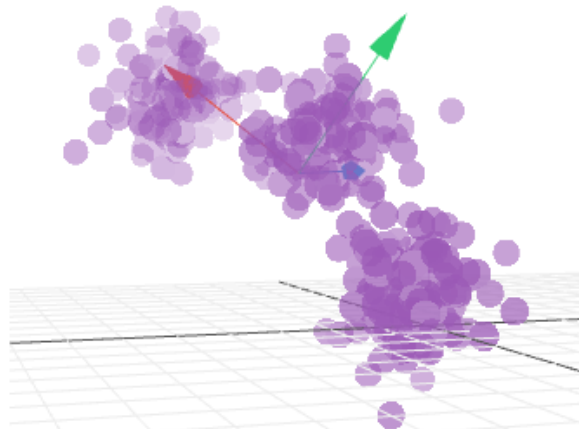
The real beauty of this transformation can be seen when we return to the two dimensional point of view, where it can be observed that we've maximized variance.
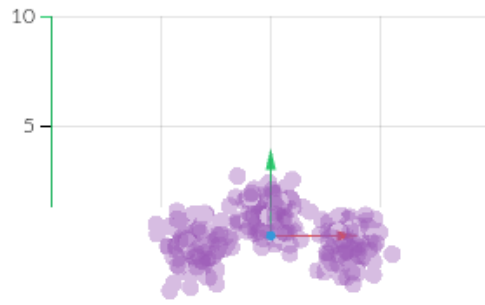
From here, in order to reduce dimensionality from 3 to 2, we simply just drop PC3. Below is an alternative visualization of the data.

First, let's examine the variance given the default configuration of the axes, with no PCA applied.
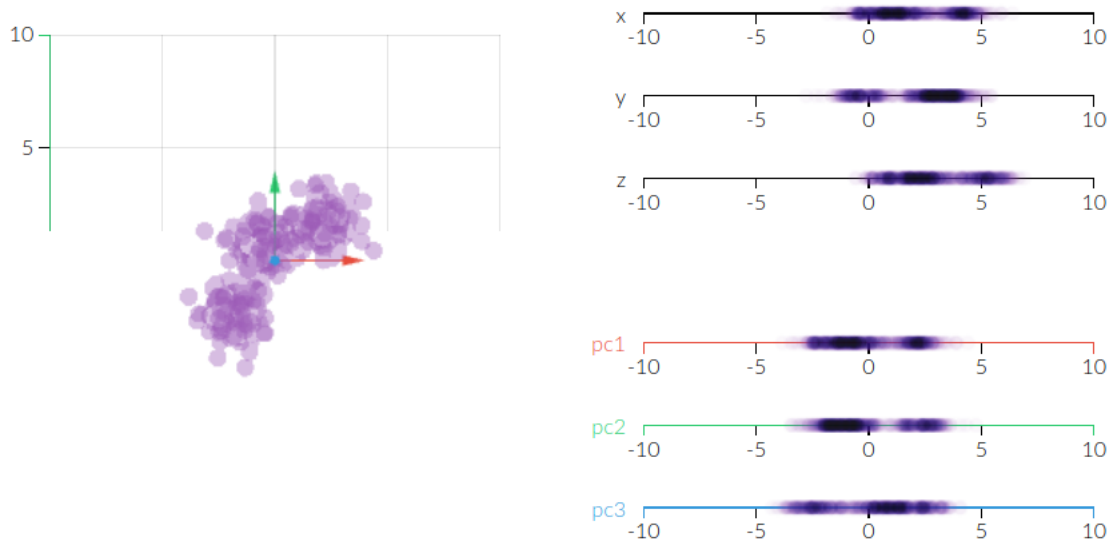
After the transformation, we can see that with the new axes specified by PCA, the variance in PC1 is maximized.
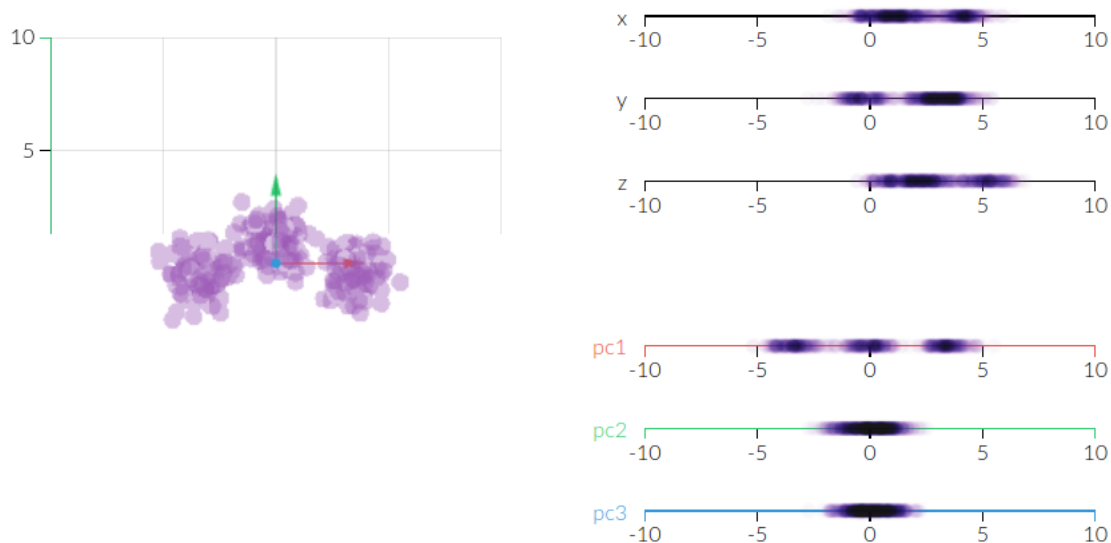
17-Dimensional Example

