“Big Data Computing” Notes

Prof. Gabriele Tolomei. Notes written by Alessio Lucciola during the a.y. 2022/2023. These notes are incomplete and don’t cover the whole course.

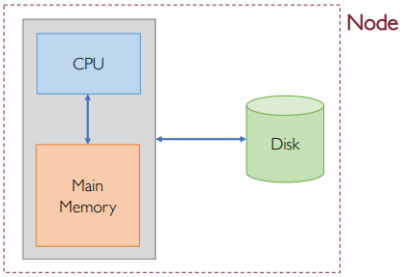
Notes may contain errors or typos. If you see one, you can contact me using the links in the Github page. If you find this helpful you might consider buying me a coffee 😄.

1. Introduction

Big Data is a phenomenon that can be described by some concepts that are:

- **Volume**: We usually work with a very large amount of data (orders of TB or PB);
- **Variety**: We usually deal with different formats of data such as structured (relational tables), semistructured (JSON files) and unstructured (text, audio, video);
- **Velocity**: Insane speed at which data is generated (e.g. Twitter stream);
- **Veracity**: Reliability of the data used to drive decision processes. We should be able to extract values from the data we get in order to make decisions out of it.

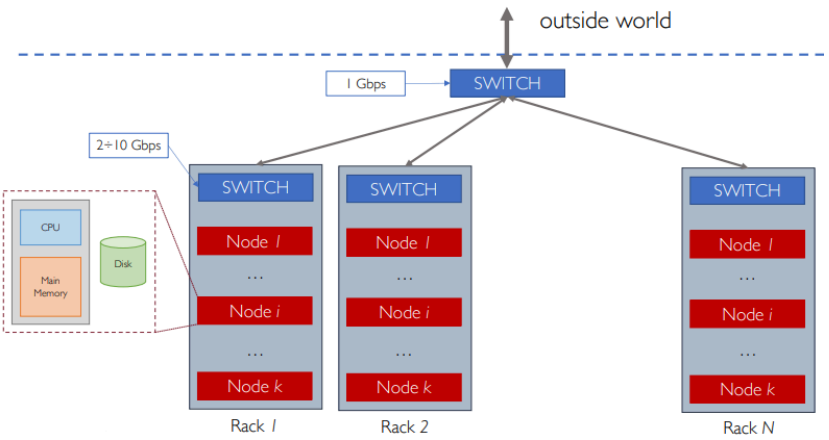
Once we get the data, it has to be properly stored, managed and analyzed in order to get the most value out of it.



First of all, let’s summarize the **architecture of a node** in order to understand how data storage works. The main components of a node are the CPU, the main memory and the disk. Everything is ok as long as data fits entirely into main memory and this is because the main memory is much faster with respect to the disk (few accesses to the disk are still tolerated). The transfer rate between the CPU and the main memory is about 25,600 MB/sec while the one with the disk ranges between 100-500 MB/sec (2 orders of magnitude difference between data transfer rate!).
In order to reduce the computational time needed to perform an operation with data we could exploit **scalability**, in particular:

- **Vertical scaling (scaling up)**: We have a disk, we buy a more performing one. It is an easy solution but improvement is physically limited (e.g. 2.5x or 5x) and it is quite expensive;
- **Horizontal scaling (scaling out)**: We buy a set of commodity "cheap" disks and let them work in parallel. This approach is flexible (improvement is not bound apriori, just add new disks as needed) but we need an extra overhead to manage parallel work.

The **Cluster Architecture** is based on the **scaling out principle**. There are a lot of **commodity nodes** communicating with each other and each group of 16/64 nodes is arranged in a so-called **rack**. A cluster is made of multiple **racks**. In order to allow the nodes to communicate, there are **switches** that can be **inter-rack** (1 Gbps) or **intra-rack** (2/10 Gbps). In the cluster architecture there are **3 major challenges** that deals with:



- **Node failures**: We would like to ensure **reliability** upon a node failure. Suppose we have a cluster of N nodes and each node has a

MeanTime To Failure (MTTF) = 3 years ~ 1,000 days. This means the probability for a node to fail is:

$$p = P(node_i \text{ fails}) = 1/1{,}000 = 0.001$$

We can associate with each node a random variable $X_{i,t}$ which is Bernoulli distributed. $X_{i,t}$ ~ Bernoulli(p) outputs 1 (failure) with probability p = 0.001 and 0 (working) with probability (1-p) = 0.999. We assume for simplicity p is the same for all nodes and independent from each other. Under the (simplified) assumption that $X_{i,t}$ are all i.i.d. then the number of failures in a certain day t (given that the probability of one machine failing is p) is just the summation of all the variables $X_{i,t}$ (remember that the output of a variable can be 0 or 1). This means that the expected value of T is n times p.

$$T = X_{1,t} + X_{2,t} + \ldots + X_{N,t} \qquad T \sim Binomial\ (N, p) \qquad E[T] = Np$$

A single-node failure on a day may be quite a rare event (0.1% chance) but, for example, if we have 1 million nodes then the (expected) failures per day becomes 1k.

- **Network bottleneck**: We would like to **minimize network communication bottleneck**. Moving data across nodes both intra- and inter racks may be costly;
- **Distributed programming**: Distributed programming can be really complex. Programmers should focus on the (distributed) task rather than dealing with the complexities of the cluster architecture.

SUM. Data is generated at an unprecedented rate (that's why we talk about big data). Extracting knowledge from such big data is incredibly valuable. Traditional algorithms/techniques often don't scale very well. There is the need for new "tools" which allow storing, managing, and analyzing big data painlessly.

2. Map Reduce

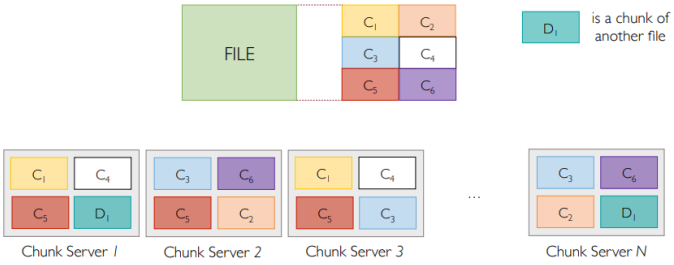
Map Reduce is a **programming model** for processing big data sets with parallel, distributed algorithms on a cluster. It addresses the 3 main challenges of cluster architecture described in fact:

- It stores data redundantly on multiple nodes to ensure data/computation availability;
- It moves computation close to data to minimize network data transfers;
- It provides a simple computational model to hide all the complexities of the distributed environment.

