Data Lexicon

The following is large, thematic lexicon, of various terms relating to the domain of "data". It is mostly oriented to individuals trying to build or improve a technical profile in the domain. However, it was designed to also be of interest to anyone that might need to work with technical data profiles, even if they themselves might not be one (business, HR, legal, designers, etc).

It includes vocabulary from mathematics (everything from combinatorics to topological data analysis), computer science (programming paradigms to systems engineering), machine learning, professions in data science, etc. As such, it can also serve as a general introduction to computer science.

Code for the level of various terms

Below is a code to understand the relative difficulty or importance of various terms in this lexicon. Note that even if a term is "above your level of understanding or ambition", it might still be useful to have heard of it. Also note that some terms might be fundamental for one profile, but only necessary later on for others (typically, all of linear algebra); this will be described through letter combinations.

For example "(AM, BE)" would mean fundamental to mathematicians, but intermediate-level for engineers. "A", alone, would mean fundamental for everyone, in terms of general culture or being an informed citizen; "AMELT" would mean fundamental for all technical profiles; while "AMELTO" would mean fundamental to anyone required to communicate regularly with a technical profiles, even if they are not technical themselves.

Difficulty / importance code:

- **A**: fundamental or easy, to be known and understood for any kind of work profile in data that you're trying to build or improve (even non-technical ones).
- **B**: important or intermediate, to be understood when you start to attempt productive work in the domain, or needed to follow basic conversations with colleagues about their domain.
- **C**: advanced or difficult, to be understood by those trying to build a specialist profile.
- **D**: very advanced or very difficult, to be understood by those trying to build an expert profile.

Thematic code:

- **M (math)**: applies to profiles which rely on mathematics a lot (data science, statistics, ML engineering, ML research, algorithmics research, etc).

- **E (engineer)**: applies to profiles which rely on computer engineering or software engineering a lot (systems engineering, network engineering, distributed computing, etc).
- **L (learning)**: applies to profiles which rely on machine learning, algorithmics and statistics, and their specific techniques, outside of general mathematics.
- **T (technical)**: applies to other profiles which may have an interest in this document, and have a technical profile (physical modelling, financial analysis, business analysis, etc).
- **O (other)**: applies to any other profiles which may have an interest in this document (designers, illustrators, data collectors, lawyers, businesspeople, etc), but do not themselves have a technical profile (though they may need to communicate with such people).
- **G (general)**: interesting to know, but peripheral to the subject of data (general culture).

Data generalities

Data generalities: fundamentals

- **Data (A)**: "data" describes any form of raw information about the world (real or virtual) that can be measured, collected, described, transmitted, organized, transformed, modelled, and/or analyzed. The human goal for "data" is to turn it into "analysis" (semantic information, information in the "non-raw" sense), in order to aid human decision-making by improving our understanding the dynamics of our world and its various component systems.
- **Domain / Business domain (A)**: a "domain" is a field of human activity, generally one of business activity. This includes everything from the management of shipping ports, to mariage counseling, to meteorological modeling. "Domain knowledge" refers to the ways through which people who are active in this domain tend to describe it; this includes all the vocabulary, processes, or mental models specific to actors of that domain.
- **System (A)**: a "system" is collection of "**entities**" (or "agents", or "nodes") and the "**interactions**" (or "links", or "edges") between these entities. In a system, you generally have various collections of entities which can be grouped together (because they are of the same "type": while distinct, they are analogous to each other, like "atoms", or "humans"). Different kinds of interactions exist between each other based on their respective types, though generally, interactions between elements of type A and elements of type B can be described by a common "interaction type" A-¿B (like "covalent bonds" between atoms, "relationships"

between humans, or "ownership" of an object by a human). "Systems engineering" in particular, generally refers to the specific case of information systems / computer systems, and the network interactions between these computers, or their interfaces with their users.

- **Model (A)**: a "model" is a mathematical way of describing some entity, or set of entities, or domain; within the real or virtual world. Nowadays, all models (which used to be mostly abstract tools) tend to become implemented into computers somehow. Models insist in different ways on different mathematics. Some are specialized for statistical analysis and/or predictions, some are specialized to describe how an information system is meant to be designed, some are specialized in describing existing non-information (such as physical) systems, etc. Generally, when building an overall system for a specific domain, there are various types of models which are required, and these need to be associated, designed in common, or linked together for the overall domain to be modeled properly.
- **Big data (A)**: this term is somewhat hard to define cleanly, and is used more as a marketing buzzword, rather than having an actual technical definition. After all, how "big" is "big" ? Is a spreadsheet with 10 million columns "big" ? Analyzing it would have required dozens of computers some decades ago; nowadays a single average laptop can typically handle it. We can perhaps divide the expression into its two subterms: big and data. "Big" would then generally refers to "big enough to require a large-scale information (computer) system, and/or algorithms that are used to handle extremely high volumes of data, or dimensions of data". Data is defined above. In practice, in this sense, "big data (analytics)" is actually comparatively rare: "big" alone (just needing a distributed information system for some use case, but where data analysis is not the central goal of the system) or "data" alone (just running algorithms on moderate volumes of data; even complex algorithms that also work on high-dimensional data) are much, much more frequent in practice.
- **Pipeline / Data pipeline (A)**: a "data pipeline" is a term used to describe the set of structures and processes that allow humans to go from a raw measure of some domain to analyses of this domain. One should generally consider the following steps or aspects of a data pipeline.
  - **harvesting / ingestion / collection / extraction**: this 1st step consists of obtaining raw data. This encompasses multiple aspects. Firstly, the choice of what's interesting and useable, metric-wise: this is mostly an exercise in modeling and understanding what is necessary to feasibly answer domain-specific questions. Secondly, the building of some tool that's capable of data extraction (this includes everything from survey forms, to mechanical thermometers). Finally, the agglomeration of these harvested metrics into some entity in the system, so that it maybe retrieved or transmitted later for further processing.

- **wrangling / preprocessing / transformation**: this 2nd step consists of preparing the raw data so that it can be used properly. A mathematical-programmatic model is chosen to represent the data. Raw data often has issues (inconsistent formatting, outlying errors due to failures of measurement, improper modeling by someone not conscious of the requirements of the rest of the pipeline or analytics) and these issues need to be resolved before that data can be used. Once the preprocessing is done, the data generally goes through a validation phase.
- **systems engineering / deploying / linking / networking / servicing**: this 3rd step consists of ensuring that setting up and managing access to the system, both internally and externally, is properly handled. This includes multiples aspects, such as:
  * handling updates consistently (both to the data and the system itself);
  * referencing content (such as setting up registry tables or distributed hashmaps);
  * selecting the right hardware for the right role (storage, compute, caching, load balancing, logging, archiving, testing, monitoring, etc);
  * network engineering (setting up, then optimizing, the interactions between machines);
  * making sure only the appropriate parts of the system are accessible to the right actors (security analysis, system administration, pentesting);
  * providing structured ways of speaking to the system for it to act on our behalf (such as APIs or query languages);
  * making sure that the system can evolve (scalability, content distribution, code quality audits);
  * monitoring, maintaining, and improving the overall health, performance and functionality of the system (end-to-end testing, logging, tracing, automatic fault recovery, audits, making backups, archiving, disaster recovery planning, vacuuming...).
- **storing / loading**: this 4th step consists of putting the well-formatted, well-modeled, cleaned data into the appropriate databases and computers. There are various tradeoffs, mostly engineering-wise, that should be considered for this step. One can understand this step as "translating" an abstract mathematical model (such as a relational table, initially represented as some spreadsheets) into a specific technological architecture and implementation (such as an SQL database based on PostgreSQL). This step should above all consider how best to distribute the volume of data so that it can be accessed efficiently.
- **consumption / exploration / analysis**: this 5th step consists in actually using the data to produce useful (non-raw) "information":

analyses. Generally one first starts with an exploration of data, trying to understand its various attributes, and its underlying geometry, using various visualization tools (such as software libraries like Matplotlib, or various algorithms like bar charts or more advanced ones, like doing a dimensionality reduction (e.g., PCA) first). Then, analyses are run. For example, a data scientist than elaborates hypotheses over the data, meant to test *a priori* domain questions, or answer questions / remarks that arose from the process of exploration. In other cases, an ML engineer can choose an appropriate neural network model and define its topology in order to train models over the dataset. Finally, isolated reports or dynamic software that helps communicate or make this data useful for human decision-making, or to help process automation.

Note, however, that these steps are not neatly separated, nor neatly sequential. The above is a neat abstraction that allows one to have a simple mental model of the various factors required for a data pipeline. Dividing "data pipelines" into the above does not mean that all systems neatly follow the above division; far from it. You may very well have some ingestion immediately happen into a pre-existing, already well-networked system. You may have a company where the wranglers are also the data architects and write a single piece of software that goes from dirty raw data to everything being neatly ingested into a distributed database. The networking step can be done before the loading step if we know in advance the amount of data to ingest is going to be massive and are building a system from scratch; but it can also be done after, if we're scaling an existing small system into a bigger system. If the data is stored improperly, because the model was poorly thought-out, some wrangling is generally necessary before analysis. The list of exceptions and caveats goes on and on; however, given the overall complexity of modern systems, the above is still an excellent mental model to keep in mind, if one wants to have a simple set of questions to ask themselves when trying to analyze or design a system.

Also note that there are peripheral steps that don't necessarily integrate into the pipeline *per se*, such as running a prior analysis of the domain and prototyping models and systems on paper before even building anything; or the study of the pipeline itself in order to improve it over time; or documenting the pipeline for new workers.

Data generalities: professions

This is a (non-exhaustive) list of professions, designed to give you an idea of the various roles that are available when dealing with data.

- **Data wrangler (A)**: individual knowing how to create, synthesize, prepare, normalize, and/or homogenize data so that it is coherent and can be processed by algorithms.

- **Data engineer (A)**: individual knowing how to store, distribute, and route data so that it can be processed by algorithms on specific computing machines. They implement pipelines, and ensure data pipelines are efficient, scalable, and reliable.
- **Data analyst / Data scientist (A)**: individual who knows how to analyze input data, formalize a problem and elaborate hypotheses, choose an algorithmic protocol for data processing, provide visualization tools, and finally evaluate the quality of the predictive model generated by the algorithm. Their work is then used to aid strategic decision-making.
- **Data architect (A)**: A catch-all term for data engineers or mathematicians who are tasked with designing data pipelines. It is generally a managerial, or at least, a high-responsibility position. One can generally distinguish between **systems architects** (who design overall information systems, generally more on the engineering side; themselves divided into network architects, computer architects, web architects, etc) and **database architects** (who design how best to mathematically model the data from a given real-world domain in order to answer its questiosn, and help define how this data should be stored and serviced). Obviously, these profiles still do have a lot of overlap in their skills and responsibilities, hence the catch-all term.
- **Machine learning researcher (A)**: individual who uses mathematical, data science, and engineering principles to invent new machine learning algorithms, or improve existing ones (performance improvement, broadening of the application domain, etc).
- **Machine learning engineer (A)**: individual who uses data science and machine learning principles to implement existing ML algorithms to concrete business cases, e.g., automatic optimal logistics management. They work on selecting algorithms, preprocessing data, tuning model parameters, and evaluating model quality.
- **Database administrator (A)**: individual esponsible for managing and maintaining databases, including ensuring data integrity, security, backup, and optimization for efficient data retrieval.
- **AI ethics officer (A)**: invidual dedicated to ensuring ethical and responsible development and deployment of artificial intelligence and machine learning systems.
- **Data governance manager (A)**: individual responsible for establishing and implementing data governance policies and procedures to ensure data quality, security, compliance, and ethical use.
- **Data privacy officer (A)**: individual who oversees data protection measures, ensuring compliance with data privacy regulations and policies to safeguard sensitive information.

- **Chief data officer, CDO (A)**: high-level executive responsible for driving data strategy within an organization, overseeing data management, analytics, and aligning data initiatives with business goals.
- **Chief technological officer, CTO (A)**: high-level executive responsible for information systems strategy within an organization, overseeing hardware and software decision-making, cost (money and time) vs quality tradeoffs, and communicating technological stakes with the more "business" side of the company. To caricature, he CTO is the "highest technical position" in a company.
- **Vice-president of engineering, VPoE (A)**: high-level executives responsible for developer and engineer management, overseeing hiring decisions of technical profiles, promotions, team-management, and company culture for technical profiles. To caricature, the VPoE is the highest "tech HR" role.

Computer Science

Computer Science: Programming

This section describes terms that are fundamental to understand how one can use a computer to automate processes and ideas.

- **Program (A)**: a program is a series of mathematical, software or electronics instructions or expressions; written either in a given programming language, or in machine code; stored as some file or set of files; and that can be executed as a common unit by a computer.
- **Algorithm (A)**: mathematical process capable of transforming inputs (data that is provided) into outputs (data that is generated), according to a determined and structured protocol.
- **Heuristic (BMELT)**: a technique of algorithmic design that allows one to trade precision or correctness of the solution against a faster calculation time. These are often used for very complex problems. Heuristic algorithms are algorithms that provide a "good guess" to a complex problem.
- **Programming language (A)**: a programming language is a way of writing text, usually a hybrid between English and mathematical language, to translate mathematical/algorithmic ideas into a form that an electronic computer can understand and execute.
- **Machine language (BMELT)**: a language based on the binary alphabet (containing series of just 2 symbols, '0' and '1'; or 'the current is blocked' and 'the current is flowing'), allowing a computer to process information automatically.