More realistically, our data will have far more dimensions (and thus, it won't be such a huge and clear loss of information to move down a few dimensions, relative to what we're given.)

In the given table is data for the average consumption of 17 types of food per person per week for every country in the U.K. (*grams per person*)

	England	N Ireland	Scotland	Wales
Alcoholic drinks	375	135	458	475
Beverages	57	47	53	73
Carcase meat	245	267	242	227
Cereals	1472	1494	1462	1582
Cheese	105	66	103	103
Confectionery	54	41	62	64
Fats and oils	193	209	184	235
Fish	147	93	122	160
Fresh fruit	1102	674	957	1137
Fresh potatoes	720	1033	566	874
Fresh Veg	253	143	171	265
Other meat	685	586	750	803
Other Veg	488	355	418	570
Processed potatoes	198	187	220	203
Processed Veg	360	334	337	365
Soft drinks	1374	1506	1572	1256
Sugars	156	139	147	175

We can glean some information from this table, and it shows us some interesting variations, but **overall differences aren't very notable**. PCA may be able to help us eliminate dimensions to help us emphasize how countries differ.

E.g. here is the plot of the data for the first principal component. We can see that Northern Ireland is looking different from the others.

(This is a number line, as we're only observing a one dimensional slice of the 17 dimensional data)

We can take this analysis further by looking at the first and second principal components.
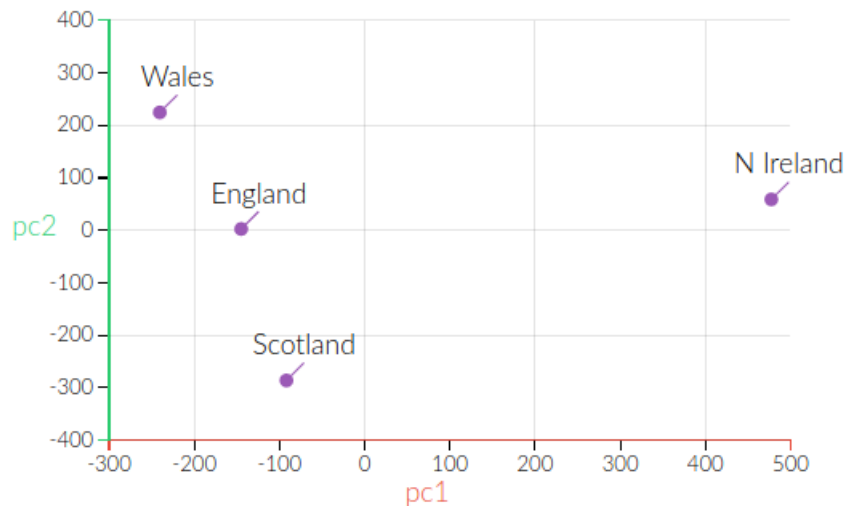
We can see that in the first and second principal components, Northern Ireland is a major outlier. This makes sense: the Northern Irish have such differing diets from the rest of these countries because it's the only one of the four countries not found on the island of Great Britain.