Some examples of Map Reduce implementation are the **Google File System**, **Hadoop File System (HDFS)**, **Flink** and **Spark**.

In order to solve the node failure challenge, the **distributed file system** has a **reductant storage infrastructure** that provides global file namespace and availability across different nodes in a cluster. This infrastructure is mainly used when we have to deal with **large files** (100 GB/10 TB) and we have **many "read" operations** and **few "updates" (append)**. To build an infrastructure like this, there are 3 main components:

- **Chunk servers**: Large data files are split into contiguous "**chunks**" of fixed size (e.g. 16/64 MB). Each chunk is **replicated** across multiple nodes (chunk servers). There are usually 2 or 3 replicas per chunk and each replica is on a different node. At least, one replica is on a different rack (useful if the rack fails). Chunk servers act also as **computational servers** so they can make computation on the data;



- **Master nodes**: It stores **metadata** about files in the distributed file system in particular, the information how many chunks each file is split into and where each of those chunks are located. It is possibly **replicated** to **avoid single-point of failure**;
- **Client API**: It allows clients to **access data** stored on chunk servers. The client asks the Master Node through the API where a particular chunk is located and it replies with the information needed. Afterwards, any communication between the client and the chunk server storing the data happens directly (i.e., without the Master Node).

MapReduce is a **style of programming** designed for: Easy parallel programming, invisible management of hardware and software failures, easy management of very-large-scale data.

Map Reduce is great when used with problems that require a **lot of sequential data access** (from dick) and we have **large batch jobs** (i.e., scheduled jobs that don't require an interaction from the user). Contrarily, it is not suitable for problems that require **random access to data** (because we usually divide the document into different chunks possibly stored in different nodes), when **working with graphs** and with **interdependent data** (data in multiple tables are dependent on each other).

Let's assume we have to count the words in the document. The result of the task will be a list of (word, count) pairs. **If the total number of (word, count) pairs fit into main memory** we don't have particular problems because we can simply initialize an empty hash map/table, process one line at a time, extract each individual word from a line and update the hash map, add the current word in the hash table (with value one) if it's the first time we see it or increase the value of the word by one if we have already seen it (it's already in the hash table).
Let's suppose now we have a really large document (e.g. 10s of TB) that clearly **doesn't fit into the main memory**. We can use **Map Reduce** to make this operation possible.
The input of this approach is a set of (key, value) pairs as well as the output. Map reduce defines two methods which are **Map** and **Reduce**. We also have an intermediate **Shuffle** step which is provided by the framework.
Let's now define the steps more formally. Let's assume we have a set of input key-value pairs $\{(k_1, v_1), (k_2, v_2), \ldots, (k_M, v_M)\}$. The **Map function** takes an input key-value pair and outputs a set of 0 or more new, intermediate key-value pairs. We usually have one map function call for each input key-value pair (k_i , v_i). We talk about "**map task**" as multiple map calls executed in parallel on a subset of the input key-value pairs.
The **Reduce function** reduces together all values v_i' associated with the same key k_i'. One reduce function call is usually performed for each unique key k_i'.

$$\mathbf{map}(k_i, v_i) \rightarrow \{(k_i', v_i')\}^*$$

$$\mathbf{reduce}(k_i', \{v_i'\}^*) \rightarrow \{(k_i', v_i'')\}^*$$

```
map(key, value):
# key: docID; value: text
    foreach word in value:
        emit(word, 1)
```

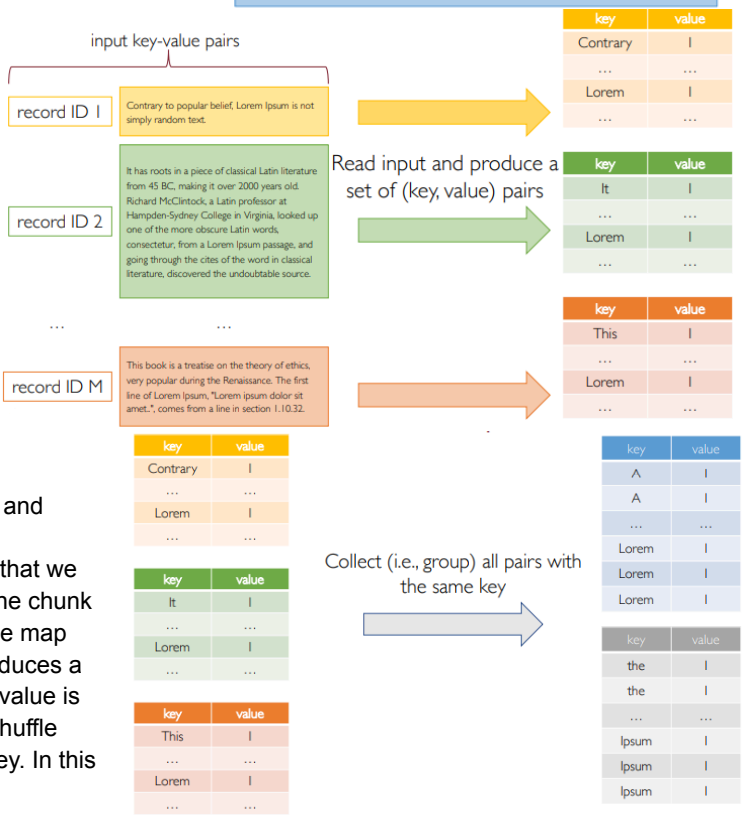
```
reduce(key, values):
# key: word; values: iterator
    result = 0
    foreach v in values:
        result += v
    emit(key, result)
```

Note: Remember that the Kleene (star) operation can be described as the set containing the empty string and all finite-length strings that can be generated by concatenating arbitrary elements of the initial set, allowing the use of the same element multiple times.