- **Type (AMELTOG)**: the category to which a given "real-world concept" belongs in a program. It is one of the most important concepts in computer science, both in practice and in theory. Mathematically, a type corresponds to the mathematical space to which a computer quantity belongs. We can distinguish, for example, the type "Integer" (signed integers), the type "Float" (floating point number, serving as an approximation of real numbers), the type "String" (textual character strings), "Bool" (true or false values), and their composites. Things like a "Player", or a "Color", in a video game are also examples of more complex types. Note that in what follows, the word "type" should be understood to have this specific definition.

- **Integer / Signed integer / Unsigned integer (A)**: an integer, or whole number, is a number that is used for counting. Unsigned integers, also called natural numbers, are the positive whole numbers (including zero). Signed integers, also called simply integers, are the positive and negative whole numbers (including zero). They are universal, and fundamental, to all programming languages.

- **Float / Floating point number (A)**: a floating point number is a number inside a computer meant to approximate real numbers (fractions, or general numbers with infinite fractional expansion). Floating point numbers rely on a special encoding that resembles scientific number notation, but instead based on powers of 2 rather than powers of 10. They are universal, and fundamental, to all programming languages.

- **Boolean / Proposition (A)**: a proposition, or boolean, is a value which can be either true or false. They are universal, and fundamental, to all programming languages, as well as philosophy.

- **String (A)**: a string is a sequence of characters (alphanumeric, punctuation, spaces, emojis, etc). It is a value which can be used to represent human language inside a computer. They are universal, and fundamental, to all programming languages.

- **Code (A)**: code is text written according to the rules of a programming language.

- **Syntax (A)**: the *grammatical structure* of a language. For example, English and French are often simply described as languages where the order of words is "Subject-Verb-Object" (with some eventual Complement, indicating things like location or temporality, that can float around). Another example, the syntax of a programming language can restrict the programmer to declare the type before a variable (e.g., `Type_A variable_a`) or after the variable (e.g., `variable_a : Type_A`). An instruction ("sentence" of a programming language) that does not respect the syntax of its language will not be executable by the machine, because it is not translatable to machine language.

- **Semantics (A)**: the *meaning* taken by an instruction/phrase in a language. An example of a semantic error would be `Integer my_variable = "bobo"`: "bobo" is not an integer, so the statement, while syntactically correct (in a language like C), is semantically incorrect. Here's a famous example from "real-world" language by Noam Chomsky: "Colorless green ideas sleep furiously". This is not a grammatical error, it's a valid sentence, syntax-wise; however, there's clearly an error with the semantics: this specific association of words is meaningless.

- **Predicate (A)**: a predicate is a function that returns a boolean. These are very important in computer science, logic, and philosophy. And example would be "X exists", with X as input. "Water exists" would return true, but "Santa Claus exists" would return false.

- **Variable (computer science) (AMELT)**: in computer science, a variable is a "word" declared through writing, used to store a mathematical value in memory, and then use this mathematical value conceptually. For example, (`Float speed = 5.75` would declare a variable for "speed". In the rest of the code, the programmer would then use "speed" in the code (repeatedly and in the adapted way). This makes code much clearer to read than a "5.75" that hangs around, and is meaningless with a deep understanding of the context.

- **Function (computer science) (A)**: in computer science, function is a series of instructions callable from other places in the code, which can take zero, one, or more inputs and return zero, one, or more outputs. Functions can also affect the state (memory) of a program or computer (side effects). This is for example the case for functions that display information on a screen: they affect the electronics of the computer. A "pure" function is one which does not have side effect: one which is a purely mathematical calculation that can be figured out with pen and paper, given some inputs.

- **Condition (AMELT)**: A condition is a statement that translates a logical calculation (Boolean, a question with a true/false answer) into a conditional redirection (a branching) of the code. Keywords: `if, elif, else, and, or, not, then`.

- **Loop (AMELT)**: A loop is a series of instructions that can be repeated, with minimal configurable changes, as long as a given condition remains true.

- **Turing-completeness (BMELTG)**: roughly speaking, a programming language is said to be "Turing-complete" if it is "as expressive as possible". That means that it is able to execute computations on all types, store information in its memory, execute its computations contextually/conditionally, and redirect its reading head. Turing-completeness is a fundamental notion of theoretical computer science, and for reasons that are long and

complex to explain, it is in a way the ultimate "speed limit" to computation. A Turing-complete language is able to execute any (reasonable) function. Almost all programming languages have a Turing-complete level of expressivity.

- **Argument (AMELT)**: an argument is a synonym for a function input.
- **Return (AMELT)**: a return (value) is a synonym for a function output.
- **Signature (AMELT)**: the signature of a function is the declaration of the name of the function, and the types and names of its inputs and outputs. The function's type is defined as "types of inputs -¿ types of outputs", and is a direct consequence of these.
- **Interpreted language / Compiled language (AMELT)**: a language is said to be "interpreted" if a software called "the interpreter" must be launched to read the code line by line, as the code runs, so that it executes. A language is said to be "compiled" if a software called the "compiler" must read the whole code and produce an executable ahead of time, before the program can be executed. Python is an example of an interpreted language. C is a compiled language.
- **Static typing / Dynamic typing (AMELT)**: A "typing" is how the type system for a programming language was designed. It is said to be static if it is necessary at the level of the programming language's syntax, or if it can be inferred before the code is run. A typing is said to be dynamic if the interpreter or software is only able to infer the type of a computer value from its context, at runtime (and thus the declaration of the type of a variable or the arguments of a function is not necessary). Python is a dynamically typed language. C is a statically typed language. Statically typed languages are generally more restrictive in their syntax, but also generally more rigorous and reliable.
- **Paradigm (of a programming language) (BMELTO)**: way to design a programming language. The 3 most frequently used paradigms are: imperative, object-oriented, and functional. You may also see the term "procedural" used instead of "imperative". These paradigms are not necessarily mutually exclusive. Imperative languages are very close to how the machine works, giving orders as concrete instructions (typical example: C). Object-oriented languages structure their components in "classes", types that contain both data (i.e., state: nouns or adjectives) and functions (i.e., actions: verbs). Most modern languages are inspired by object-oriented design; Python included, among others. Functional languages are very close to mathematics and are inspired by the lambda calculus; they have the advantage of producing very solid code because they are close to a mathematical proof, when they are handled by the interpreter or the compiler. It's helpful to think of the history of

programming, with one side (functional) starting from the works of mathematicians, and the other starting from the works of electronics engineers (imperative). There are other paradigms (such as array-oriented), though their use is much less frequent.

- **Version control / Git / GitHub / GitLab / Bitbucket (AMELTO)**: Technology for managing project data and archiving code, used in software development. There are many historical ones, but today, git has become the *de facto* norm. A git "repository" contains all the archives and changes in a project's data since its creation. This means that you can use it to find anything that was saved at any time in a project's history (up to voluntary deletion of that history). Platforms like GitHub, GitLab and BitBucket allow sharing of, and collaboration for, code projects online. Using git daily is a ***fundamental*** practice when programming, *and* when collaborating with programmers.

Computer Science: Hardware

This section describes the important electronic components that allow an electronic computer to work.

- **Computer (A)**: There are two major definitions of computer: one is a mathematical model for "how to automatically do math" (see Turing-completeness); another is a physical (generally electronic) machine, serving as a platform for general computation (see Von Neumann architecture). The more common definition is this second one.
- **Central Processing Unit, CPU (AEL, BMTG)**: the fundamental calculation unit of a computer.
- **Graphics Processing Unit, GPU (AEL, BMTG)**: a computing unit specialized in parallelized computing. Modern GPUs are efficient in performing tasks like simultaneous scalar products, making them suitable for tasks involving linear algebra.
- **Register (AE, BMLTG)**: a memory unit that contains data directly accessible to the CPU, and over which it can run its computations. It is the fastest and smallest kind of memory generally available on a computer.
- **Random Access Memory, RAM (AE, BMLTG)**: a memory unit that contains data and software currently running on the computer. It is cleared as soon as the computer turns off. It is generally produced in the form of flat stick of complex electronics. It is a medium sized, medium access speed memory component.
- **Read-Only Memory, ROM (AE, BMLTG)**: a memory unit that contains inactive but saved data and software of the machine. It is generally in the form of a rotating hard disk or a flash memory drive. It is a big sized but comparatively slow accessed memory component.

- **Motherboard (AE, BMLTG)**: The control unit that contains most of the software necessary for the computer to start. It serves as an interface to various hardware components.
- **Power supply unit, PSU (AE, BMLTG)**: An electronic unit that provides and adapts power from an electric sector outlet to the computer, ensuring its proper operation.
- **Port (computer electronics) (AE, BMLTG)**: A female-type (hole-like) plug on a computer that the motherboard can use to interface with other electronic components, such as external memory drives, sound systems, or network cables.
- **Card (sound card, network card, etc) (AE, BMLTG)**: A specific electronic component meant to handle some specific function which generally is not simple computation or handling of memory. This includes managing software and devices used to connect to computer networks, or producing sounds from binary data and programs. For "graphics card", see GPU.

Computer Science: Data Types

This section describes various data types that are fundamental to algorithmics in general, and data engineering/science in particular.

- **Array / List (AMELTG)**: an array, or list, is a series of values, usually of the same type, stored (usually) contiguously in memory, and accessible by index. In the vast majority of languages, the indexing of an array starts at 0. For example, if you have the following array of integers `Array<Integer> my_array = [1, 4, 6, 10, 0]`, then `array[2]` returns 6. In languages where a distinction is made between arrays and lists, arrays are contiguous in memory, and lists are implemented as **linked lists**, which consists of a series nodes at various places in memory, each node containing both a value, and a "pointer", indicating where the next vaue can be found in memory. "Doubly" linked-list have each "next" node in the list also point to the previous from which it was referred. "Cyclic" lists have the last node point to the first.
- **Tensor (computer science) (AML, BETG)**: a tensor is an array of arrays of arrays of arrays of arrays... of elements usually of the same type. The "rank" of a tensor is its nesting level. A simple value is a tensor of rank 0. A simple array is a tensor of rank 1. An array of arrays is a rank 2 tensor. An array of arrays of arrays is a rank 3 tensor. Etc. An Excel spreadsheet with only number is an example of a rank 2 tensor.
- **Graph (A)**: a graph is a set of points (usually of the same type, called nodes), which are linked together 2-by-2 (by connections called edges). A very rudimentary social network, for example, can model its users as "one

node per person" and "one edge between two people if they are friends". Graphs can be **oriented**, if the links are directional (such as the "parenthood" link between a parent and their child), or **non-oriented**, if the links always go both ways (such as "being related by blood"). If a graph has numerical values (on its nodes or edges) it said to be "**weighted**". If it has qualifiers (on its nodes or edges) it is said to be "**labeled**". The **distance** between two nodes of a graph is the length (in amount of edges, or sum of edge weights) of the smallest path between these two nodes. The "**diameter**" of a graph is the maximal distance between any pair of nodes in the graph. An edge from a node to itself is called a "**loop**".

- **Tree (AMELTG)**: a tree is a graph without cycles. One of the nodes of a tree can singled out to be the source of some operation on the tree, or some structure, in which case it is called the tree's **root**, in this case, we tend to speak of "parent" nodes and "child" nodes, where the root is the universal common ancestor (even if the tree is non-oriented). A node that has no children is called a **leaf**. The **depth** of a tree is the distance between the root and its furthest leaf. There are special kinds of trees, such as binary trees (where every node has at most 2 children), binary search trees (BST, weighted binary trees where each left child contains only values inferior to the parent, and each right child only values that are superior), B-trees (which generalize BSTs to n possible children), red-black trees (a type of BST where one can ensure a certain amount of balance between branches, i.e, minimizing tree depth).

- **Stack / Last-in first-out, LIFO (A)**: a stack is a fundamental data structure, consisting of blocks of the same type, and just 3 operations: push (adding an element to the top of the stack), view (looking at the value of the element at the top of the stack), and pop (removing the element from the top of the stack). At its most primitive, mathematical level, a program's memory is generally modeled as a stack. This concept also has applications in logistics, when it comes to strategies for handling inventory.

- **Queue / First-in first-out, FIFO (A)**: a queue is a fundamental data structure, consisting of blocks of the same type, and just 3 operations: enqueue (adding an element to the back of the queue), view (looking at the value of the element at the front of the queue), and dequeue (removing the element from the front of the queue). This concept also has applications in logistics, when it comes to strategies for handling inventory.

- **Data point cloud / Data frame / Dataframe / Data point space / Data table / Relational table (A)**: a data frame is two-dimensional table, with "individuals" (aka "entities") as rows, and "attributes" as columns. When its data is visualized geometrically, it corresponds to a point cloud in a mathematical coordinate frame, where each column of the table defines a "dimension", i.e. an axis of the coordinate frame. (Note

that for non-numeric columns, these can generally require 2 axes, one for the various labels/categories, and 1 for some metric, generally a count of values.) Each entity corresponds precisely to a single point in this frame. Most data tables have too many dimensions for the human brain to visualize (since we are limited to 3 spatial dimensions, 1 temporal dimension, and possibly 1 or 2 color gradients, for a total of 6, which is in practice very rarely attainable in an understandable way). Mathematically, if one considers each attribute as a given set (or type), then a dataframe can be understood as a subset of the repeated cartesian product of the attributes.