We've come to a conclusion that reflects real-world geography.

General Notes

Unsupervised learning methods are useful for Exploratory Data Analysis. You're able to cluster or reduce your data to a few dimensions and visualize. Usually, this stuff is a means to an end, not an end within itself.

Examples:

- Run PCA and transform your data to optimize for variance while reducing dimensions, then run an appropriate supervised classifier.

- Use k-Means to set up radial basis functions by getting k centers, then creating the RBF Features as needed.

Recommender Systems

In simple terms, a recommender system is just a system that predicts a user's rating of an item.

Little bit of a mix between supervised learning and unsupervised learning.

- Like **Supervised Learning** in that we have some semblance of labels there

- Like **Unsupervised Learning** in that we are trying to find some latent structure within the data

Generally, this sort of analysis falls somewhere in between information filtering and retrieval. However, this **does not stop us** from phrasing the problem in terms of hypothesis classes, loss functions, and optimization problems.

Collaborative Filtering

> **Definition:**
>
> **Collaborative Filtering** $\rightarrow$ recommender systems that predict based only on the expressed preferences of other users for an item.

Overview

Collaborative filtering is a technique that can filter out items that a user might like on the basis of reactions by similar users. It works by searching a large group of people and finding a smaller set of users with tastes similar to a particular user.

Note the following visual example, where users 1 through 4 have ratings for items 1 through 4 if applicable.

	i_1	i_2	i_3	i_4
u_1	1			3
u_2		2	5	
u_3		3		5
u_4	4		4	

Columns are items

Rows are users

The blanks indicate where users have not left ratings. very simply, our job is to figure out what these users *could* rate the items that they haven't rated. In other words, we are **predicting what they will**

rate items based on the expressed preferences of others.

	i_1	i_2	i_3	i_4
u_1	1	?	?	3
u_2	?	2	5	?
u_3	?	3	?	5
u_4	4	?	4	?

Columns are items

Rows are users
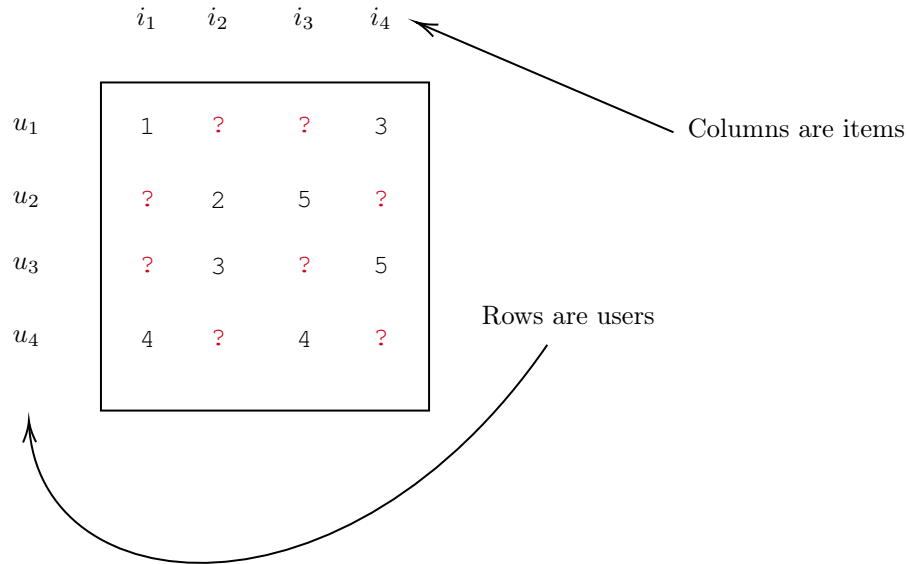
The basic goal is to **fill in the matrix** as best we can. In terms of data structures, this is usually stored as a sparse matrix.

Approaches

There are three main ways to approach Collaborative Filtering. **User-user**, **Item-item**, and **Matrix factorization**.