EX. There exists a UNIX command that perfectly describes the Map Reduce operation:

print_words(doc.txt) | *sort* | *uniq -c*

Let's assume we apply the word counting operation on a document "doc.txt". The map function basically corresponds to *print_words*: it takes as input the original data (e.g., a chunk of the whole doc.txt file) and produces as output something out of the data called **intermediate keys** (e.g., a word for each line in the chunk).
Then we have the *sort* operation: The intermediate keys generated by the map function are sorted and shuffled. Note that **intermediate keys are not unique** (it may happen that the same word appears at different times in the document). Finally we have the *uniq -c* that resembles the role of the reduce function: it takes as input the groups of intermediate keys, it computes an aggregating/filtering/transforming function over those keys and outputs the result.
So, let's assume we divide the file into different chunks so that we have (key, value) pairs in input where the key is the ID of the chunk and the value is the part of the document. We can apply the map task operation for all elements (chunk of the file) and it produces a set of (key, value) pairs where the key is the word and the value is always 1 (there might be duplicates!). Then we apply the shuffle step where we collect (i.e. group) all pairs with the same key. In this

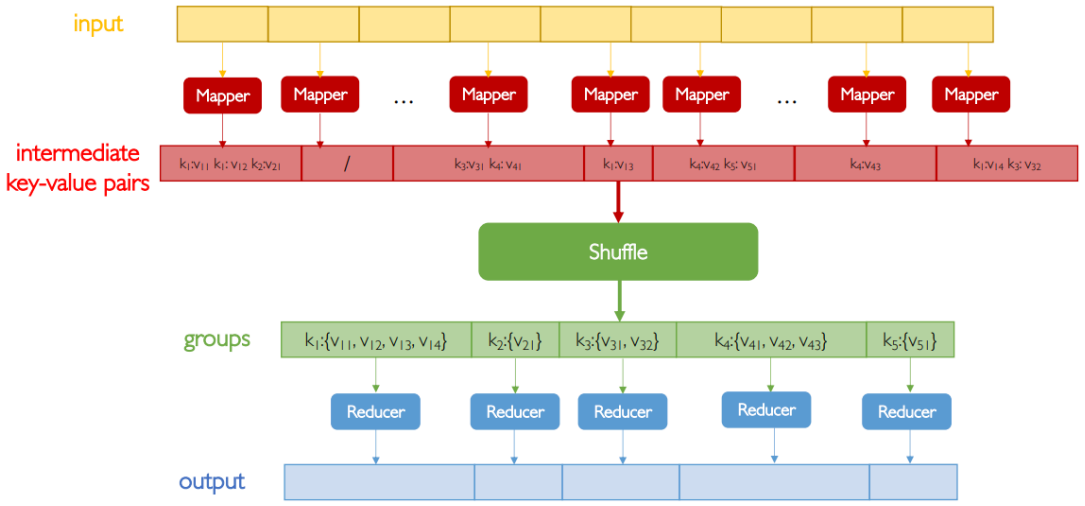


step it is necessary that all the intermediate keys of one specific intermediate key are put in the same partition (it doesn't matter if the words are sorted, what is important is that they are all in the same partition so that we avoid communication with different nodes). Finally we apply the reduce step in which we process all values belonging to a given key and output the result.

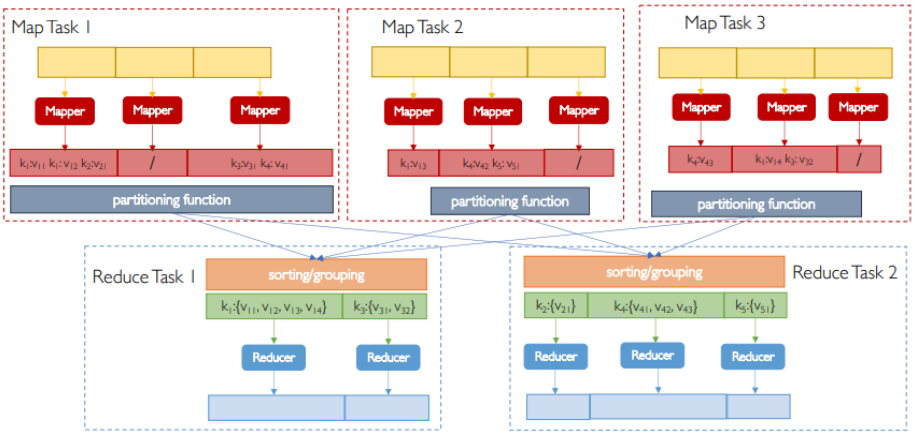
Now, let's see what happens with **Map Reduce on a Single Node**. Let's assume we have a certain large input file in which we have physical blocks (blocks) and a logical split (InputSplit). If we don't do anything, as programmers, there is an equivalence between a physical block and a logical split (one physical block corresponds to one logical split). The number of logical splits determines the number of map tasks (we have a map task for each physical block if we don't do anything).
Block is a continuous location on the hard drive where data is stored. In general, FileSystem stores data as a collection of blocks. In the same way, HDFS stores each file as blocks. The Hadoop application is responsible for distributing the data block across multiple nodes. The data to be processed by an individual Mapper is represented by InputSplit. The split is divided into records and each record (which is a key-value pair) is processed by the map. The number of map tasks is equal to the number of InputSplits. Initially, the data for MapReduce task is stored in input files and input files typically reside in HDFS. InputFormat is used to define how these input files are split and read. InputFormat is responsible for creating InputSplit. The difference between the two are as follows:

- Block: The default size of the HDFS block is 128 MB which we can configure as per our requirement. All blocks of the file are of the same size except the last block, which can be of the same size or smaller. The files are split into 128 MB blocks and then stored into Hadoop FileSystem. It is the physical representation of data. It contains a minimum amount of data that can be read or write;
- Input Split: By default, split size is approximately equal to block size. InputSplit is user defined and the user can control split size based on the size of data in MapReduce program. It is the logical representation of data present in the block. It is used during data processing in the MapReduce program or other processing techniques. InputSplit doesn't contain actual data, but a reference to the data, in particular it includes the location for the next block and the byte offset of the data needed to complete the block.

We have several mappers that make a map call, the shuffle operation that groups together intermediate keys and finally several reducers applied to each shuffled group that computes the output.



In the case of **Map Reduce on a Cluster**, we could have different partitions based on where the data is stored:



Now, remember that we **only need to specify the map and reduce functions**. Everything else is managed by the infrastructure such as the input data partitioning (physical = chunk/block and logical = split), the scheduling of tasks across nodes of the cluster, the shuffling/group by of intermediate keys output by mappers, the handling of node failures and the management of inter-node communications.