- **Simplex (BML, CET)**: (pl. simplices) an n-simplex is the n-dimensional equivalent of the 1-dimensional line, 2-dimensional triangle, 3-dimensional tetrahedron, etc.
- **Simplicial complex (BML, CET)**: a simplicial complex is a collection of connected simplices. Triangle meshes from computer graphics (video games and CGI) are a kind of simplicial complex.
- **Hypergraph (BML, CET)**: a hypergraph is a graph where edges can be n-ary, rather than just 2-ary. More generally, a hypergraph is any subset of the power set of a set of nodes. (Non-oriented) graphs and simplicial complexes are specific kinds of hypergraphs.

Computer Science: Computer engineering

This section has some vocabulary which useful for computer engineering, on the software side. This can also be understood as a section on low-level software engineering. Note that there is also another section on systems engineering, and yet another network engineering, specifically.

- **Operating System, OS (A)**: an operating system is a piece of software that allows the management and operation of other software. It is generally the first piece of software that you want to run when turning on a computer. Examples of such architectures include Mac OS, Windows, Linux-Debian, Linux-RedHat, iOS, Android, Raspberry Pi, Microsoft Azure, and more.
- **Bootloader (BE, CG)**: For most modern computer architectures, since modern OSes are quite complex to launch, a smaller operating system, called a bootloader, is used to launch the main operating system (in fact, this can be a multi-stage process where a bootloader loads another, more complex bootloader). Examples of first-stage boot-loaders include BIOS and UEFI. Examples of second stage bootloaders include GNU GRUB and rEFInd. In more complex (but important) scenarios, the bootloader and OS's data can be received from some other computer in the network at startup.
- **Cross-platform (AE, BMLTO)**: Refers to software or code that can run on various operating systems.

- **Process (computer engineering) (A)**: Refers to an isolated, identified, and managed piece of software that is running on an operating system. Whereas a program is the "specification" of instructions that a computer should run, a process is the form that a program when it is "run" (executed) by a computer. A program can instantiate of multiple processes (a process called forking).
- **Kernel (computer engineering) (AE, BG)**: the kernel of an operating system is the fundamental "manager" program of the computer, which is tasked to control how other programs run. Its tasks include:
  - creating, registering, monitoring, managing and killing processes;
  - scheduling and executing tasks required by processes;
  - allocating (or refusing to allocate) memory and compute ressources to processes when they ask for them;
  - handling user management, and the subsequent permissions that various processes are granted;
  - handling fundamental errors (interrupts);
  - handling the filesystem (this includes files, but also peripheral devices, network connections, etc.);
  - etc.
- **Bit (A)**: a bit is the fundamental unit of information. It corresponds to a choice 2 values: 0 or 1 (equivalently, "true or false", or yet again, "passing current or blocked current"). A sequence of bits (such as 1000111011) is called a "word in binary language".
- **Byte (A)**: a byte is an ordered sequence of 8 bits, corresponding to a choice between $2^8 = 256$ values. It is the fundamental unit that modern computers use to divide information into organized chunks.
- **Address / Memory address (AE, BMLT)**: a memory address is a specific location on a computer, described as a numeric integer value, which contains some data, usually the size of a byte.
- **Pointer (AE, BMLT)**: a pointer is value of a specific type in imperative programming, one which expresses a memory address in the computer as an integer value. Pointers are a fundamental construct in imperative programming.
- **Memory buffer (AE, BMLT)**: a memory buffer is a fixed-sized (n bytes) piece of computer memory, allocated for some process by the kernel, in which binary data can reside. The address of a memory buffer is generally taken to be that of the first element in the buffer.

- **Cache / Memory cache (BE, CMLT)**: a memory cache is a memory device (or area in a memory device) reserved to store values that were either recently used, frequently used, or both (depending on the strategy). Caches are one of the major ways to engineer speedups in memory retrieval, but they do have some cost. Some types of caches exist on a single machine, while others can be used to speed up data retrieval from a network of machines.

- **Thread (BE, CMLT)**: a thread is a specific resource that can be allocated to allow multiple parts of a program (or the same part, but multiple times) to run simultaneously. Threads are the fundamental concept of concurrent and parallel programming. Ending a thread (and eventually returning to sequential programming) is called "joining" the thread. All threads generated by a process are dependent said "parent" process (if the process is killed by the OS, .

- **Readhead (BMELT)**: a readhead is a location at which the code is currently being read. Sequential (synchronous and single-threaded) programs have a single readhead at any time. Concurrent (multi-threaded and/or asynchronous) programs typically have multiple readheads.

- **Task (BE, CMLT)**: a task is an ambiguous term describing a unit of work in a computation. This catch-all term is used when the actual implementation of a concurrent program is unknown and we need to think about it abstractly. To give an idea, a task might be a sequence of instructions that a thread repeats regularly, or the unit of work done by a single thread over its lifetime, or the unit of work done by a full process. We then use tasks to describe architectures where the question of "how do we execute multiple tasks ?" is key. In certain programming languages (like Ada or Erlang), "tasks" are a much more well-defined concept, used explicitly in the respective language's concurrency model.

- **Concurrency / Concurrent programming (BE, CMLT)**: Concurrent programming refers to the act of writing a program so that its process (or processes) can read various parts of the program (or the same part, multiple times) simultaneously. Concurrent programming has 2 main goals: efficiently switching between tasks to make the most of available resources (such as having a music player and a browser run simultaneously on an OS, or calculating various aspect of a video game's graphics at the same time), or making sure that processes that *need* to run at the same time *can* run at the same time (such as being able to interact with a program's window and having the program communicate with the network in the background). There are two main types of tools for concurrency: multithreading and asynchrony. Concurrent programs are tough to write, and can lead to very harmful, difficult to resolve bugs, namely deadlocks, and race conditions. All modern CPUs allow for concurrent programming.

- **Parallelism / Parallel programming (BMELT)**: parallelism refers to a specific kind of concurrency, when (roughly) the same task needs to be run on multiple threads, or processes, or machines, at the same time. The most famous parallel compute device is the GPU. We can distinguish between shared-memory parallelism (on a single machine) and distributed parallelism (over multiple machines, and memory sharing is done by network communication).

- **Single instruction single data, SISD (BEL, CMT)**: refers to sequential programming. In a SISD architecture, there is only one thread at any time, and it reads the code sequentially.

- **Single instruction multiple data, SIMD (BEL, CMT)**: refers to parallel concurrency. In a SIMD architecture, multiple instances of data of the same type, but with differing values, are processed by multiple threads, executing the same program simultaneously, instruction-by-instruction (hence the "single instruction" principle). An example is as an array of 3D vectors, where each 3D vector in the array is to be processed by the same algorithm. The fundamental problem of SIMD programming are conditional branches ("if/else" statements) which may cause slowdowns, because these break the single-instruction principle. SIMD is the way that GPUs do their calculations.

- **Multiple instruction multiple data, MIMD (BEL, CMT)**: refers to general concurrency. Any instruction sequence can be applied concurrently to any data. MIMD is the way that modern (multi-core) CPUs do their computations. Any MIMD device can run some version of a SIMD program (to some extent), but they might not be as specialized and effective as a device designed for SIMD (such as a GPU).

- **Multithreading (BEL, CMT)**: multithreading a program is the act of having the program create multiple threads in order to run concurrently. If a program is a restaurant's kitchen, multithreading is like hiring multiple cooks, so that each can handle a given task. One can write a program that is both multithreaded and asynchronous, since these are 2 different concepts allowing concurrency.

- **Asynchrony (BE, CMLT)**: asynchrony is a way that a program can run concurrently on a single thread, by having multiple readheads, and switching between these. If a program is a restaurant's kitchen, asynchrony is like a single cook preparing a sauce, vegetables, meat, etc, simultaneously in their own respective pot, by doing each step that should be done as soon as it can be done, but never being idle, and moving from pot to pot regularly. For example, asynchrony can be implemented via things like an event loop (which manages the different readheads and their execution order on a single thread), coroutines (which allow functions to suspend themselves and let their parent start executing again), or message passing

between readheads (where a readhead starts up only when it receives a specific message from another readhead). Typical constructs through which asynchrony can be used in programming languages include: callback functions (a function that is passed as an argument, and is to be executed once the fed function returns), (monadic) promises or futures (which are type wrappers (functors) which allow the programming language to know that their content is running asynchronously), or async/await keywords (used to specify whether a function is asynchronous, and whether a certain call of this function should be blocking). Asynchrony is a very effective mechanism to handle operations that require lots of IO (typically, handling multiple simultaneous network connections; or reading/writing from/to multiple files) while using only minimal resources (e.g., a single thread). One can write a program that is both multithreaded and asynchronous, since these are 2 different concepts allowing concurrency.

- **Mutex / Lock / Semaphore (BE, CMLT)**: a mutex (for "mutual exclusion") is a tool that multithreaded or asynchronous programs use to avoid race conditions and deadlocks. A mutex lock defines a section of the code that should only be accessed by a single readhead at any time, sequentially; a mutex unlock defines the moment where the code can be read concurrently again. There are multiple possible implementations for mutexes, but the following gives a good idea of the expected behavior. A mutex is basically a global counter that is incremented each time the "lock" operation is called, and decremented at each "unlock". When a readhead arrives at a mutex lock, if the counter is at 0, it increments it to 1, and keeps reading what follows. If another readhead arrives at the lock, it increments it to 2, and gets put into a queue. Once the first readhead reaches the unlock, the global counter is decremented to 1, and the first readhead in the queue can now get its turn to execute. Typically, mutexes are used when trying to get a resource from a resource pool that is shared between threads and/or asynchronous readheads (such as available connections for a server).

- **Deadlock (BE, CMLT)**: a deadlock happens when multiple threads or readheads are trying to access the same resource(s) and some readhead is refusing to release resources that are necessary for the program to continue. A typical example is the dining philosophers problem, which we will caricature and generalize here. We have two parties (readheads), A and B, that each needs 2 resources (R1 and R2) to do their respective task. Neither will release a resource if they have it, until their task is complete. The program is running concurrently, so both A and B can act simultaneously. A picks up R1. B picks up R2. Neither A nor B can now pick up both resources, and neither will release their own resource. The program is stuck, it has reached a deadlock.

- **Race condition (BE, CMLT)**: a race condition happens when a resource is accessed simultaneously by multiple readheads and this leads to

logical inconsistencies (which can lead to crashes in the worst case). Say two parties (readheads), A and B, share a pair memory registers to run an addition. A writes 5 to the first register. B writes 1 to the first register. A writes 7 to the second register. A runs the addition and obtains 8, rather than the expected 12. Generally, race conditions are non-deterministic bugs, since the overall result depends on the order that concurrent readheads accessed the resource, which varies upon multiple executions of the same program. For example, here, if B had had the time to write 4 to the second register before A ran its addition, A would have obtained 5 instead.

- **Vectorization (BMELT)**: Vectorization is the expression of computations on arrays (*a fortiori*, tensors) in such a way that the computations can take place simultaneously (to leverage parallelism). This works according to the SIMD (Single Instruction, Multiple Data) principle: you run exactly the same operations on lots of data of the same type, at the same time. It is this principle that allows GPUs to do graphical computations or computation for data or neural networks faster than CPUs.
- **Clock (BE, CMLT)**: a logical electronic circuit's (*a fortiori*, a computer's) clock is a device that sends regular, evenly timed, electronic signals, like a beat or metronome. This allow operations to be synchronized, which is essential to avoid things like deadlocks and race conditions that could happen through the raw, physical irregularities of electronic components. To increase the manufacturer's default clock rate on a computer's CPU, in order to improve performance (at the potential cost of excess energy consumption, heat generation, and component failure or damage), is called "overclocking".

Computer Science: Network engineering

This section has some vocabulary which useful for network engineering. Note that there is also another section on computer engineering, and yet another systems engineering specifically.

- **Server (A)**: a server is a piece of software that allows other computers to interact with it. You can think of a "server" as a program "at whose door you can knock".
- **Client (A)**: a client is a piece of software that can make requests to another computer. Web browsers are the typical "client" software that most people know about. You can think of a client as a program "that goes to knock on other people's door".
- **Peer (AG)**: a peer is a piece of software that is both a client and a server.
- **Gateway / Gateway machine / Gateway server (AG)**: the first machine that is reached to or from an internet, and serves a "gateway" (door) into an intranet (isolated computer network).

- **ISO model (of network layers) (AE, CG)**: a description of the various layers required for modern computer networks to functions, down from electronics, up to applicative logic (TODO: go into more detail).
- **Data packet (AE, BG)**: a memory buffer containing specific data, which is given a "header" (small chunk of data to provide addressing, verification, or other information) in order to be replicated from machine to machine and reach a specific destination within a computer network.
- **Intranet (A)**: an intranet is a small to medium network of computers, all put into a common basis for communication via some software protocols (most important of which are IP, TCP, and UDP). Generally, an intranet is either isolated from the Internet completely, or has specific machines, called gateways, which manages Internet access to and from the intranet.
- **Internet (A)**: the internet is a extremely large network of computers, all put into a common basis for communication via some software protocols (most important of which are IP, TCP, and UDP).
- **Internet Protocol, IP (AG)**: the internet protocol is a collection of software and protocols allowing the transmission between two machines that are not directly linked, but need to go through an intermediary network to talk to each other. It is the fundamental building block of the modern internet.
- **Transmission Control Protocol, TCP (AG)**: a network protocol allowing the streamed transmission of data packets from computer to computer. Unlike UDP, TCP checks if the packet sent was well-received (complete, uncorrupted) via supplementary communication from the receiver to the sender. Most network communications (except video streaming) are sent via TCP, and brought to destination via IP, therefore, we often refer to their combination as **TCP/IP**, which is the basis for the modern internet.
- **User Datagram Protocol, UDP (AG)**: a network protocol allowing the streamed transmission of data packets from computer to computer. Unlike TCP, UDP does not check if the package was well received. This allows software using UDP to keep sending data, regardless of its arrival status, thereby greatly improving communication performance and scalability. It is in particular the protocol used for video streaming.
- **HyperText Transfer Protocol, HTTP (A)**: a protocol used as a way of requesting or sending data so that it can be used by software called a "browser" to visualize or run content, (almost always) by receiving it from some distant computer (a server). HTTP runs over TCP/IP, providing it a form of semantics.
- **Socket (AE, CG)**: a socket is a software-level numerical identifier for a connection within a program. It allows programmers to establish distinct

connection to other machine via IP within their software. Sockets work by requesting specific resources from the operating system.