- **Item-Based CF** → Here, we want to calculate the similarity between two items and then decide that users will rate it based on that. **We have done this before in a different context.** First, we should **only consider users who have rated both items**. For each user in that list, we should calculate the difference in ratings for the two items, and then take the average of this difference over the users. We **can also find the cosine similarity of these items**.

- **User-Based CF** → This approach is very similar to the previous one, we just do the converse of it.

- **Matrix Factorization** → This one's a little more formal in nature. We **want** all the entries i, j of the ratings matrix $X \in \mathbb{R}^{m \times n}$.

> **Idea:** Approximate this as $u_i^T \cdot v_j$, where $u_i \in \mathbb{R}^k$ represents the user-specific weights (we're using the transpose of this matrix), and $u_j \in \mathbb{R}^k$ represents the item-specific weights.
>
> **Hypothesis function:** We want this to produce a matrix built to our specifications, i.e. $h_\theta(i, j) = u_i^T \cdot v_j$
>
> **Loss function:** Least squares on observed entries, i.e. $l(h_\theta(i, j), X_{ij}) = (h_\theta(i, j) - X_{ij})^2$
>
> **Optimization problem:** Where S is the observed entries, we need to minimize

Transaction ID	Items Purchased
1	Bread, Milk
2	Bread, Diaper, Beer, Eggs
3	Milk, Diaper, Beer, Coke
4	Bread, Milk, Diaper, Beer
5	Bread, Milk, Diaper, Coke

θ on

$$\sum_{i,j \in S} l(h_\theta(i,j), X_{ij})$$

Matrix factorization as a problem is known to not be a convex function, i.e. there's no one optimal solution that we're going to nail every time. Therefore, we won't go for perfect results.

Essentially, we are factorizing the ratings matrix X as $X = UV$ where $U \in \mathbb{R}^{m \times k}$ and $V \in \mathbb{R}^{k \times n}$ but only penalizing mismatches between UV and X at the observed entries in X.

Association Rules

We can also find rules that **associate** the presence of one set of items with that of another set of item.

An excellent example of this is the Amazon 'Customers also bought' section.

Association rules are usually represented in the form $X \rightarrow Y$, where X (rule antecedent) and Y (rule consequent) are disjoint conjunctions of features. Rule quality is usually measured by rule support and confidence.

> **Definition:**
>
> **Rule support** $\rightarrow$ The percentage of records containing both X and Y.
>
> **Rule confidence** $\rightarrow$ The conditional probability of finding Y given X, i.e. the conditional probability that a transaction will contain Y given that it contains X. It describes the strength of the implication, and is given by
>
> $$c(X \rightarrow Y) = \frac{s(X \rightarrow Y)}{s(X)}$$

Examples

- buys(x, bread) $\wedge$ buys(x, eggs) $\rightarrow$ buys(x, milk) <support: 0.6%, confidence: 65%>

- age="30-45", income="50k-75k" $\rightarrow$ car="SUV"

Basic Concepts of Association Rules

To explain these concepts, we'll make use of the following table.

- **Itemset** $\rightarrow$ A set of one or more items. E.g. {Milk, Bread, Diaper}.

- **K-itemset** $\rightarrow$ An itemset that contains k items.

- **Support Count** (σ) $\rightarrow$ Frequency of occurrence of an itemset- the number of transactions in which it appears. E.g. $\sigma(\{$Milk, Bread, Diaper$\}) = 2$

- **Support** $\rightarrow$ Fraction of the transactions in which an itemset appears. E.g. s({Milk, Bread, Diaper}) = $\frac{2}{5}$

- **Frequent Itemset** $\rightarrow$ An itemset whose support is greater than or equal to a minsup threshold.

Miscellaneous

There are other considerations of how 'interesting' a rule is, including **lift**, **conviction**, etc.

In Practice

`Orange3` is a GUI and Python API that enumerates frequent itemsets and performs association mining. It plays nicely with `sklearn`.

Generally, we can make use of Association Rules for interpretable, fast data mining. However, this sort of analysis typically doesn't consider order or scalability issues.

17 Big Data and MapReduce (L24)

Big Data

> **Definition:**
>
> **Big Data** $\rightarrow$ extremely large data sets that may be analyzed computationally to reveal patterns, trends, and associations, especially relating to human behavior and interactions.

There is no formal point where we say, 'oh, that is now **big data**.' However, we can look at a few notable inflection points in terms of how large your actual data is.