For what concerns the **Data Flow**, both input and output are stored on the distributed file system. Map Reduce scheduler tries to allocate map tasks "close" to data (the master node knows how data is distributed and uses this information to optimize the workload distribution). **Each map task running on a node will be using the chunks of data that are stored on that node** (chunk server). **Intermediate results of map/reduce tasks are stored on the local filesystem of each node** and this is done to avoid copies/replicas of useless files across the cluster (DFS).
Basically, the **Master Node** takes care of node coordination/orchestration, in particular the status associated with each task (either map or reduce) that can be **idle**, **in-progress** or **completed**. Idle tasks are eligible to be executed as soon as a worker node becomes available. When a map task completes it sends notification of that to the master node who propagates that information to the reducers. The master node periodically pings mappers/reducers to detect failures. In the case of a **failure**:

- When a **map worker fails**: All the map tasks completed or in-progress at the (failed) worker node are reset to idle. Idle map tasks will be eventually rescheduled later on other worker node(s);
- When a **reduce worker fails**: Only in-progress tasks are reset to idle (completed ones have already output to the DFS). Idle reduce tasks will be eventually rescheduled later on other worker node(s);
- When the **master node fails**: The whole Map Reduce job is aborted.

How many Map/Reduce tasks do we need?
Let's assume N is the nodes of the cluster, M are the map tasks and R are the reduced tasks. The rule of thumb is:

- M >> N (in fact, one map task per DFS chunk is pretty common);
- Having M >> N speeds up recovery from node failures;
- R < M (convenient to have the output spread across a limited number of nodes).

This is usually transparent to the programmer.

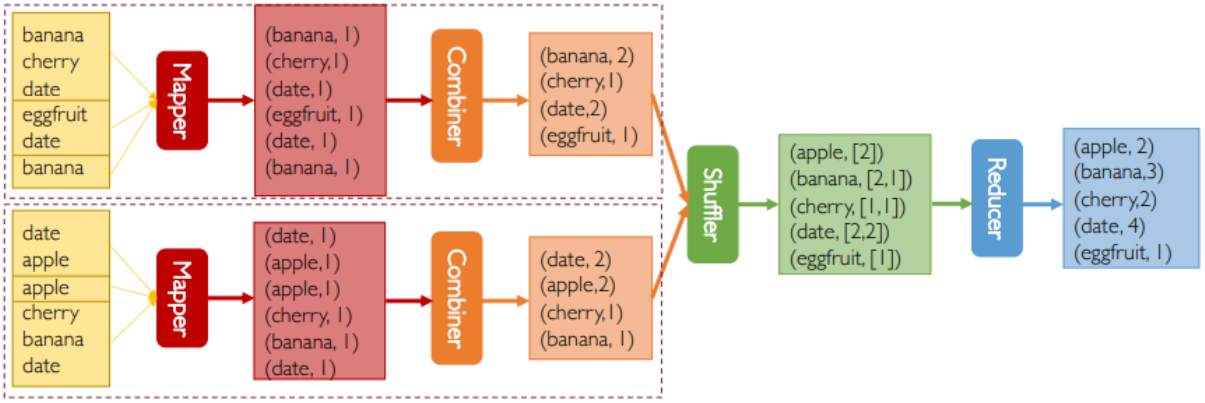
EX. Another MapReduce task (Natural Join) → Slide.

We saw before that a map task may produce many pairs of the form (k, v_1), (k, v_2), etc.. all sharing the same key k. Can we do better?
EX. Consider again the word counting task. A word w may appear several times in the input chunk associated with a mapper. Still, the mapper will output the same key-value pair (w, 1) every time it will find an occurrence of w. Eventually, all these (same) key-value pairs must be transferred to a reducer.
We can do better with a **Combiner**: Combiners can save network transfers by pre-aggregating values at the mapper's end. Usually, the combiner computes the same aggregating function of the reducer.

$$\texttt{combine}(k, \{v_1, v_2, \ldots, v_m\}) \rightarrow (k, v')$$

where v' is the result of an aggregating function computed on $\{v_1, \ldots, v_m\}$.

EX. An example using combiners:



Combiner combines values associated with the same key yet coming from a single mapper (i.e. 1 mapper: 1 combiner). The problem with combiners is that it can be only used when the **reduce function is commutative and associative** (e.g. sum, product). Sometimes workarounds exist to take benefit from combiners even if the reduce function is not commutative and associative.

EX. Average operation: Instead of letting each combiner output the local average from its own input data, make the combiner output the pair $(k_i, (sum_i, count_i))$ where sum_i is the sum of the values associated with the key k_i and $count_i$ is the total number of values with that key k_i.In this way, the reducer can compute the average associated with the key k_i by simply doing $[(sum_i)_1 + .. + (sum_i)_m]/[(count_i)_1 + .. + (count_i)_m]$.

The combiner trick seen before is not applicable to every function. It works only for those functions which can be expressed as the composition of commutative and associative operators (e.g. the trick can't be used on the median operation).

The **Partition function** controls how intermediate key-value pairs produced by mappers are distributed across (i.e., sent over) reducers. Assuming R reducer nodes, default partition function is:

hash(key) mod R

Sometimes it may be useful to override the default partition function with a custom one.

Two **major limitations of MapReduce** paradigm:

- Hard to program directly: Many problems are not easily described as map-reduce;
- I/O communication bottlenecks cause performance issues: Persistence to disk slower than in-memory computation.

In short, MapReduce is not suitable for large applications composed of several map-reduce steps.

3. Spark

In the previous section, MapReduce was introduced: It is a distributed framework suitable for working with large scale datasets that is useful when data need to be accessed sequentially. MapReduce uses 2 ranks of tasks that is the map and the reduce where data flows from the first rank to the second one. The existence of these intermediate steps is useful because if failures happen, it is possible to recover from a certain point in the computation instead of starting again from scratch (so it is possible to recover faster from failures).
It is rather convenient to have a more abstract data flow system that allows any number of "ranks"/tasks and allows any functions other than just map and reduce. Moreover, as long as data goes in one direction only, assuming to have multiple ranks, it is still possible to recover at an intermediate rank. That's where Spark comes in. In addiction to MapReduce, **Spark** provides:

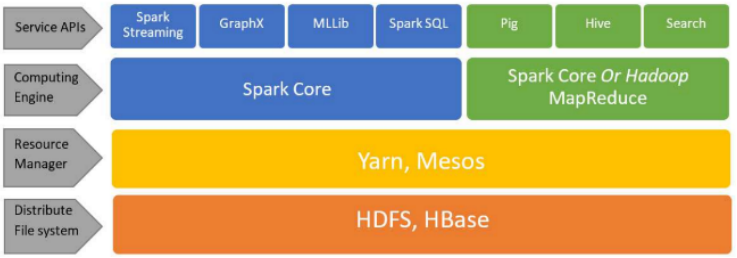
- **Fast data sharing**: There are no intermediate saving to local disks so the main memory is extensively used and this is done to make the computation much faster (remember that writing and reading the data from the disk is a much slower operation);
- **General execution graphs (DAGs)**: The steps performed for processing data can be seen as abstract steps that aren't really applied on the data until a certain action is done: this is called **lazy evaluation.** All these steps that have to be done can be represented by a graphs;

- Richer functions than just map and reduce.