- **Port (AE, BG)**: a port is an OS-level numerical identifier that allows one to distinguish between different types of protocols when communicating. The only ports that are (legally) open by default on every OS are port 80, for HTTP, and 443, for HTTPS. Of note is port 22, for the SSH protocol.

- **Firewall (AE, BG)**: a firewall is a specific piece of software that manages which ports are open and which are closed (both for outgoing and incoming connections), in order to protect a given OS from foreign intrusion.

- **Cryptography (A)**: from the greek words for "hidden" and "writing/drawing", cryptography is the science of being able to communicate messages in a way that their information can only be read by the intended receiver. Cryptography dates back to early Antiquity and has a truly fascinating history, touching frequently on political, military, and scientific history.

- **Public-private-key cryptography, PPK cryptography / Asymmetric cryptography (A)**: describes the fundamental way that computers communicate cryptographically. The idea is to use the solution of a mathematical problem as a key to make an encrypted (illegible) message legible. If you can solve the problem, you can use its solution to get access to the message, but the problem would take even the most powerful computers unreasonable amounts of time to solve. On the other hand, verifying that the solution is correct is very easy and quick to do (just like solving a Sudoku vs verifying that a filled-out sudoku is valid). Admittedly the idea of a "public key" is a bit of a misnomer. Think more of a "public key" as an "open treasure chest that locks down as soon as you close it, that you can duplicate an infinite amount of times". It is what allows one to create a "complex mathematical problem" from their message, in a way that is hard to rewind, unless you know the solution for all messages. Party A (referred to as Alice) wants to send a secure message to party B (Bob). Alice asks Bob for his public key (treasure chest). Alice writes her message and encrypts it with Bob's public key (she puts it in her copy of Bob's treasure chest). Now, even Alice can't open the treasure chest ! She sends the encrypted message (treasure containing the message) back to Bob. Bob has the "private key", the solution to the problem (the key to the treasure chest that must never be shared). He can use it to decrypt the message (open the chest) and read Alice's message.

- **Peer-to-peer network (A)**: a peer-to-peer network is a network in which every node is a peer. Peer-to-peer networks are powerful because they are highly distributed, making for efficient transmission of data, or a good source of decentralized (but federated) compute.

- **Secure shell, SSH (A)**: a communication protocol used to securely access a user's terminal on a distant machine. Uses port 22 by default, but

a firewall might block port 22 and reserve another (hidden) port for SSH instead, in order to make a machine more secure by obfuscating the entry port.

Computer Science: Systems engineering

This section has some vocabulary which useful for systems engineering. Note that there is also another section on computer engineering, and yet another network engineering specifically.

- **Relational database / SQL database (AMELTO)**: a database is a set of (usually massive) data tables, possibly logically linked together by 2-column tables called "junction tables". These have been the most important data structure for a long time. Relational databases have an advanced mathematical formalism, called "**relational algebra**", developed by a certain Edgar Codd. There are programming language specific to interactions with databases, including the SQL family of languages. Databases are so frequently relational that anything else is generally referred to under the umbrella term "NoSQL".
- **Normalization / Relational database normalization (AE, BMLTO)**: the relational algebra specifies what are called "**normal forms**" for a relational table. These allow the prevention of a certain amount of database problems generally referred to as "**anomalies**".

  Here are the standard anomalies which one needs to look out for. Our example uses a fictitious, non-normalized database, where data about a player in an MMORPG, and their inventory, are stored in the same table, with roughly one row per inventory item.

  - deletion anomaly: deleting a row (such as a player consuming their last item) deletes extra unrelated info (such as the last reference to the player itself);
  - update anomaly: updating a row (out of two that should be updated, such as the price of only one of two items) leaves an incoherence (typical when there is duplicate info);
  - insertion anomaly: adding known information (such as "a new player should begin the game with a normal status") requires adding a row, which can't be added because other info is lacking (they start with an empty inventory, so we can't neatly add the row).

  The following describes how the requirements a table must meet to be in a given rank of normal form.

  - 1NF requirements: presence of a primary key (a column, or group of column, from which one can uniquely distinguish each entity), unordered rows (the order of rows should not provide any semantic value; the list of entities is actually not a list, but a mathematical set),

simply-typed columns (the content of column is consistently typed, no mix of string and int, etc.), no repeating groups (aka atomization: no cell should contain a variable amount of information; no list in a cell).

- 2NF requirements: the table is 1NF, and when a primary key in the table is composed of 2 fields, each non-key attribute (column) must depend on (=can only be infered through) the entire primary key.
- 3NF requirements: the table is 2NF, and no attribute depends on another transitively.
- BCNF, 3.5NF (Boyce-Codd NF): "every attribute in all tables should depend on their whole primary key, and only on their primary key".
- 4NF requirements: the table is 3NF, and the only maltivalued dependencies allowed are multivalued dependencies on the key.
- 5NF requirements: the table is 4NF, and cannot be described as the result of a junction of other tables.

There are some case where one could want to denormalize a table, for example for performance reasons in a database which is read-only.

- **Index / (Relational database) index (AMELT)**: an index is a data structure, generally some form of B-tree (possibly a BST or red-black tree), where the leaves of the tree are additionally structured as a doubly-linked list. These allow the retrieval of information in logarithmic time complexity (rather than linear), which, given the size of most data tables, immensely improves retrieval performance. Since creating an index comes with a cost (replicating one full column in memory, and needing to update the index every time the corresponding column is updated), we generally create an index for one of two reasons. Either we create an index for a column which appears in many read queries and is rarely updated (such as a Name column), or we create an index tailored for a specific query which is very frequently used and is always the same (and thus deserves a particular, custom-made speedup).
- **NoSQL database (AMELTO)**: a NoSQL database is an umbrella-term for any database that isn't SQL. Some of them are even SQL-like, but with extra structure. Here is a list of frequently seen NoSQL database architectures.
  - graph database: useful to model "ecosystems" or networks, any type of domain where there are analogous element between which there are many relations. Generally comes with a query language, like Oracle's property graph query language (PGQL). A famous example is neo4j.
  - key-value store: a way of organizing data as key-value pairs (typically, hashmaps). Keys are generally strings, values are generally primary types, arrays, or dictionaries. There is no specific interaction

language like SQL for them; so managing their content is generally handled in a custom manner at the application level.

  - blob (binary large object) store: used to stores lots of binary data files (videos, music, large text, exe, etc). Famous examples include Google cloud storage, Simple Storage Service (Amazon S3), or Azure blob store.
  - time series database: stores a bunch of data which consists of metrics that happen at regular, sequential timed events. Famous examples include influxDB and prometheus.
  - document database: stores data as documents containing JSON or BSON data (structured, but with variable schemas) that are spread across machines. JSON and BSON can be described as "key-value trees" or "nested key-value pairs". Collections of documents can be indexed, nested or linked. A famous example is MongoDB.
  - full-text database: works like the index at the end of a book (finds a term via a hashmap of terms-to-locations), and is used for building performant search engines. Examples: solar, elasticsearch, lucene, algolia, meilisearch.
  - columnar TODO
  - wide-column / column-family store: like a key-value store, but with a bit more depth. The values can generally be understood as denormalized relational tables. These are good for high-write low-read (eg, lots of time series data, or event logs), and specific filtered per-column reads (e.g., Apache Cassandra).
  - multimodel: is said of systems that will try to cleverly design what's required as a data model for you, based on your specification (such as fauna with GraphQL).

- **Structured Query Language, SQL (AMELTO)**: SQL is a programming language which is not general purpose, but instead specifically tailored to interacting with a relational database. There are multiple variations on the SQL language for various other database technologies (such as MySQL or PostgreSQL).

  The various types of SQL queries are often classified under the acronym "**CRUD**".

  - Create (CREATE, INSERT): allows things like creating new tables, indices, or adding rows to existing tables;
  - Read (SELECT): allows to specifically select, and/or combine, and/or format, and/or filter, and/or sort, rows from one or more tables.
  - Update (UPDATE): allows to find and edit one or multiple rows' attribute(s).
  - Delete (DROP, DELETE): delete a row from a table, or delete a full table.

However, some SQL operations are not of this kind, but still useful, such as the EXPLAIN/ANALYZE family of commands.

When it comes to custom queries, these tend to mostly be Read type queries. For this reason, it is important to understand the general order of execution of the various components of an SQL read query. This is necessary, since you can get to the same result in various ways, and some ways are much more costly than others.

  - SELECT: choose columns, then (optional) apply a map/function to its values, and put function's result into a kind of "variable", which acts as a new temporary column;
  - FROM: choose the source table;
  - JOIN: choose second table source and define the kind of join (from least costly to most costly: INNER, LEFT or RIGHT, CROSS);
  - ON: express which columns to match in FROM / JOIN;
  - WHERE: predicate to filter based on column value for the chosen column(s);
  - GROUP BY: fold on specific column;
  - HAVING: also a filter, but unlike WHERE, works for *grouped* records;
  - ORDER BY: sort by a specific column;
  - LIMIT / TOP: cuts to keep only the first n results.

- **Transaction / Database transaction (AE, BMLTO)**: a transaction is a logical unit of work when writing to, or reading from, a database. A transaction can be composed of multiple sub-operations.

- **ACID (BMELTO)**: this is an acronym which describes the key properties that must be respected for a database transaction to work properly and be properly designed. Most modern database technologies are said to be ACID-compliant because they adhere to these principles.

  - **atomicity**: either the whole transaction is executed, or it fails fully (e.g.: no adding money to someone without removing money from someone else). If the operation succeeds, we commit the result (put it into effect). If some operation fails, we rollback to the state before the transaction started.
  - **consistency**: transactions must respect logical constraints (specified by the domain) on the data (e.g.: no one can spend more money than they are stated to own in the database).
  - **isolation**: in effect, transactions are applied sequentially (e.g.: if two people try to concurrently retrieve money from a shared account, this principle prevents them from withdrawing a sum of money greater than the account owns overall).

  - **durability**: valid transactions are perennial and survive a system failure. If a system or machine fails mid-transaction, the correct state can be recovered. This is ensured by some things like *write-ahead logging*, where before any transaction or commit, we write to a file the transaction that is to be committed. If the transaction is committed successfully, that log file is cleared of the now validated transaction.

- **Distribution / Distributed computation / Distributed system (AMELTO)**: refers to the act of having a program run as processes over multiple machines in a network, such a network being called a "distributed system".

- **CAP theorem (BE, CMLT)**: the CAP theorem (standing for Consistency/Availability/Partition tolerance theorem) is a theorem which states that out of keeping data over a system globally consistent, available, or globally connected (without a partition of the overall network graph into 2 pieces or more), only two of those three can be ensured at any time. In practice, since faults are unavoidable, this boils down, when a partition happens, to either choosing consistency (preventing transactions because they can't be ensured to be consistent, since part of the network is unreachable) or availability (allowing transactions to happen, knowing that they'll need put the system in an inconsistent state which will need to be resolved once the network is fixed and the partition disappear).

- **Sharding / Vertical sharding / Horizontal sharding (BE, CMLT)**: sharding refers to the division of a large relational table into smaller tables. This is done to divide the table across multiple machines, whether to lighten storage, or to improve performance.

  Horizontal sharding refers to dividing the table by rows, generally based on some index (such as sections based on a Name column, and sorted in alphabetical order: people with a name A to C on one machine, people with names starting from D to F, etc). For each horizontal shard, the row corresponding to the column names is (of course) duplicated.

  Vertical sharding refers to dividing across columns. For example, if a "person" entity has dozens or hundreds of attributes, we might divide these according to themes that should be analyzed in common (say we have a table with 81 columns, and we divide it as 1 primary key attribute, 10 attributes for professional situation, 20 attributes for medical history, 30 attributes for personal interests, and 20 attributes for consumer behavior; we could then vertically shard into a total of 4 new tables, stored on 4 different machines). The primary key of the original table is duplicated across all vertical shards. A join can then be used to obtain the rows of the original table, with all their original attributes.

- **Scaling / Vertical scaling / Horizontal scaling (AE, BG)**: Scaling, in the language of systems engineering, refers to increasing the perfor-

mance and capabilities (in memory, compute, and/or networking) of a computer system. Vertical scaling refers to increasing the overall specs (components) of a given machine, or of each machine in the system. Horizontal scaling refers to adding more machines to the system, or improving the system self-management software, so that more machines can contribute to the system's overall performance.