Note how we're going based on disk and RAM sizes. This is because the definition of big data will inevitably vary with the amount of data we can easily manage.

1. Your data fits in RAM on a single machine.

2. Your data fits in disk on a single machine.

3. Your data fits in RAM or disk on a small cluster of machines.

4. Your data fits in RAM or disk on a large cluster of machines.

It's probably reasonable to refer to 3 and 4 as big data, but many would only consider 4 as big data. Again, the definition varies.
Distributed Computing rose to prominence in the 70s and 80s, and was made possible in part by innovations in computer architecture like the **Message Passing Interface**, a key technology that allowed multiple computer systems to work together on problems.

MPI is extremely powerful but is complicated and brittle- it's imperfect.

MapReduce

Overview

When Google first started building their datacenters, they used clusters of off-the-shelf commodity hardware. Machines had different speeds and failures were common given cluster sizes.

Data itself was distributed redundantly over many machines, they **wanted as much as possible to perform the computation on the machine where the data was stored**.

This led to the development of the **MapReduce** framework at Google, which was made popular through the Apache Hadoop open source implementation.

Hadoop

Hadoop (owned by Apache) is an open-source framework designed for distributed storage and processing of very large data sets across clusters of computers. Apache Hadoop consists of components including:

- **Hadoop Distributed File System** (HDFS), the bottom layer component for storage. HDFS breaks up files into chunks and distributes them across the nodes of the cluster.

- **Yarn** for job scheduling and cluster resource management.

- **MapReduce** for parallel processing.

- Other libraries.

MapReduce Details

Initially used for analyzing search results, **MapReduce gained popularity due to its ability to split and process terabytes of data in parallel, achieving quicker results**.

At the core of MapReduce are two functions: Map and Reduce. Here are their definitions in relation to this algorithm.
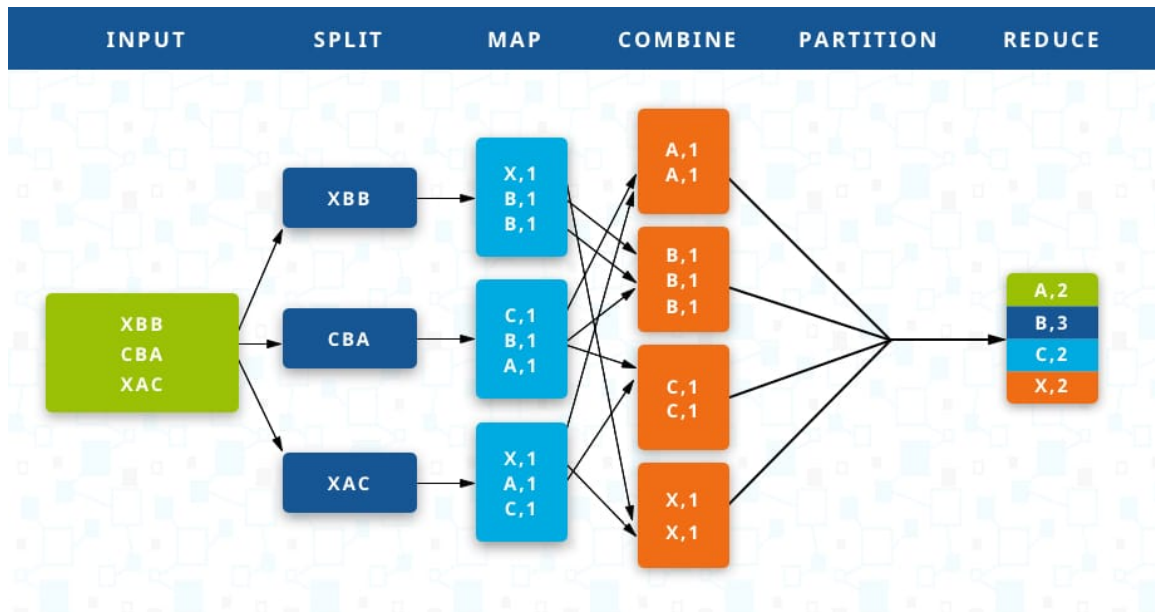
> **Definition:**
>
> **Map** → The mapper reads in data from the filesystem and outputs modified data.
>
> **Reduce** → The reducer collects all of the mappers' outputs on the keys, and then outputs 'reduced' data.