Spark is formed by a unified **computing engine** which is called **Spark Core** and a set of high-level APIs that can be used for data analysis that are Spark SQL (used to handle structured data), MLlib (used for machine learning), GraphX (for graph analytics) and Spark Streaming (for stream data processing).
Unlike Hadoop, Spark does not come with a storage system in fact it provides interfaces for many local and distributed storage systems such as HDFS, Amazon S3 or classical RDBMS (this is done in order to make Spark run with different storage systems and increase flexibility). Additionally, Spark's APIs are available for many programming languages such as Scala, Java, Python, and R.

The **main features of Spark** are:

- **Fault-tolerant system**: Nodes must be able to properly cope with failures;
- **In-memory caching** which enables efficient execution of multi-round algorithms (i.e. multiple sequential tasks), so each task is executed in the main memory and this leads to an increase in performance with respect to Hadoop.

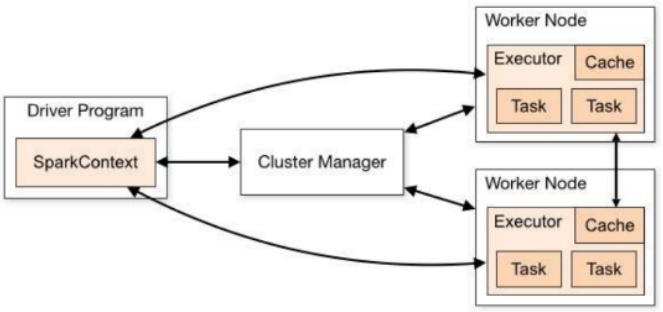


Moreover, Spark can run:

- On a single machine a local mode: Both on the local machine or using services such as Google Colab that provides a sort of isolation through the cloud (it is like we use a single machine though the cloud);
- On a cluster managed by a **cluster manager** (e.g., Spark Standalone, YARN, Mesos).

The Spark application is composed by different elements that are:

- The **driver process** (the **master** in MapReduce terminology) runs the application's entry point from a node in the cluster that is responsible for maintaining information about the application (the metadata information about the file in the distributed file system), responding to a user program or input and analyzing, distributing, and scheduling work across executors. The driver is represented by an object called **Spark Context**;
- **Executor processes** (the **workers** in Hadoop terminology) actually compute the tasks assigned by the driver. Each executor is responsible for: Running the code assigned to it by the driver and reporting the state of the computation back to the driver;
- The **cluster manager** controls physical machines and allocates resources to applications.



So, we have the driver process (that works like a JVM process) which is the entry process of an application, and several worker nodes in which there might be one or more executors. The executors can be seen as different JVM processes with different tasks that perform some activity on a partition of the data (so they are different threads). The cache is the main memory associated with the worker node that is usually divided between the executors in each node. The caches of different workers can communicate with each other in order to conclude a certain task. Notice that in case of **Spark running on a single machine**, both the **driver program and the executor** (only one in this case) are **on the same node**. If there are multiple cores, we may have different tasks within the same executor (so in the same JVM we could have **several separated threads**: one for the driver, one for the executor and others for the tasks). Executors usually run Scala code and the driver can be governed using several APIs that allow the user to use other programming languages such as Python.

Resource Allocation is an important aspect during the execution of any spark job. If not configured correctly, a spark job can consume entire cluster resources and make other applications starve for resources. In general we can use a:

- **Static allocation**: When we launch the installation of Spark, we can decide to manually configure some parameters such as the number of executor cores, the number of executors and their memory. Let's

assume we have 10 nodes, 16 cores per node and 64GB per node. There are some possible configurations such as:
    - **Tiny Executors**: In this case we use **one executor per core**. This means that we have a total of 16 cores*10 nodes = 160 executors with just one core (one thread). The memory for each node is 64GB/16 cores = 4GB. With only one executor per core we'll not be able to take advantage of running multiple tasks in the same JVM since we can only do one task in parallel. Also, we are not leaving enough memory overhead for the cluster manager and other components.
    - **Fat Executors**: In this case we use **one executor per node**. This means we have 10 executors with all 16 cores per executor and each of them has 64GB of memory. We have a degree of parallelism of 16 (so we can execute 16 tasks in parallel). In this case the problem is that we don't have space for additional threads we need for components such as the cluster manager.
    - **A balance between the two**: A way better solution because it allows us to achieve parallelism of a fat executor and best throughputs of a tiny executor. There may be different configuration and here one of idea is presented:
      EX. Assuming to have 10 nodes, 16 cores per Node and 64GB RAM per node. Let's assign 5 cores per executors (we have 5 tasks running in parallel on a given node) and leave one core per node for the daemons (e.g. cluster manager) so the number of cores available per node is 16-1 = 15 and the total available of cores in cluster are 15*10 = 150. Remembering that we assign 5 cores per executors then the total number of executors is 150/5 = 30, per node is 30/10 = 3. The memory per executor is 64GB/3 = 21GB. Assuming that we leave one executor as the application manager, we count off heap overhead that is 7% of 21 GB = 3GB. So, the actual executor memory is 21-3 = 18GB.[1]
- **Dynamic allocation**: Everything is managed by the spark application so the installation and management of the application is easier but, of course, it may have a large overhead.