- **Blade (AG)**: a blade is a thin and powerful computer (in terms of memory, compute and network bandwidth) that is used as a unit computer in a some forms of supercomputers, in a datacenter or in a server farm.
- **Rack (AG)**: a rack is like a closet that holds, and powers, lots of blades. They are used in server farms, supercomputers and datacenters.
- **Supercomputer (AG)**: a supercomputer is an industrial-grade computer, which much, much higher capabilities than a regular PC. It is generally composed of multiple computers, or a single computer with a lot of memory and compute units working in tandem.
- **Server farm (AG)**: a server farm is a place (often a building like a warehouse) containing hundreds of blades, which are mainly used as servers, to provide web-based services.
- **Datacenter (AG)**: a datacenter is a server farm used specifically for the purpose of servicing, or running compute, over massive amounts of data.
- **Fault tolerance (AE, BMLTO)**: "fault tolerance" describes a system's capacity to keep working as intended (servicing data, routing requests, staying consistent, etc) even in case of machine failures (where unit computers become partly, or fully, unavailable).

Computer Science: Python programming for Data Science and Machine Learning

This section presents the various tools that programmers who work in data science and machine learning use (most of the time). We explicitly ignore the R programming language, since the goal is to prepare the student to Machine Learning.

- **Python (A)**: Python is a programming language designed to look a lot like plain English, to be simple to learn, to be dynamically typed, to be interpreted, and to have a richness of syntactic expression. This makes it a very good language for discovery through code and experimentation/prototyping. However, this language is sometimes quite slow at runtime, not easy to develop cross-platform, and dynamically typed. This makes it often avoided in production-grade environment, unless special care is taken. Python is above all the language of research, learning, scripting, and prototyping.

- **Jupyter (AMELTO)**: Technology used to launch a local server which makes one capable of running Python in a browser. It is very useful for easy Python development, given the problems of cross-platform distribution of Python, as well as those in the production of Python executables.
- **Numpy (AMELT)**: Standard linear algebra library in Python.
- **Scipy (AMELT)**: Standard scientific computing library in Python.
- **SciKit-Learn (AMELT)**: Standard Machine Learning library in Python.
- **PyTorch / Torch (AMELT)**: A sort of combination of Numpy and SciKit-Learn which has in recent years become the new standard for Machine Learning development, and increasingly general data science as well.
- **Pandas (AMELT)**: Library for managing dataframes (data tables) in Python. Very useful, quite standard in data science.
- **Polars (AMELT)**: Library for managing dataframes (data tables) in Python. Very useful, less famous, but more performant than Pandas.
- **Matplotlib (AMELT)**: Complete, but not very pretty, data visualization library in Python. A bit complex to get into at first, but essential to know.
- **Seaborn (AMELT)**: Data visualization library specialized in data science. Does every visualization task that is very "classical" quite well, simply, and beautifully. It is however sometimes difficult or impossible to do data visualizations with it, if these visualizations are too custom.

Computer Science: famous Apache technologies for (big) data engineering

This section presents some of the distributed technologies under Apache license that are often used for big data management and analytics.

- **Apache Software Foundation, ASF (A)**: the Apache Software Foundation is a decentralized, non-profit corporation. It oversees the maintenance and evolution of a lot of software distributed under the Apache license, which provides open-source and free access to the code to which this license applies, while also leaving way to build closed-source software, or provide software services, based on the Apache-licensed software. This made it so that a lot of software companies, famous ones included, have built their business using Apache-licensed software.
- **Zeppelin (CELG)**: Zeppelin is a code notebook technology (like Jupyter) under the Apache license, and readymade to practice working with some famous Apache technologies (like Spark).

- **Hadoop (BE, CMLTG)**: Hadoop is an open-source, distributed computing framework / platform. It consists of 3 major pieces of software / architecture (**HDFS**, **YARN**, and **MapReduce**) that together, allow for horizontal scaling of computer systems. Hadoop was made to allow high levels of fault tolerance (i.e., often allowing for the distributed system to keep working even when multiple whole computers crash together). Hadoop also allows you to scale your system horizontally without downtime (a fancy, shorter way to say "you can add computers to the system while the system is running, without needing to restart the system"). Hadoop's various parts work as a cluster of machines, under a master / slave architecture (where one machine directs what the other machines have to do).

- **Hadoop Distributed File System, HDFS (CE, DMLTG)**: one of the three core parts of the Hadoop framework. It is the part that takes care of the storage and servicing of data. Like (almost?) all filesystems, it is block based, meaning that large files are cut up into chunks of a (fixed, configurable) maximum size, then saved where appropriate within the storage device(s). However, unlike most filesystems, data blocks are replicated multiple times (by default, 3) across different machines (and even, if necessary, different racks), with a spread of blocks that also improves performance. This is one of the major factors that allows Hadoop to be fault tolerant. In practice, blocks for data that's more frequently accessed are replicated more times.

  HDFS can be monitored and managed via a CLI (with most standard unix commands work (but not 'cd'!); and others (like 'put', for uploads); or browsing techs in its ecosystem, like Hue.

  In regular HDFS, you cannot edit files. You can only store, retrieve, change somewhere else, store the updated version as a new file, then delete the old version. You can append to files, however. But not edit. At least not without some other Apache technology, like Spark.

  HDFS is generally composed of the following pieces:

  - a *gateway* machine, which ensures access to the cluster and points to the Active NameNode as well as the Active Resource Manager (see YARN definition)
  - the *NameNode*: the master node of the cluster, when it comes to handling storage information. It keeps a directory tree of all files in the distributed file system in form of metadata, and tracks where, across the cluster, each file's data is kept. Client applications talk to the NameNode whenever they wish to locate a file, or when they want to add/copy/move/delete a file. The NameNode responds the successful requests by returning a list of relevant DataNode servers where the data lives. There is a single Active NameNode per cluster, and potentially many StandBy NameNodes; these serve as a failover

to maintain uptime in case the Active NameNode goes down; they elect a new Active NameNode via a leader election algorithm. The industry convention is that around 5000 DataNodes, you start to need one more NameNode to handle the volume of compute and memory required.

  - the *DataNodes*: these are the storage slave machines. They contain various data blocks.

- **MapReduce**: a computation software architecture for distributed parallelism, allowing large-scale data processing. It was originally invented by Google, taking inspiration from the "map" and "reduce/fold/join/flatten" operations of monads and functional programming (which provide a way to abstractly express parallel operations, but require pure functions), and popularized by Hadoop. The main interest of such an approach is to distinguish fields of competencies: mathematicians can concentrate on the algorithms to run, computer scientists can concentrate on how they are run in software, and computer engineers can concentrate on how they are run in hardware. A core idea, rather than bringing data to processes, is to instead bring processes to data: like a human election; not everyone comes to the capital to vote. We vote locally, count locally, than tally results nationally by summing the local counts.

  In practice, MapReduce consists of the following operations:

  - **map**: some mapping function f (a pure function from an input data type A to an output data type B) is chosen (this is up to the developer to code). The data of type A we want as input is **split** (automatically) amongst the cluster (where possible, it kept locally on the same node where the operation will run). The function is applied to all elements of type A, and the result is kept as **key-value pairs** $(A : B)$, so that both inputs and outputs can be used for the next step (this is a key difference with the standard monad "map" operation). Critically, the data is (automatically) stored on disk (written to ROM) in order to survive a shutdown mid-MapReduce, though this greatly increases computation time.
  - **reduce**: The data is **shuffled** (we aggregate partial results based on their keys; this happens automatically, but is quite costly) and **sorted** (we sort the KV-pairs by keys for some natural order, e.g.: alphabetical; this happens automatically). The **partitioner** (software) distributes the KV-pairs across reducers (by default, the hash partitioner works automatically, but you can also specify how each key should be divided, in a custom way). The data is then "reduced"/**"folded"**: we combine partial results for each key based on some operator (this is code that needs to be written by the developer). Finally the data is **aggregated**: we bring back the results from each key together to get the results list for all keys.

Various remarks about and pros and cons of MapReduce:

  - In practice, we generally want a 10:1 ratio of mappers to reducers.
  - It is difficult to write filters with MapReduce (even if they are fundamental to monadic parallelization) because of the KV pattern. Spark alleviates that.
  - No data movement / node locality principle: MapReduce works well when executed on the node where the data resides; and it helps enable that. However, with larger data over a network, this can be complicated for Hadoop to handle.
  - Since there's no possibility to share memory, MapReduce is pretty terrible with (node) graphs. For a while, there was Giraph, but it's quite complex to use. Today, most people use Spark's GraphX.
  - MapReduce works with pure functions only, so must be stateless. It's thus difficult to run some algorithms like K-means.
  - MapReduce is very, very bad at real-time (i.e.: no online transaction processing, since that goes too fast to follow for MapReduce).
  - YARN distributes who will run the various operations of MapReduce in Hadoop.
  - Fault tolerance / machine failures are not a problem with MapReduce.

- **Yet Another Resource Negotiator, YARN (CE, DG)**: Hadoop's resource manager. It allows the launching of apps (mappers and reducers) on the cluster. It can run any program on the cluster, not just a MapReduce program. If a target execution machine is too busy, YARN will deal with it and ask some other machine. If the target execution machine is running many things, YARN will know not to delete shared resources (such as a JVM) when one of them ends. Various tools exist to manage YARN: YARN Web UI, Hue Job Browser, and YARN CLI.

  There are various components to YARN:

  - a gateway machine, which ensures access to the cluster and points to the Active NameNode as well as the Active Resource Manager (see HDFS definition)
  - *Resource Manager, RM*: the master machine for task scheduling. It mediates the resources amongst the cluster between competing applications. It has two main elements, a scheduler (which just allocates memory, CPU, and network resources to containers), and applications manager (accepts jobs submissions, finds an AppMaster container to start it, monitors the cluster, and relaunches any AppMaster in case of error). Originally, the RM was a single point of failure. However, in recent versions, multiple machines are Resource Managers: one is Active, and multiple are Standby. This is a failover: in case of RM failure, automatic leader election of the standby nodes).

  - *AppMaster, AM*: manages an app's lifecycle. It negotiates with the NM to get containers on various machines and run the distributed application (algorithm). As a general rule, every app has its own, single AppMaster, but you can also build an AppMaster for a set of apps.
  - *Node Manager, NM*: manages the resources on a single machine. It leases jobs and resources for applications, and controls the lifecycle of multiple containers. Does not manage tasks, only resource usage by tasks (kill in case of memory overuse, etc).
  - *Containers*: they're a unit of resources that are allocated for applications. There are multiple containers per machine. A container handles environment variables, dependencies, security tokens, application processes, computation, etc.

  Here is the overall process of an app's lifecycle in YARN:

  - A client submits an application (a task) to the RM.
  - RM chooses a container to be an AM.
  - AM contacts various NMs, so that it can obtain containers on various nodes to assign the computational gruntwork (including calling the HDFS NameNode).
  - NMs launch the containers (and has them run the AM's algorithm). When done, each NM notifies the AM.
  - AM reports the end of the task to RM, shuts itself down, and releases its own container.

- **Cloudera (AG)**: famous company that provides services based around Hadoop and its ecosystem.

- **Hadoop ecosystem (CE, DMLTG)**: Hadoop was foundational, but had multiple issues (mentioned in the defintions for HDFS, YARN and MapReduce). Consequently, a bunch of companion technologies were built, also under the Apache license, in order to resolve these issues. Here is a non-exhaustive, imperfectly described, list:

  - Hue: a filesystem explorer for HDFS and YARN (Hue Job Browser), in a browser
  - Hive: for SQL-like querying over HDFS; converts an SQL query into a MapReduce program to run on Hadoop; now mostly replaced by Spark SQL
  - Pig: for data flow scripting, much less popular than Spark nowadays
  - Sqoop: for SQL data import/export into HDFS; it can push to multiple destinations.

- Flume: for NoSQL data import/export into HDFS; unlike Sqoop, it is point-to-point, only one place to read from, and one place to send, so generally, one sends data to a Kafka cluster to distribute to various places.
- HBase: for NoSQL databases and real-time databases. Helpfully, can do real-time reads and writes inside Hadoop, and not on the entire block. It has its own custom language. A sister technology, Phoenix, can do SQL queries on HBase.
- Flink: real-time processing, mostly replaced by Spark
- Storm: real-time processing, mostly replaced by Spark
- Impala: for real-time queries (very fast, in-memory, mail fail, unlike MapReduce), Cloudera tech
- Hawq: same as Impala but not a Cloudera tech.

- **Spark (CMEL, DTG)**: Spark is an open-source analytics engine under the Apache license. It is a framework used for real-time processing (e.g.: immediate fraud detection) and batch processing (e.g.: weekly reports). Its main aspect is its in-memory execution engine, this makes it faster than MapReduce. Also, because it is in-memory, it cannot resist all failures, but Spark has a clever "lineage" mechanism to minimally reconstruct parts of the data that were lost during a failure. Finally, Spark uses a lazy evaluation mechanism: it creates an execution plan, and optimizes it before running it. It only effectively runs when prompted (through function calls like collect(), count() or show()).

  Its various facets include:

  - Spark Core: provides basic I/O, task dispatching, task scheduling, memory management, and fault recovery. Can work with Hadoop, but not mandatory.
  - Spark SQL: run sql on structured data
  - Spark Streaming: operates on data streams (ie, for "near" real-time processing, e.g., receiving stuff from kafka). This allows the same code that was written for batch processing to be run on a single small machine to get results in real-time.
  - MLlib: provides classification, regression, clustering, etc.
  - GraphX: provides (node) graph analysis. Relies on Pregel (apparently a combination of the words Parallel, Graph, and Google) which is a data flow paradigm and system for large-scale graph processing.