Mappers and reducers are the Hadoop servers that run the Map and Reduce functions respectively.

How MapReduce Works

- **Map** $\rightarrow$ The input data is first split into smaller blocks, and then each block is assigned to a **mapper** for processing. Mappers read in a chunk of data provided by the splitter (typically 64MB), and then emit a set of data derived from the input.

- **Combine** $\rightarrow$ This is an optional process. The **combiner** is a reducer that runs individually on each mapper server. It reduces the data on each mapper further to a more simplified form before passing it downstream.

- **Partition** $\rightarrow$ This is the process that translates the kv pairs from mappers to another set of kv pairs to feed into the reducer. It decides how the data has to be presented to the reducer and assigns it to a particular reducer.

- **Reduce** $\rightarrow$ After all the mappers complete processing, the framework shuffles and sorts the results before passing them onto the reducers. *Note that a reducer cannot start while a mapper is still in progress.* This is the last step in the process, and the dataset that reducers produce is typically written to disk.

Key Notes about Mappers/Reducers

- Mappers and reducers are typically single threaded and deterministic. Determinism allows for the restarting of failed jobs, or *speculative execution.*

- If you need to handle more data, you can just add more mappers or reducers.

  - No need to handle multi-threaded code.
  - Since these systems are all independent of each other, you can run a nearly arbitrary number of nodes.
  - A machine typically has a few nodes that can be allotted to mappers and reducers, it's usually one mapper or reducer per processor core.

- Mappers/reducers run entirely independently of each other. (in the case of Hadoop, they run in separate JVMs)

MapReduce's Pros and Cons

- **Pros**
  - All you need to do is write a mapper and reducer; they can then be scaled as much as you need.
  - Can get away with not exposing any of the internals if you're using a premade engine.

- **Cons**
  - Lots of reading and writing from disk
  - Sometimes, communication between processes is necessary

MapReduce in Python

Python's `mrjob` library allows you to write mappers and reducers in Python, which you can then deploy on Hadoop systems, Amazon Elastic MR, and Google Cloud.

18 More Probability + Statistics (L25)

> **Definition:**
>
> **Statistical Inference** → The discipline that concerns itself with the development of procedures, methods, and theorems that allow us to extract meaning from data taht has been generated by stochastic (random) processes.

It's like going from real life, to data, and then back to real life again. The goal is often **to develop a statistical model of the real world**.

Usually, we'll end up with one of the following.

- An estimate
- A Confidence Interval
- Rejection of a hypothesis
- Clustering or Classification of data points into groups

Review of Probability

Probability is concerned with the outcome of a *trial*. (i.e. experiment, observation)

Basics

> **Definitions:**
>
> **Sample space** → Set of all possible outcomes of a trial.
>
> **Event** → specification of the outcome of a trial.

As such, the probability of the sample space will always be 1. *(Recall that probability is measured on a float scale from 0 to 1)*

Example:

> Trial = Tossing a coin; Sample space = {Heads, Tails}; Event = Heads

If two events E and F are **independent** then the probability of E does not change if F has already happened. I.e. for **two independent events E and F:**

$$P(E|F) = P(E)$$

Additionally, if two events E and F are **mutually exclusive**, then:

$$P(E \cup F) = P(E) + P(F)$$

Bayes Theorem

Bayes Theorem is a very simple equation, but it's fundamental to **Bayesian Inference**.

Aside: Bayes' Theorem

Bayes' Theorem describes the probability of an event, based on prior knowledge of conditions that might be related to the event.

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}$$

Conditional Independence is the idea that events A and B are conditionally independent given C if the probability of A and B happening given that C happened is the same as the probability that A happened given that C happened, times the probability that B happened given that C happened. Formally:

$$P(A \wedge B|C) = P(A|C) \cdot P(B|C)$$

Entropy is a measure of the uncertainty in a probability distribution.

Hypothesis Testing

This is the process in which we **accept** or **reject** a statistical hypothesis about a population.

We will refer to H_0 as the null hypothesis, and H_1 as the alternative hypothesis.

Background Information

<div style="border: 2px solid #a00; border-radius: 8px;">

Definition:

Null Hypothesis → A statement about a population parameter. We can test the likelihood of this statement being true in order to decide whether to accept or reject our alternative hypothesis. *Can include* $=, \leq, or \geq sign$.