The **main abstraction** that Spark uses to represent data is called **Resilient Distributed Dataset (RDD)** that is used to indicate a collection of elements of the same type, where resilient refers to failures. So, it is a generalization of MapReduce's key-value pairs. **RDDs can be partitioned** and **possibly split across multiple nodes of the cluster**. In particular, each RDD is split into chunks called **partitions** distributed across nodes and determines the workload across the nodes (together with the parameters seen before). A program can specify the number of partitions for an RDD (otherwise Spark will choose one according to the setup we have). Programmers can also decide whether to use the default **Hash Partitioner** or a custom one that controls how the partition is done. A typical number of partitions is 2 or 3 times the number of cores. Partitioning enables the following:

- **Data reuse**: The data is kept in executors' main memory so as to avoid expensive access to external disks;
- **Parallelism**: Some data transformations are applied independently to each partition thereby avoiding expensive data transfers (we may apply some transformation to the data independently from the partition).

RDDs are **immutable** (so they are read-only) and can be created either from data stored on a distributed file system (e.g. HDFS) or as a result of transformations of other RDDs. The idea is that tasks apply some transformations that possibly transform an RDD into another RDD (like in a pipeline). At each step, we don't have to materialize the RDD (at an intermediate step) since each **RDD maintains a sort of "trace" of transformations** (lineage) that led to the current status. So we separate the transformations that we actually do on the RDDs from the actual materializations of the actions and, in this way, **RDD can always be re-created even upon a failure**. If Spark could wait until an Action is called, it may merge some transformations or skip some unnecessary transformations and prepare a perfect execution plan. Let A be an RDD, the following 3 operations are possible:

- **Transformations**: Generate a new RDD B from the data in A. When applying a transformation, we perform a certain task on the data but it isn't physically applied until an action is performed (this is called **lazy evaluation**). We can distinguish between different transformations that can be:
    - **Narrow**: Each partition of A contributes at most to one partition of B (e.g. map). In this case there is no need to shuffle data across nodes and Input and output stay on the same partition.

[1] More info: http://beginnershadoop.com/2019/09/30/distribution-of-executors-cores-and-memory-for-a-spark-application/

    - **Wide**: Each partition of A may contribute to multiple partitions of B (e.g. groupBy). In this case there is the possibility to transfer data across nodes, so input from other partitions may be needed;
- **Actions**: Launch a computation on the data in A, which returns a value to the application. It is what physically triggers a computation on the data so they materialize the transformations applied to the data. An example of action is the count method that returns the number of elements of the RDD.
- **Persistence**: Save the RDD in memory for later actions.

RDDs are the most basic data model used by Spark. On top of RDD API, Spark SQL module provides 2 interfaces to operate on structured data like tables in relational databases: **DataFrame API** and **Dataset API**. DataFrame is a distributed collection of data organized into named columns. It allows higher level abstraction than plain vanilla RDDs and it is really similar to Pandas DataFrame.
Spark DataFrames are immutable: once created **they cannot be modified**. As for RDDs, Spark may apply 2 kinds of operations on DataFrames which are transformations and actions. Lazy evaluation allows queuing transformations applied to elements of a DataFrame until an action is called. DataFrame (and Dataset as well) can be turned back to RDD.

Some **differences between Spark and Hadoop** are:

- **Performance**: Spark is usually faster. Spark uses in-memory data processing while Hadoop uses data persistence to disk after any map/reduce step. Spark requires lots of memory to run fast, otherwise its performance deteriorates. MapReduce integrates better with other services running;
- **Ease of use**: Spark provides a higher-level API which is easier to program;
- **Data processing**: Spark is more flexible and general.

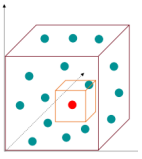
4. Curse of Dimensionality (Clustering)

Clustering is an **unsupervised learning** technique to group a set of objects into classes of **similar objects**, so the data is not labeled. It can be also used as a method of **data exploration** in which we can look for patterns of interest in data. If we assume a set of points, clustering is about understanding their “structure” in order to find groups of data and insert each data into one of the found groups.
EX. Some examples of problems that recall clustering are categorizing documents on the same topic, grouping customers by their behaviors and so on and so forth.

More formally, given that set of points we have to define the notion of distance between those and be able to group them depending on how members of a cluster are close/similar to each other (in this case we talk about **high intra-cluster similarity**) and members of different clusters are dissimilar to each other (in this case is a **low inter-cluster similarity**). Clustering is not and easy task in fact there are many practical issues such as:

- **Object representations**: Data can be represented in different ways and each point can have a lot of features. This can lead to having **very high-dimensional spaces** that makes grouping difficult;
- The ways to measure the similarity between objects has to defined: this can be done with a proper **distance measure**;
- The **number of output clusters** has to be defined: it could be data-driven or fixed apriori.

Data points are not always easily and clearly separable and finding a clear boundary between clusters may be hard in the real world. If we have a small number of data points, clustering is not that hard but the problems arise when we have a **lot (hundreds or thousands) of dimensions**, in fact in high-dimensional spaces almost all pairs of points are at the same (large) distance. In high-dimensional space, the data tends to be **sparser** (**more dissimilar** to each other) than in lower dimensions. In Euclidean space, the distance between two points is large as long as they are far apart along at least one dimension. So, the higher the number of dimensions the higher the chance this happens: this is called **Curse of Dimensionality**.



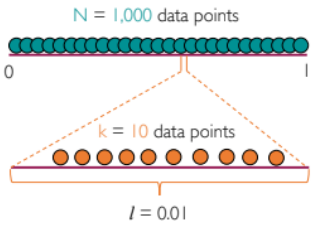
Now, let’s assume we have a R^d d-dimensional cube. In order to introduce the topic, we can choose d=3 so that it is possible to visualize what happens. We have N data points that are randomly (i.e. uniformly) distributed in H. Let’s take a random point p among N. We can define as l, the edge size of the hypercube h in H that contains p and k nearest points (with regards to p). We can consider **edge points** the ones whose distance from p is at most (l/2)*√d (remember that in the example d=3). So, we want to find out how small the hypercube h should be in order to contain the k nearest points.

$$l \approx \left(\frac{10}{1000}\right)^{1/d} = \left(\frac{1}{100}\right)^{1/d}$$