  Its core concept, and fundamental data structure, is the Resilient Distributed Dataset (RDD). They are in-memory, distributed collections of data. They are immutable (once created, they cannot be edited, but can be used to create new RDDs), resilient (they can be reconstructed, in case of failure, from lineage information); typed or untyped; and can handle

multiple types and collections of data. Basic operations on RDDs include: map, filter, reduce, join (in the sense of SQL), groupByKey (makes a dictionary to prepare aggregation), reduceByKey (per key aggregation), sortByKey, distinct (returns an RDD with unique values), union, intersection.

How it works:

  - Spark works as a column-centric database (ignore all attributes not part of the compute), thereby greatly decreasing memory load. This also simplifies the compression of similar values.
  - Everything is loaded into memory, removing the need for things likes indexes, aggregates, etc. This also somewhat simplifies concurrency, since everything is already loaded, and reduces querying costs. All-in-all, it's a 10^4 to 10^6 factor speedup over Hadoop's MapReduce.
  - Spark does not need Hadoop, but implements a way to use Hadoop (and other such DFS).
  - Spark allows a lot of the analytics features to be done in a single common platform, greatly reducing codebase complexity.

- **Kafka (CMELT, DG)**: Kafka is an open-source distributed event store and stream-processing platform. It generally is implemented in its own cluster. It receives data in a pub/sub manner (typically, when something like Flume brings data into HDFS, it can on the side send it to Kafka simultaneously). Multiple processes can produce the data for, or consume the data from, the Kafka cluster simultaneously. Data points on a Kafka server is stored for a default of 7 days. It used to depend on Apache ZooKeeper for distributed coordination and management tasks. However, recent Kafka versions are moving away from this dependency to depend on Kraft (Kafka Raft). Typical use cases of Kafka include log aggregation (making sure the monitoring of a system can be centralized), event sourcing (analyzing user behavior on a website), or metrics collection (such as IoT data or embedded systems for physical measurements).

  Here is Kafka's terminology:

  - *Message*: a small to medium piece of data (raw bytes).
  - *Producer*: an application that publishes data to Kafka. They write messages to topics, and Kafka stores them for a specified retention period.
  - *Consumer*: an application that reads data from Kafka. They subscribe to topics and process the data. They read messages from partitions and can process them in real-time.
  - *Broker*: a kafka server. Kafka brokers are servers that store data, serve client requests, and manage partitions. A Kafka cluster consists of multiple brokers working together.

- *Cluster*: a group of computers sharing a workload for a common purpose.
- *Topic*: an identified (named) data stream. Kafka data is organized into such topics. Topics are like named channels or feeds where data records, referred to as messages, are stored and published. 1 or n producers can publish to the same topic; 1 or n consumers can subscribe to the same topic.
- *Partition*: a fraction of a topic. Each topic is divided into partitions, which are the basic unit of parallelism and scalability. Partitions allow data to be distributed across multiple servers. The developer who maintains the Kafka server(s) must manually say how big a partition should be for each topic. A given partition cannot be re-subdivided.
- *Offset*: position of a given message in a partition. It's the index at which one can find a given message in a partition. Offsets are immutable, and local to a partition. Any message can be found with the trio TopicName-PartitionNumber-Offset.
- *Consumer* Group: a logical unit of multiple consumers sharing a task. Multiple consumers can be organized into consumer groups. Each partition in a topic is consumed by only one consumer within a group, enabling parallel processing.

The things that Kafka provides, or not:

- publishing data: Producers send messages to Kafka topics. These messages are appended to the partitions of the topic.
- storage: Kafka stores these messages for a defined retention period. This storage duration can be configured.
- consuming data: consumers subscribe to topics and read messages from partitions. They can track their progress in each partition using offsets.
- scaling: Kafka scales horizontally by adding more brokers or consumers to handle increased load.
- ideally, the Kafka Producer application needs to be installed at the source; if impossible, you can use Flume, which is pull-based.

To query topics, Kafka uses a concept called *windowing*. This serves to do more interesting time-based analytics, by running topic like time series, and replicating the behavior of algorithms on signals from A to B. Here are the 4 main types of Kafka windows.

- *hopping*: bound by time, with fixed points (start A, and end B), a fixed size, and a fixed advance size.
- *tumbling*: same as hopping, but the advance size is equal to $B - A$.

- *sliding*: like hopping and tumbling, it has fixed A, B and fixed advance size. However, it advances with user activity.
- *session*: A and B are not fixed points, but determined by the events themselves. An inactivity gap defines the session window. Each new event is added to the current session window, and when no events come in "inactivity gap" amount of time, the session window ends, and a new one is created.

Math for Data Science

Math for Data Science: Abstract Algebra

This section describes the fundamentals that one tends to see during the first weeks of an undergrad of mathematics. While mastery is not essential for machine learning, understanding the following concepts makes the learning of all the mathematics related to data science much, much easier.

- **Set (A)**: a set is a collection of mathematical objects. For example, the binary alphabet is the set containing the symbols 0 and 1, noted $\{0, 1\}$. The natural numbers form another set, noted $\mathbb{N} = \{0, 1, 2, 3, \dots\}$. You can think of a set as a deck of cards are unique, and there might be an infinite amount of cards.
- **Space / algebraic structure (A)**: a space is the combination of a set and with some rules/laws/properties and/or operators on its elements. For example, the '+' operator on the binary language (set of all combinations of symbols of the binary alphabet noted $\{0, 1\}^* = \{$"","0","1","00","01","10","11",...$\}$) is concatenation, and glues the "words" together (e.g., `"1010" + "111" = "1010111"`). The '+' operator in natural numbers is addition (e.g., $5 + 7 = 7 + 5 = 12$). An example of a property is commutativity: $a + b = b + a$. This property is verified in the natural integers but not in the binary language. You can think of properties as the rules you want to choose to play with your chosen deck of cards.
- **Function (mathematics) (A)**: a function is an association of the elements of a set of inputs (called a domain) with a set of outputs (called a codomain), where each input has at most one output image, so that the result of a function, given a fixed input, is deterministic. The square root is an example of a "1 input, 1 output" function. The classical operators (like addition) are examples of "2 input, 1 output" functions (for example, written in this way $+(5, 7) = 5 + 7 = 12$, this fact becomes clearer).
- **Application / Map (AG)**: You can consider that "application"/"map" and "function" are synonymous terms in mathematics (by abuse of language, even if there is a technical distinction between the two in French).

- **Image (AMELT)**: the image y by a function f of an element x of the domain is the unique element $y = f(x)$ of the codomain. Put more simply, the "image" is the result value of applying the function, for a given input/output pair.
- **Antecedent (AMELT)**: an antecedent x of an element y of the codomain of a function f is an element of the domain such that $y = f(x)$. Put more simply, the "antecedent" is the "origin" of some result of the function, for a given input/output pair.
- **Variable (mathematics) (A)**: a variable is a value which is named and typed (declared to be a member of a certain space) but not specified. This makes it so that this named value can act as a placeholder, to represent "any value" from the space in which it exists.
- **Parameter (A)**: a parameter is a variable which is given a specific value, for experimentation's sake, but which could just as easily be changed.
- **Function composition (A)**: if f is a function from A to B and g is a function from B to C, then there exists a function $h = g \circ f$ from A to C, such that h is the concatenation of f and then g (ie, first applying f, then g), where the return value of the function f is given as an argument to g.
- **Continuous function (A)**: a function is said to be continuous if its graphical representation (linking its inputs and outputs) has no "break".
- **Derivable function (A)**: a derivable function is a continuous function which does not have any "sharp corners" in its graphical representation.
- **Derivative (A)**: the derivative of a function is another function, representing the slope of rise or fall at each point of this function as a numerical value. A rise for an input point corresponds to a positive value of the derivative at that same input point; a fall corresponds to a negative value at that point.
- **Exponential function (A)**: the exponential function is the "fundamental" function whose derivative is the exponential function itself. Among its many properties, we have $e^{a+b} = e^a \cdot e^b$, i.e., it transforms an addition of inputs into a multiplication of outputs.
- **Logarithmic function / Neperian logarithm (A)**: the reciprocal of the exponential function. The derivative of the neperian logarithm is the function $(x \to \frac{1}{x})$. Also, $\ln(a \cdot b) = \ln(a) + \ln(b)$, i.e., the neperian logarithm turns a product into a sum.
- **Logistic function (AML, BET)**: The logistic function is a continuous, strictly increasing function, varying from 0 to 1 (in its outputs) from $-\infty$ to $+\infty$ (in its inputs).

- **Integral (A)**: the integral is the "ruler" (in the sense of a tool to measure) of mathematics. It is also the "anti-derivative".

- **Symbolic notation of iterated sums and products (AMELT)**: A large Σ represents an iterated sum, a large Π represents an iterated product. Example: $\sum_{i=0}^{i=5} 2^i = 2^0+2^1+2^2+2^3+2^4+2^5 = 1+2+4+8+16+32 = 63$.

Math for Data Science: Linear Algebra

This is the meat of data science as a subject. All of machine learning, statistics, data science, most of physics, economics, etc, rely on linear algebra in some form or another.

- **Vector space (AMELT)**: a vector space E is an algebraic structure with commutative addition and subtraction (elements of the vector space, called **vectors**, behave like an "abelian group" under addition). It is always accompanied by another structure K called "field", itself having usual addition, subtraction, multiplication and division (except by 0): this is the usual arithmetic that you've seen throughout school. Elements of K, called **scalars** allow one to change (scale) the size of vectors. This structure is coupled with a series of laws that allow the combined operation of K and E in a correct way.

- **Scalar (A)**: a scalar is an element of K. In general, we choose $K = \mathbb{R}$, the field of real numbers. A scalar is a tensor of rank 0.

- **Vector (A)**: a vector is an element of E. In general, a vector will be an array of n elements. The number n is then shared by all elements of E and is called the "dimension" of E. E is then denoted $\mathbb{R}^n$ (for the vector space of real numbers of dimension n). If n can also be some infinite quantity. For example, the functions from $\mathbb{R}$ to $\mathbb{R}$ are also an example of vectors, because they respect the same laws (addition and scaling) within their enclosing space: the space $(\mathbb{R} \to \mathbb{R})$ is a vector space. A vector is a (contravariant) tensor of rank 1.

- **Vector addition (A)**: an operation representing addition of $E \times E \to E$ (2 inputs in E, one output in E). Addition is done by matching the basis elements of each operand, and summing each respective pair of coefficient scalar: e.g. $(1, 2, 3) + (10, 20, 30) = (11, 22, 33)$

- **Vector scaling (A)**: an operation which is a form multiplication of $K \times E \to E$ (1 input in K, 1 input in E, 1 output in E). It is done by multiplying all coordinates of a given vector by the same chosen scalar: e.g. $4 * (1, 2, 3) = (4, 8, 12)$.

- **Linear combination (AMLT, BE)**: any possible arrangement of scalings and additions of a set of vectors. For example, if we have 3 vectors

u, v, and w, then $72 \cdot u + \frac{1}{3} \cdot v - 4.5 \cdot w$ is a linear combination of vectors u, v, and w. We can always simplify a linear combination back to a form where each vector is multiplied by a single scalar, and the list of these scalar-vector products is summed (as is used in the example).

- **Linear independence (AMELT)**: a set of n vectors is linearly independent if the only way to get the zero vector (the zero of E, the element that changes nothing by vector addition) by linear combination is to have the scalar factor in front of each scalar-vector pair be zero. Intuitively, this means that each vector in the set contributes to creating a new dimension, independent of those created by the other vectors: there is no way to express one of the vectors as a linear combination of the others.

- **Basis (A)**: a set of linearly independent vectors that can describe any point in a vector space. There are always precisely as many elements in a basis as there are dimensions in a vector space.

- **Dimension (A)**: number of elements in any basis of a vector space; number of axes needed to represent a vector space.

- **Colinearity (AMELT)**: two vectors are said to be colinear if their direction describes the same line passing through the origin. This means that one can be scaled into the other.

- **Linearity / Linear application (AMELT)**: Said of a function from E into F, where E and F are vector spaces, that sends the origin of E (its zero vector) to the origin of F, and keeps all parallel lines in E parallel in F. Algebraically, a function/application is said to be linear if and only if $f(0_E) = 0_F$ and $f(ku + v) = kf(u) + f(v)$, for all k in K and all u and v in E. Geometrically something "linear" is something straight (a line, a plane, etc); something non-linear is something curved (a sphere, a parabola, etc). "Linear" often also refers to polynomials of degree 1 (which have straight shapes), "quadratic" refers to polynomials of degree 2, "cubic" to polynomials of degree 3, etc.

- **Matrix (AMELTG)**: a matrix is a rectangle of scalars. A matrix of size $m \times n$ (m rows, n columns) represents a linear map of $\mathbb{R}^n \to \mathbb{R}^m$ (i.e., a function with as input a vector of dimension n, and as output a vector of dimension m). A matrix is a tensor of rank 2.

- **Matrix multiplication (AMELTG)**: matrix multiplication is the function composition of linear maps.