Alternative Hypothesis → A statement that directly contradicts the null hypothesis. We determine whether or not to accept or reject this statement based on the likelihood of the null hypothesis being true. *Can include* $=, <, or > sign$.

P-value → The probability of obtaining the observed results of a test, assuming that the null hypothesis is correct; a smaller p-value means that there is stronger evidence in favor of the alternative hypothesis.

Alpha → The significance level; the probability of rejecting the null hypothesis when it is true. Also known as the *Type 1 Error*.

</div>

To elaborate, the **null hypothesis** is the hypothesis that sample observations result purely from chance. The null hypothesis tends to state that there is no change. The **Alternative hypothesis**, on the other hand, is the hypothesis that sample observations are influenced by some non-random cause.

We can map these definitions to the following example from *towardsdatascience* for better understanding.

> Imagine that you and your friend play a game. If a coin lands on heads, you win \$5 and if it lands on tails he wins \$5.
>
> Let's say the first two coin tosses landed on tails, meaning your friend won \$10. Should you be worried that he's using a rigged coin? Well, the probability of the coin landing on tails two times in a row is 25% (see above) which is not unlikely.
>
> What if the coin landed on tails six times in a row? The probability of that occurring is approximately 1.56% (see above), which is highly unlikely. At this point, it would be fair to assume that the coin is rigged. Typically, one would set a threshold, usually 5%, to determine if an event occurred by chance or not (if you learned this before, this is known as the alpha!)
>
> In this case:
>
> - The **null hypothesis** in our example is that the coin is a fair coin and that the observations result purely from chance
>
> - The **alternative hypothesis** would then be that the coin is not fair, and that the scenario did not happen by chance.
>
> - The **p-value** in the scenario of flipping tails 2 times in a row is 25%, (it's 1.56% when you're flipping tails 6 times in a row)
>
> - The **alpha**, or level of significance would be 5%.

Accepting and Rejecting

Given some data x, we want to accept or reject the null hypothesis in favor of the alternative hypothesis. If H_0 is true, but you reject it, you have a Type I error (false positive). If H_1 is true, but you accept H_0, you have a Type II error (false negative).

Rejecting vs. Not Rejecting the Null

The main rule in deciding whether or not you reject the null hypothesis is **PGADRN**.

If the **P** value is **G**reater than the **A**lpha, **D**o not **R**eject the **N**ull.

> In the case of flipping tails 2 times in a row, we **would not** reject the null, since 25% (the chance of that case happening) is greater than 5% (our α value).
>
> In the case of flipping tails 6 times in a row, we **would** reject the null hypothesis, since 1.56% < 5% (our α value).

What's the Point?

Now we understand the terminology and the use through an example, so what's the point of this process?

It's used **to determine how likely or unlikely a hypothesis is for a given sample of data**. That last part is key, because more often than not, you won't be able to get an infinite amount of data, or even data that represents the *entire* population.
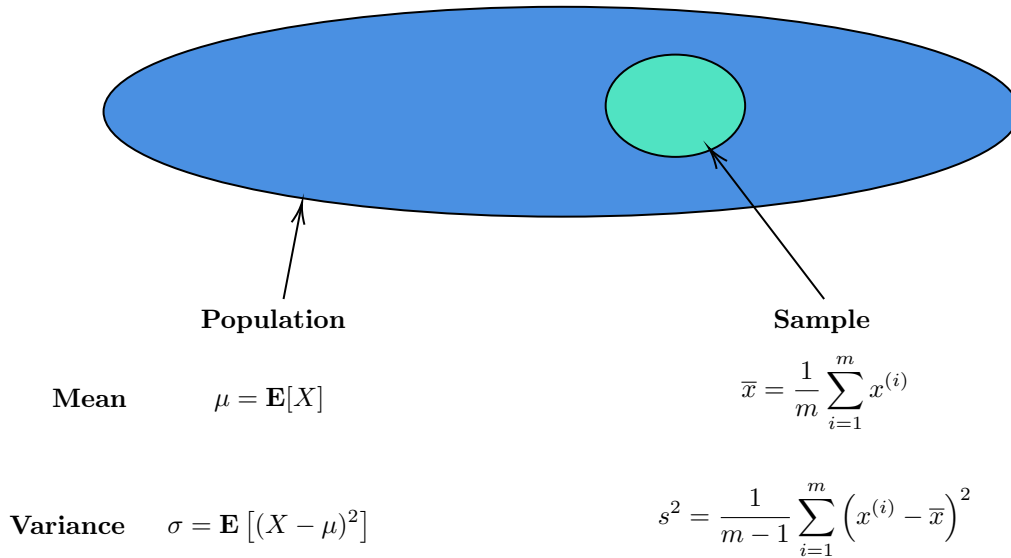
General Steps