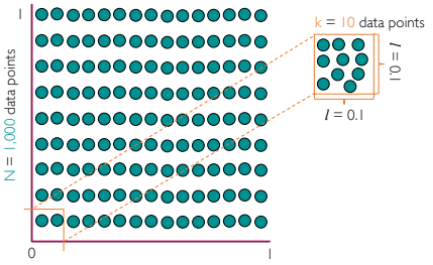
Given V_h the volume of the hypercube h then $V_h = l^d$, and Vh must contain roughly k/N points (since they are randomly distributed), this means that:

$$l^d \approx \frac{k}{N} \text{ therefore } l \approx \left(\frac{k}{N}\right)^{1/d}$$

d	l
1	0.01
2	0.1
3	0.215
…	…
10	0.631
…	…
1000	0.995



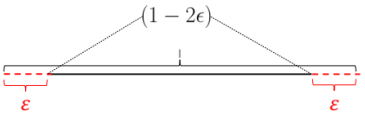
Let's try to understand how l should be if we plug in some numbers. If we have N = 1000 and k = 10. We pick one of the N numbers randomly and we want to understand the size of l depending on how we vary the dimension of the space d. If d=1 then we have a hypercube that is no more than a line, so the size of l to contain k=10 data points is l=0.01. If d=2 then we have a rectangle where the data is sparser, so we intuitively need a larger hypercube to contain the k=10 data points so here l=0.1. If d=10 then l=0.631 that means the length of the edge of the inscribed hypercube is already about 63% of the largest hypercube. If d=1000 then l=0.995 so it is like there are no differences between the two hypercubes. The data points get far away from each other as the dimension increases, so the notion of closeness gets invaluable.



We must care about the problem of the curse of dimensionality because points are more likely to be located at the edges of the region (so they are sparser at the edge of the region) and the points we consider to be near to each other are not close at all. Distance between points indistinguishable (**distance concentration**) so:

- It is hard to separate between nearest and furthest data points;
- It is hard to find clusters among so many pairs that are all at approximately the same distance.

Let's now go more into details and define what an edge is. Let **ε** define the **edge** (i.e., border) of our space, we can see how the probability of picking a data point that is not located at the edge changes as the number of dimensions grows. So we start from the 1-d case and we see what is the probability of picking a point among the ones that are randomly (uniformly) distributed on the space that is at the edge, then we increase the dimensionality and see what happens. If d=1 then the hypercube is just a **unit length line** (as already seen before) so assuming that ε is the edge then the probability of being not at the edge is just **(1-2ε)**. If we **increase the dimensionality** (so assuming that d>1), the probability of being not at the edge is the probability of being not at the edge on every single dimension so:

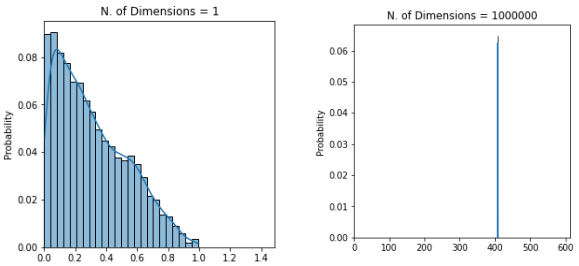


$(1-2\epsilon)^d$ ← Assuming each dimension is independent from each other (so it like we are computing the product of the probability computed for each dimension).

$$\lim_{d\to\infty}(1-2\epsilon)^d = 0$$

As d grows larger and larger then the probability tends to 0 (the base is 0＜1-2ε <1 and the exponent d≥1).

Note: It is possible to see a visual representation of the curse of dimensionality problem here. In general it is possible to see that as the dimension grows larger and larger, the distance between the data points is higher and the concentration around the value of the distance is concentrated (the notion of distance is not important anymore).



So, if the data is uniformly distributed in a high-dimensional space we can do nothing about it but in the real world **they have data patterns underneath** (so they are **not random**). What usually happens is that if we have a high dimensional data (e.g. images), it usually lies in a low-dimensional subspace embedded in the high-dimensional space and there are some techniques to perform some **dimensionality reduction** to reduce the features with which we represent the data: what said so far is also called the **Manifold Hypothesis**.

Now, we have to understand **how to measure the similarity of different objects**. First of all, let's introduce what **metric space** is. Let's assume X is a set and **δ** is a function: **δ**: X x X → [0, ∞), where:

$1.\delta(x,y) \geq 0$ $(\textbf{non-negativity})$
$2.\delta(x,y) = 0 \Leftrightarrow x = y$ $(\textbf{identity}$ of indiscernibles$)$
$3.\delta(x,y) = \delta(y,x)$ $(\textbf{symmetry})$
$4.\delta(x,y) \leq \delta(x,z)+\delta(z,y)$ $(\textbf{triangle inequality})$

Then **δ** is a metric (or distance function) and X is a metric space. There are some distance function we can use to understand what is the distance (dually, the similarity) between different objects:

- **Euclidean Distance**: The set X = R^d (X is the d-dimensional space) and **δ**: R^d x R^d → [0, ∞) (delta takes in input 2 dimensional vectors R^d so two points in R^d and gives back a real non-negative value) and returns the square root of the sum of squared differences of each components:

$$\delta(\mathbf{x},\mathbf{y}) = \sqrt{(x_1-y_1)^2+\ldots+(x_d-y_d)^2} = \sqrt{\sum_{i=1}^{d}(x_i-y_i)^2}$$

The position of a point in a Euclidean d-space is a **Euclidean vector** while the **Euclidean norm** of a vector measures its length (from the origin) and it can be just seen as the Euclidean distance between vector's tail and tip (notice that · is the dot product):

$$||\mathbf{x}||_2 = \sqrt{x_1^2+\ldots+x_d^2} = \sqrt{\mathbf{x}\cdot\mathbf{x}} \qquad \mathbf{x}-\mathbf{y} = (x_1-y_1,\ldots,x_d-y_d)$$

Given x-y the displacement vector between x and y (the vector obtained by making the component-wise difference of x and y) then the Euclidean distance between x and y is just the Euclidean norm of the displacement vector:

$$\delta(\mathbf{x},\mathbf{y}) = ||\mathbf{x}-\mathbf{y}||_2 = \sqrt{(\mathbf{x}-\mathbf{y})\cdot(\mathbf{x}-\mathbf{y})}$$

Let's see what happens with different dimensions. In the **1-dimensional case** (d=1) then x and y are both scalars so the Euclidean distance between any two 1-d points on the real line is the **absolute value** of the numerical difference of their coordinates:

$$\delta(\mathbf{x},\mathbf{y}) = \delta(x,y) = \sqrt{(x-y)^2} = |x-y|$$

In the **2-dimensional case** (d=2) so when x, y ∈ R^2 then x=(x_1, x_2), y=(x_1, x_2) then the Euclidean distance between any two 2-d points on the Euclidean plane equals to the **Pythagorean theorem**:

$$\delta(\mathbf{x},\mathbf{y}) = \sqrt{(x_1-y_1)^2+(x_2-y_2)^2} = ||\mathbf{x}-\mathbf{y}||_2$$