- **Dot product (A)**: the dot product is a product of $E \times E \to K$ (2 vectors of E as input, one scalar of K as output). It is computed as the sum of the term-to-term product of each coordinate of its two inputs. It is denoted $\langle u, v \rangle$, and is also equal to $\|u\| \cdot \|v\| \cdot \cos(u, v)$. Algebraically, for two vectors u and v of dimension n with coordinates u_i and v_i, the dot product is defined as: $\langle u, v \rangle = \sum_{i=1}^{i=n} u_i \cdot v_i$. The dot product algebraically

encodes two pieces of geometric information (one about angles, and one about lengths) into a single number. In practice, the dot product of two vectors of norm 1 is equal to the angle between these two vectors; the dot product of two collinear vectors is equal to the multiplication of their norm.

- **Norm (A)**: The norm of a vector is the distance of displacement that this vector represents. It is usually defined using the dot product, as $\|u\| = \sqrt{\langle u, u \rangle}$ (Euclidean norm, also called the 2-norm, the one from the Pythagorean theorem). There are different norms (useful in ML), like the "supremum norm/∞-norm" or the "Manhattan norm/1-norm".
- **Quadratic norm (AG)**: the quadratic norm is the norm of a vector squared. It is usually defined as $\|u\|^2 = \langle u, u \rangle$.
- **Cosine (A)**: value used to define the angle between 2 vectors. Algebraically, $\cos(u, v) = \frac{\langle u, v \rangle}{\|u\| \cdot \|v\|}$.
- **Distance (metric) (A)**: function to define a measure of the distance between two vectors. If the vector space is normed (has a norm function), a distance can always be defined from it as $d(u, v) = \|u - v\|$.
- **Span (generated vector subspace) (AMLT, BE)**: the vector subspace generated by a family (set) of vectors is the set of points that are reachable by linear combinations of the vectors of this family.
- **Normalization (linear algebra) (AMELT)**: vector scaling of a vector u by the value $\frac{1}{\|u\|}$, in order to find a vector $\hat{u}$ of norm 1 and the same direction as u.
- **Tensor (AMELT)**: A tensor is an arrangement of scalars. A tensor of rank zero is a point of numbers (a single scalar). A tensor of rank one is a line of numbers: either a vector (column vector) or a covector (row vector). A tensor of rank two is a rectanle of numbers (a matrix). A tensor of rank three is a cube of numbers (hypermatrix). Etc.
- **Linear form / covector (AMELT)**: A linear form is a linear map of $E \to K$. Linear forms are row vectors.
- **Vector subspace (AMLT, BE)**: A vector subspace is a vector space contained in another vector space. A vector line, or a vector plane (i.e. passing through the origin) in a vector space of dimension 3 are examples of a vector subspace. Any space is a vector subspace of itself.
- **Hyperplane (AMLT, BE)**: A hyperplane is a vector subspace of dimension $n - 1$ in a vector space of dimension n. The 2D planes are the hyperplanes of 3D space. The 1D lines are the hyperplanes of 2D space. The role of a hyperplane is to separate a vector space into exactly 2 pieces. For example, any "mirror" symmetry is done with respect to a hyperplane, whatever the dimension n of the enclosing space.

- **Eigenvectors / Eigenvalues (BMLT, CE)**: Any linear map has vector subspaces that are stable (i.e. if an input is in that special stable subspace, then so is its output by the linear map in question). The are called eigenspaces. Linear maps in 1D eigenspaces amount to dilations (since both input and output are colinear). The vectors serving as the basis for these vector subspaces are called eigenvectors; their respective dilation coefficient is called an eigenvalue.

- **Euclidean space (AMLT, BE)**: vector space along with a dot product operator, allowing it to define a norm (the Euclidean norm) and therefore a metric (the Euclidean metric). This is the usual kind of space in which we do math and physics in high school.

- **Normed vector space (BMLT, CE)**: vector space along with a norm operator over vectors, allowing it to define a metric.

- **Metric space (BML, CET)**: vector space having a notion of distance between vectors (though not necessarily a notion of size for vectors).

Math for Data Science: Probability Theory

This section describes the math of probability theory, which forms a lot of the backbone of statistics. Statistics is basically the combination of probability and linear algebra.

- **Universe (of discourse) (A)**: set describing the collection of possible outcomes of a random experiment (some situation with a random outcome). For example, the universe of discourse for a single roll of a regular (six-sided) die (also called a "1d6") is the set $\Omega = \{1, 2, 3, 4, 5, 6\}$.

- **Probability (A)**: A probability is a function over a universe of discourse that returns an outcome between 0 (ie, 0% chance) and 1 (ie, 100% chance). This universe of discourse is geometrically an object of measure 1 (a length of 1 in dimension 1, an area of 1 in dimension 2, a volume of 1 in dimension 3, an n-hypervolume of 1 in dimension n). The probability function assigns a measure between 0 and 1 to all collections of outcomes. For example, in the universe of discourse described above (the 1d6), the event "an even number is rolled" corresponds to the set $e = \{2, 4, 6\}$ and is precisely one half of the full space (measure 0.5), which corresponds to a 50% chance.

- **Conditional probability (Bayesian probabilities) (A)**: A conditional probability is a probability considering/knowing that an event e is necessarily true. This is geometrically equivalent to restricting oneself to the section of the universe of discourse where e is true, and considering that this new geometric object is now of measure 1 (ie, certain). For example, take a 1d6, $e_1 = \{1, 2, 3, 4\}$, ie, "rolling 4 or less", and $e_2 = \{2, 4, 6\}$, ie,

"rolling an even number". Then, the probability of "rolling an even number *knowing that* we rolled 4 or less" is 0.5, since the set $e_3 = \{2, 4\}$ is half the measure of the the set $e_1 = \{1, 2, 3, 4\}$ which is our new, "knowing that", universe. This should not be confused with the probability of "rolling an even number *and* rolling 4 or less", which would be 1/3.

- **Random variable (AML, BETG)**: probabilistic experiment where the result is assigned a "success" value (in general a real number). Algebraically, a random variable is a function of the universe of discourse in (in general) the set of real numbers, usually denoted $X : \Omega \to \mathbb{R}$. For example, let's say I win 0.50€ per point on a 1d6 die, but if I roll a 6, I instead lose 3€. Our function looks like $X = \{1 \to 0.5; 2 \to 1; 3 \to 1.5; 4 \to 2; 5 \to 2.5; 6 \to -3\}$.

- **Random vector (BML, CETG)**: probabilistic experiment where the outcome is assigned multiple values (usually n real numbers). For example, if I refer to the value of two normal throws of the dice in order (2d6) as "x" for the first throw, and "y" for the second, then the example of a random vector where I win 2x candies but lose 0.5xy€ on each pair of throws can be defined as $X = (x, y) \to (2x, -0.5xy)$. Models in data science are usually random vectors that try to match the shape and properties of a cloud of data points, if the probabilistic experiment is performed repeatedly on different points.

- **Expectation (A)**: average of the results of a random variable over a universe, weighted by their probability. It corresponds to the "average return that can be expected in the long run, if we repeat the experiment an infinite amount of times".

- **Variance (A)**: Average of the squared deviations of each of the outcomes from the expectation. This gives an idea of the "spread" of the values of the experiment away from the average. The variance can also be understood as the covariance of a random variable (or vector) with itself, i.e. the quadratic norm of a random variable. In the case of a random vector, the variance takes the form of a symmetric, positive semidefinite matrix, called the variance-covariance matrix.

- **Standard deviation (A)**: the standard deviation is the square root of the variance of a variable or a random vector. It is therefore the norm of a random vector, noted σ_X.

- **Covariance (AML, BET, CG)**: The covariance is a measure of how much two random variables vary together. It is the dot product of spaces of random vectors. It is noted $\text{cov}(X, Y)$.

- **Pearson's correlation coefficient (AMLT, CG)**: a measure of how closely two events are correlated. It is the cosine of the spaces of random vectors. It is defined as $\text{cor}(X, Y) = \frac{\text{cov}(X,Y)}{\text{stddev}(X) \cdot \text{stddev}(Y)}$.

- **Bias (A)**: difference between the mean/expectation given by the model, and the one given by the experiment (the real data).

Math for Data Science: Multivariable calculus and differential geometry

This is a more advanced mathematical subject, but it is necessary for more advanced topics in data science, such as the explanation for several advanced algorithms, explainability in ML, fields of data science like topological data analysis, etc.

- **Scalar function (BMLT, CE)**: A scalar function is a function from $\mathbb{R}^n \to \mathbb{R}$ that is not necessarily linear (unlike linear forms). Ex: $f : \mathbb{R}^3 \to \mathbb{R}$, such that $(x, y, z) \to xy + z^2 + \cos(x)$.
- **Partial derivative (BMLT, CE)**: derivative of a single coordinate of scalar function. Each coordinate is obtained with the standard derivative, considering all coordinates constant, except one. E.g., going with the previous function, here is the partial derivative in the x coordinate: $\frac{\partial f}{\partial x} = y - \sin(x)$.
- **Differential / gradient (BMLT, CE)**: The differential / gradient is the "fundamental" derivative of a scalar function. The differential gradient indicates the direction in which to move to get the largest increase in the output value. The differential / gradient is calculated as the vector of all partial derivatives. The only difference between the two is that the differential (noted df) is a row vector (covector) and the gradient (noted ∇f) is a column vector (i.e., their input/output space are reversed). E.g., going with the previous function:

$$df = \begin{bmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} & \frac{\partial f}{\partial z} \end{bmatrix} = \begin{bmatrix} y - \sin(x) & x & 2z \end{bmatrix}$$

- **Multivariate function (BMELT)**: a multivariate function is a function from $\mathbb{R}^n \to \mathbb{R}^m$, which is not necessarily linear. E.g.: $f : \mathbb{R}^3 \to \mathbb{R}^2$ such that $(x, y, z) \to (xy + \cos(x), xyz + z^2)$. Note that we can write $f(x, y, z) = (f_1(x, y, z), f_2(x, y, z))$, where $f_1(x, y, z) = xy + \cos(x)$ and $f_2(x, y, z) = xyz + z^2$ are both scalar functions.
- **Jacobian (BM, CELT)**: the "fundamental" derivative of a multivariate function. The Jacobian is a matrix of size $m \times n$, where the i-th row corresponds to the differential of the i-scalar function. To caricature, "we stack the differentials". Note that the Jacobian is itself a function from $\mathbb{R}^n \to (\mathbb{R}^n \to \mathbb{R}^m)$; that is, your Jacobian gives you a different matrix for each point in your space where you compute it. Geometrically, the

Jacobian gives you the best linear approximation at a point (x, y, z) of its return in the output space.

$$J_f = \begin{bmatrix} \frac{\partial f_1}{\partial x} & \frac{\partial f_1}{\partial y} & \frac{\partial f_1}{\partial z} \\ \frac{\partial f_2}{\partial x} & \frac{\partial f_2}{\partial y} & \frac{\partial f_2}{\partial z} \end{bmatrix} = \begin{bmatrix} y - \sin(y) & x & 0 \\ yz & xz & xy + 2z \end{bmatrix}$$

- **Manifold (AMLT, BEG)**: a manifold is an object which is "locally euclidean". This means that, if you zoom in enough on your object, it looks flat. It's a kind of object on which we can define a geometry through which differentiable functions can exist. Any vector space, as well as any space of functions between vector spaces, is a manifold. The sphere, the torus, the Klein bottle, the n-dimensional Cartesian plane ($\mathbb{R}^n$), are all examples of manifolds.
- **Topology (BMLT, CEG)**: a branch of mathematics that studies the local and global structures of manifolds. Also, a term for a "fabric of reality / choice of mathematical universe" based on the language of sets, on such spaces, in order to define notions of distance, measure, continuity, derivability, etc.
- **Measure theory (DML)**: extension of topology to build a more powerful version of the integral, and to define a way to compute lengths, areas, volumes, whatever the dimension. Probabilities are geometrically understood as "n-volumes relative to a global space of n-volume 1", and measure theory is thus the foundation of all modern probability theory.
- **Topological data analysis (DML)**: a technique that aims to study the geometric form (the manifold) on which data points exist in an encompassing statistical vector space, in order to drastically reduce its dimension or to make its logical structure explicit.
- **Differential Geometry (DML)**: theory of derivation on manifolds.
- **Information Geometry (DML)**: mathematical theory relying on probability, statistics, information theory, measure theory and differential geometry to establish statistical spaces (random vector spaces; data point cloud spaces) and dynamics on general manifolds.

Data science & Machine Learning

General vocabulary of statistics, data science and ML

Here we define terms that are useful all around for scientific/technical data science and ML.

- **Regression (AML, BETG)**: a type of model learning that seeks to predict numerical values for fictitious input data, close to the real output data in relation to its input data.

- **Classification (A)**: a type of model learning that seeks to predict the categorization of some output based on an input data point.
- **Model (A)**: a model is a mathematical function used as a hypothesis to generate a virtual data point from a chosen input. The goal of a model is to predict the actual data as closely as possible.
- **"Garbage in, garbage out" (A)**: a saying emphasizing the importance of quality data for accurate results in data science and machine learning.
- **R-squared (BML, CT)**: one of the measures of the validity of a model.
- **ML fields of study (AG)**: Natural Language Processing (NLP); Computer Vision; Decision-making; Business analytics; Ranking; Darwinian algorithms/Reinforcement learning, etc.
- **Neural network (A)**: a neural network is an architecture using linear algebra to build a graph of computational cells (neurons) and run an optimization protocol (part darwinian, part search for an optimum of a hard mathematical function) to allow an algorithm to become efficient for a given task in an unsupervised way.
- **Curse of dimensionality (AG)**: the growth of data points required for accurate representation grows exponentially with the number of attributes, making many algorithms inefficient for high-dimensional spaces.
- **Dimensionality reduction (AG)**: techniques used to reduce the dimension of high-dimensional spaces while preserving information. Used for visualization and to make data tractable for algorithms.