Here are the steps for performing a hypothesis test in a general sense.

1. State your null and alternative hypotheses. *Usually, null states that everything is as normally was; that nothing has changed; alternative states that something has indeed changed.*

2. Set your significance level, the α. *This is at which point you say "okay, the probability that this event occurred is a bit abnormal".* (Usually set to 5%)

3. Collect sample data and calculate sample statistics.

4. Calculate the p-value given those sample statistics. Once you get the sample statistics, you can determine the p-value through different methods. Most commonly, we use T-score and Z-score given a normal distribution of data.

5. Reject or do not reject the null hypothesis. (Using **PGADRN**)

Sample Statistics

To be more consistent with standard statistics notation, we will introduce the notion of a *population* and a *sample.*

	Population	Sample
Mean	$\mu = \mathbf{E}[X]$	$\overline{x} = \dfrac{1}{m} \sum\limits_{i=1}^{m} x^{(i)}$
Variance	$\sigma = \mathbf{E}\left[(X - \mu)^2\right]$	$s^2 = \dfrac{1}{m-1} \sum\limits_{i=1}^{m} \left(x^{(i)} - \overline{x}\right)^2$

Statistics + Hypothesis Testing: Big Picture

Using these basic techniques, we can devise some tests to determine whether certain data gives evidence that some effect 'really' occurs in the real world.

Fundamentally, we are deciding whether **things are likely to be true about the population, given a sample to analyze**.

There are lots of caveats about the precise meaning of these terms, to the point that **many people debate the usefulness of hypothesis testing at all**.

Despite this debate, hypothesis testing is still very common in practice, and is important to understand.

A/B Testing

When we're **comparing two samples**, we want to compare the values of sampled individuals in Group A with the values of samples individuals in Group B.

Here, we ask: **Do the two sets of values come from the same underlying distribution?** Do Groups A and B point us in the same direction, statistically?

Answering this question by performing a statistical test is called **A/B Testing**.

Assume the following example:

> Assume we have a random sample of mothers and their newborns. Let us compare:
>
> - **Group A:** Birth weights of babies of mothers who smoked during pregnancy
> - **Group B:** Birth weights of babies of mothers who didn't smoke
>
> **Question:** Can we say that the difference is due to chance alone.

From here, let's assume some hypotheses as we approach this statistically.

> **Null Hypothesis (H_0):** In the population, the distributions of the birth weights of the babies in the two groups are the same. (They are different in the samples only due to chance)
>
> **Alternative Hypothesis (H_1):** In the population, the babies of the mothers who didn't smoke were heavier, on average, than the babies of the smokers.

Now, let's establish a test statistic. Here, we will use the difference between the average weights of the babies. In other words:

$$Avg_B - Avg_A$$

The larger this statistic is, the more we favor the alternative hypothesis.

Idea: Simulating Under the Null

If the null hypothesis is true, we can just **shuffle all the birth weights around, between group B and A, and the test statistic should remain the same**.

Let's **first** find the test statistic using the given data for A and B, then let's **shuffle and try again**.

19 Footnotes

Taken and compiled by Akilesh Praveen.

I don't claim ownership to any of the material in this notesheet. It is merely a compilation of resources I used to understand materials presented in this course. If a resource needed simplifying in my mind, I tried to do so to the best of my abilities. If not, I left it as it was.

References:

- CMSC320 - John Dickerson - UMD
- DATA 8 - Ani Adhikari, John Denero - UC Berkeley
- Geeksforgeeks.com
- Wikipedia.org
- Hackerearth.com
- Softwaretestinghelp.com
- CSE 591 - Stonybrook University
- Setosa.io
- Talend.com
- Mapr.com
- Towardsdatascience.com