Machine Learning: Learning

Here, we define the various terms that make up the "learning" part of machine learning.

- **Supervised Learning (A)**: type of machine learning, where the data is all labeled and specified by a human being, so that the algorithm can use this as a foundation to know if it is right or wrong.
- **Unsupervised Learning / Self-Supervised Learning (A)**: learning where the algorithm itself is supposed to classify, measure or label the data on its own.
- **Semi-Supervised Learning (A)**: A learning approach where a model is trained on a mix of labeled and unlabeled data. It aims to leverage the unlabeled data to improve model performance.
- **Reinforcement Learning (A)**: learning where the algorithm itself perceives its environment, and learns according to it in a pseudo-darwinian way to perform a task more and more efficiently, based on some fitness/cost function.

- **Transfer Learning (BLG)**: A technique where a pre-trained model on one task is fine-tuned on a new, related task. This approach can significantly speed up the training process and improve performance, especially when labeled data is limited. Large Language Models, like GPT-3, are pre-trained on massive amounts of text data and then fine-tuned for specific tasks. This transfer learning approach allows them to perform well on a wide range of language-related tasks.

- **Fitness function / Cost function (A)**: a fitness/cost function is a function that, for a whole series of inputs, calculates the distance between the output of this input passed through the model function and the actual result in the data point cloud. The sum of these distances is the result of our cost function and is a measure of the accuracy of our model (compared to real world data). Machine learning algorithms try to improve fitness / minimize cost, to become better at their given problem. The cost function defines the "moral objective" of the AI. Choosing the right cost function is not just a mathematical or engineering question, but one of AI ethics as well.

- **Hyperparameters (AL, BMET)**: the parameters that allow one to manage the learning rate or direction of an algorithm.

- **Automated machine learning, AutoML (CLG)**: Automated Machine Learning refers to the process of automating the end-to-end process of applying machine learning to real-world problems, including data preprocessing, feature selection, model selection, and hyperparameter tuning.

- **Human-in-the-Loop (ALOG)**: Incorporating human reviewers to guide and oversee generated content (often for GANs or LLMs in particular) which helps ensure higher quality and ethical standards.

Data Science & Machine Learning techniques

Here we describe various techniques or meta-approaches that are common in data science and machine learning.

- **Normalization (data science) (AML, BET)**: a change of benchmark used to rescale the distances between the points of a data point cloud, so as to express the same relative information between these data points, but so that a specific algorithm has better results.

- **Gradient descent (AML, BET)**: an algorithm using iterative gradient calculation to find an extremum of a complex mathematical function (typically, the minimization of a cost function).

- **Principal Components Analysis, PCA (BL, CMETOG)**: This is a dimensionality reduction technique (probably the best known and most widely used). Its principle is to express the underlying frame of reference

of the data space in its most "expressive" form, i.e. that maximizes the variance of the data projected on the axes of the new frame of reference. This allows having a maximum of information on the data cloud with a minimum of dimensions (axes). The origin of this new frame is the mean of the data points, and the axes are computed by Singular Value Decomposition (SVD) of the covariance matrix. Projecting the data points onto the main axes allows visualizing an approximate but rather accurate version of a much more complex high-dimensional data space.

- **Decision trees, random forests (AML, BETO)**: machine learning techniques that build decision trees (conditional "if" trees) from data, used for classification and regression. They are easy for humans to interpret.
- **Kernel methods / Integral kernel / Window functions / Statistical Kernel / Reproducing kernel Hilbert space, RKHS (DMLT)**: methods using covectors of vector spaces of functions, or modified dot products, to transform the shape of data point clouds into something easier to handle (e.g., by linear methods) and achieve better results.
- **Gradient boosting (AL, BMET)**: An ensemble learning technique that combines multiple weak learners (typically decision trees) to create a strong predictive model. It builds the model in a stage-wise manner, focusing on correcting the errors of previous iterations. XGBoost is a popular open-source implementation of gradient boosting that is known for its efficiency and performance in machine learning competitions. LightGBM is another gradient boosting framework that is designed to be memory-efficient and fast, making it suitable for large datasets. CatBoost is a gradient boosting algorithm that handles categorical features automatically, eliminating the need for extensive preprocessing.
- **Attention mechanisms (CL)**: Techniques that allow models to focus on specific parts of input data when making predictions. These significantly improve the performance of NLP models.
- **Neural style transfer (CL)**: A technique that combines the content of one image with the artistic style of another image using neural networks.
- **Anomaly detection (BL, CMET)**: A field of machine learning focused on identifying rare events or outliers in data. It has applications in fraud detection, network security, and more.

Data Science & Machine Learning algorithms

- **Linear regression (AG)**: a technique that allows establishing a model (here, an approximation of a cloud of data points) in the form of a vector subspace minimizing the distance to a set of points.
- **Logistic regression (AG)**: form of regression using the logistic function to obtain a probabilistic model of binary classification (true/false).

- **K-means clustering (AMELT)**: A classification algorithm that seeks to partition a group of n observations (an n-point data point cloud) into k "clusters" where each point belongs to the cluster with the closest mean. The algorithm organizes the points of the initial cloud by iteratively readjusting hyperplanes to converge to a local optimum.
- **K-nearest neighbors (AMELT)**: algorithm for both classification and regression, assigning to each point either the category of its k nearest neighbors (classification) or the average of the values assigned to these k nearest neighbors (regression). It is based on a choice of distance and requires normalization of distances.
- **Support Vector Machines, SVM (AL, BMET)**: supervised learning algorithm for linear classification using hyperplanes. Extensions exist for regression and non-linear classification (using kernel methods).
- **Naive bayes classifiers (AL, BME, CT)**: family of algorithms based on conditional probabilities to perform classification.
- **Multi-Layer Perceptron, multilayer perceptron, MLP (AG)**: a fundamental neural network architecture that takes input, processes through layers, and produces an output. Learning is done using the backpropagation algorithm.
- **Convolutional Neural Network, CNN (AL, BMETG)**: specialized neural network for image processing, using convolution cells to extract local shapes.
- **Recurrent Neural Networks, RNN (AL, BG)**: neural network architecture with linked non-neighboring layers that can affect each other. It is designed to handle sequences of data, making them suitable for tasks like time series prediction, natural language processing, and more.
- **Long Short Term Memory Network, LSTM (AL, BG)**: a type of complex Recurrent Neural Network, capable of learning and remembering longer sequences of data, used for NLP and speech recognition.
- **Large Language Model, LLM (A)**: Large Language Models (LLMs) are advanced artificial intelligence systems designed to understand and generate human-like text based on vast amounts of training data. LLMs, such as OpenAI's GPT-3, utilize deep learning techniques to learn the statistical patterns and structures of language. These models have the capacity to generate coherent and contextually relevant text, making them invaluable tools for natural language processing tasks. LLMs can perform a wide range of language-related tasks, including text generation, language translation, sentiment analysis, chatbot interactions, content summarization, and more. Their capabilities stem from the complex neural architectures they employ, which consist of multiple layers of interconnected

processing units. LLMs have garnered significant attention for their potential to revolutionize various industries and applications by enhancing human-computer interactions and enabling sophisticated language-related tasks. [This definition was generated using ChatGPT with GPT 3.5]

- **Gated Recurrent Unit (GRU) (BL, CG)**: A variation of the LSTM architecture with fewer parameters, often used for similar tasks as LSTMs.
- **Generative Adversarial Networks, GAN (A)**: neural network architecture with two agents, a generator (which acts like forger) and a discriminator (which acts like an inspector), that improve by competing with each other. Used to produce realistic data in image, video or audio format.
- **Transformer Architecture (BL, CG)**: A neural network architecture designed for sequence-to-sequence tasks, such as machine translation. Transformers have revolutionized NLP and are the foundation of models like BERT, GPT, and T5.
- **Q-learning (AL, BMETG)**: Model-free reinforcement learning algorithm widely used for AIs, generally by having them "play video games" (react and evolve in a given simulated environment).

Other: Design

- **Reverse engineering (AG)**: The process of studying the functioning of a product or algorithm to understand its design and production. This can involve reproducing or improving upon the original design.
- **Divergent thinking (AG)**: The ability to think creatively and generate a wide range of possible solutions to a design problem. It involves thinking outside the box and considering unconventional ideas.
- **Convergent thinking (AG)**: The ability to systematically filter and evaluate possible solutions to a problem based on a defined set of constraints. It is similar to the scientific method (aka Cartesian method).
- **Design thinking (AG)**: A mental protocol for creative problem-solving that combines both approaches of divergent and convergent thinking.
- **Algorithm design (AG)**: The process of inventing protocols for automated data processing, involving the creation of step-by-step instructions for solving specific problems.
- **Database design (AG)**: The process of inventing a data model for a domain problem, and selecting the appropriate database architecture to model it.
- **System design (AG)**: The process of constructing an architecture using computers to solve a given problem.

TODOs

Add table of contents and index. Possibly add newpage between sections.

Improve difficulty / importance code usage (right now, it uses A and B too much; many Bs should be Cs for specific profiles, or a G should be added). Probably divide Importance and Difficulty into 2 different codes. Probably use emojis for the code rather than letters, so that they can be search for easily with Ctrl+F by people of the respective profile.

Improve NoSQL section.

Maybe improve mathematical section to always have, for each term: - the ELI5 definition - an example - a non-example - the intuitive technical definition (may be the same as the ELI5 one) - the rigorous, formal definition ? Or just keep an ELI5+ and send people to go read Mathophilia instead ?

Add forgotten words: - unstructured / semi-structured / structured - frontend - backend - fullstack - stateful / stateless - microservice - daemon - functional (system requirements) / non-functional (system requirements) - XML - JSON - HTML - API - Rest API - HTTP (improve to describe in more detail) - HTTPS - TLS - SSL - backpropagation - overfitting - filesystem - map - fold / reduce / join - monad - algorithmic complexity (space, time) - hashing - SHA - precision / recall - false positive / false negative - datacenter - cloud - scalability (horizontal, vertical) - leader election (algorithms) - strong consistency vs eventual consistency - write through (write to cache and DB at the same time) vs write ahead/back (write to cache, then whenever, update the DB asynchronously) - (cache) staleness - (cache) eviction policy (least recently used, least frequently used) - content distribution network, CDN (pull based etc) - terminal - devops - devsecops - mlops - virtual machine - docker - kubernetes - hex arch - opsec (make security section / S code ?) - osint - Apache (Kafka, Hadoop, Spark, Hive, Cassandra, Flink) - MapReduce - blue team / red team - assembly - page replacement algorithm (LRU) - publisher-subscriber architecture - lazy evaluation - (system) scaling (horizontal adding machines, vertical improving machine specs) - Storage Area Network, SAN - computer network types by scale (LAN, WAN etc) - tape silo / library / jukebox - compression - information (à la Shannon) - entropy

Add section on classical algorithms and algorithmic constructs, not just basic data structures (possibly drawing analogies between discrete and continuous versions where applicable). Improve the subdivision of sections on algorithmics. - min heap (binary tree; insert: push to bottom right, percolate upward; delete root: pick bottom right, min-heapify) - sorting algorithms (insertion, bubble, merge, quick, external) - BFS - DFS - Floyd-Warshall - graph matrices (adjacency, self-similarity, laplacian) - Fourier transform, FFT - Markov chains - Monte-carlo integration - quadtree / octree etc - compression algorithms

Increase the quality of explanation of algorithms and data structures with images and more details, all throughout the lexicon.

Add section on system engineering algorithms -¿ distributed hashing -¿ consistent hashing -¿ geohash -¿ Leaky bucket -¿ Trie -¿ Bloomfilter -¿ Raft/Paxos

Add section on data visualizations -¿ Histograms -¿ Bar Charts -¿ Line

Charts -¿ Scatter Plots -¿ Heatmaps -¿ Box Plots -¿ Choropleth Maps -¿ Tree Maps -¿ Sankey Diagrams -¿ Word Clouds -¿ Parallel Coordinates

Add section specific to dimensionality reduction - Advanced dimensionality reduction algorithms: -¿ Principal component analysis, PCA -¿ t-distributed stochastic neighbor embedding, t-SNE -¿ Uniform Manifold Approximation and Projection, UMAP -¿ Locally Linear Embedding, LLE -¿ Multi-dimensional scaling, MDS -¿ Isometric mapping, Isomap -¿ Non-negative Matrix Factorization, NMF -¿ RadViz - Neural nets for dimensionality reduction -¿ Autoencoders -¿ Self-Organizing Maps, SOMs

Add "Other: legal" section -¿ free software / freeware -¿ open source -¿ closed source -¿ licenses (MIT, Apache, etc) -¿ definition of laws like RGPD and what they entail

Add a section on cybersecurity

Add a section on computer graphics

Add a section on programming languages ? Improve section on programming paradigms ?

When this reaches a stable state (e.g; after the above TODOs are completed and a more thorough cybersec and CGI sections have been added), possibly transform this into a KV-store of some kind, add concept dependencies ("requires", "is necessary for", "should be learned in tandem with"), and make a script that can automatically generate this document, a wiki, and things like ANKI cards.