The generalization of the Euclidean distance is called **Minkowski Distance** (**L^p-Norm**). Assuming to have x, y ∈ R^d then x=(x_1, .., x_d), y=(x_1, .., x_d), the distance is computed like the sum along all the dimensions of the absolute component-wise difference raised the power of p, all raised to the power of 1/p:

$$\delta_p(\mathbf{x},\mathbf{y}) = \left(\sum_{i=1}^{d}|x_i-y_i|^p\right)^{\frac{1}{p}}$$

Plugging in different values of p we get the well-known formulas we already seen before, in particular:

$$\delta_1(\mathbf{x},\mathbf{y}) = \left(\sum_{i=1}^{d}|x_i-y_i|^1\right)^{\frac{1}{1}} = \sum_{i=1}^{d}|x_i-y_i|$$

← **L^1-Norm** or **Manhattan Distance**

$$\delta_2(\mathbf{x},\mathbf{y}) = \left(\sum_{i=1}^{d}|x_i-y_i|^2\right)^{\frac{1}{2}} = \sqrt{\sum_{i=1}^{d}|x_i-y_i|^2}$$

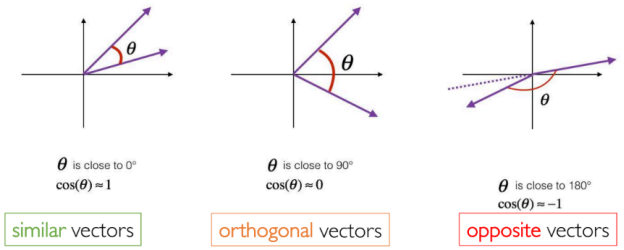
← **L^2-Norm** or **Euclidean Distance**

$$\delta_\infty(\mathbf{x},\mathbf{y}) = \lim_{p\to\infty}\left(\sum_{i=1}^{d}|x_i-y_i|^p\right)^{\frac{1}{p}} =$$

$$= \max\{|x_1-y_1|,|x_2-y_2|,\ldots,|x_d-y_d|\}$$

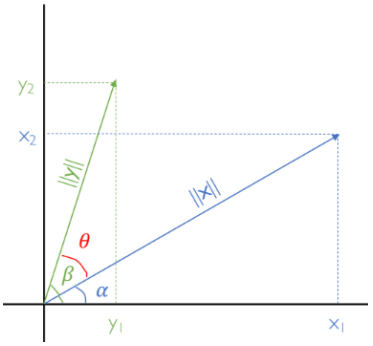
← **L^∞-Norm** or **Chebyshev Distance**

- **Cosine Similarity**: If we don't care about the magnitude of the vector involved in the distance computation but we care about the **orientation** (so we want to know if they are aligned), we can use the **cosine similarity** that is a measure of similarity between two non-zero vectors of an inner product space. It measures the cosine of the angle between vectors and it ranges between [-1,1]. There might be several cases →



We can get the similarity between two vectors by computing the angle Θ. Now, let's assume we have 2 vectors x and y, and their length is represented by their magnitude ||x|| and ||y||. We vertices are respectively x_1, x_2 and y_1, y_2. Since they are oriented, we can define α as the angle between the vector x and the x-axis, β as the angle between the vector y and the y-axis. The angle between the vectors x and y is given by:

$$\theta = \beta - \alpha$$



- The coordinates of x and y are computed like:

$$x = (\underbrace{||x||\cos\alpha}_{x_1}, \underbrace{||x||\sin\alpha}_{x_2}) \qquad y = (\underbrace{||y||\cos\beta}_{y_1}, \underbrace{||y||\sin\beta}_{y_2})$$

The dot product between the two vector is defined as the component-wise product between the two:

$$x \cdot y = x_1 y_1 + x_2 y_2 = ||x||\cos\alpha||y||\cos\beta + ||x||\sin\alpha||y||\sin\beta$$

$$= ||x||||y||(\cos\alpha\cos\beta + \sin\alpha\sin\beta) = ||x||||y||\cos\theta$$

If we normalize the vector x and y to have a unit-length norm, then the cosine product simply becomes the dot product of the normalized vector:

$$\cos\theta = x \cdot y / ||x||||y||$$

What we have seen so far, it is a **2-dimensional case**. The d-dimensional case is computed as in the case of 2-dimensional vectors. If two d-dimensional vectors are not collinear then they span a 2-dimensional plane $E \subset R^d$. This plane E inherits the dot product in R^d and so becomes an ordinary Euclidean plane. The angles in this plane are related to the dot product as they are in 2-dimensional vector geometry.

- **Jaccard Index (Coefficient)**: It is used to measure the **similarity between finite sample sets**. Whenever we define the data points to be sets then we can use the Jaccard Index to measure their similarity. It is computed like:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|} \qquad 0 \leq J(A,B) \leq 1$$

It ranges between 0 and 1 and the more similar the vectors A and B, the higher the index. As a convection, we can think of 2 empty sets to be similar so:

$$J(A,B) = 1 \text{ if } A = B = \emptyset$$

We can also use the notion of distance instead of similarity. The **Jaccard Distance** is a metric on the collection of all finite sets and can be computed like:

$$\delta_J(A,B) = 1 - J(A,B) = \frac{|A \cup B| - |A \cap B|}{|A \cup B|}$$

4.1 Clustering Algorithms

There are many **Clustering Algorithms** but we only see the **Partitioning-based** ones in particular **K-Means**. Every partitioning-based algorithm takes a set of N data points and a number K (K<N) as input and the output is a partition of the N data points into K clusters. So, we have to find a way to group all the elements into K sets. There are a lot of possible combinations but we have to find the one that **optimizes a certain criterion**. The global optimum is intractable in most of the case because it might require to enumerate all the possible partitions and the number of possible partitions is given by:

$$S(K, N) \sim K^N/K! = O(K^N)$$

So the order is exponential to the number of data points that is possibly a huge number. We need another approach that uses some heuristics to try to find at least an **approximation of the best optimum**.

Let's assume we have a set of N input data points $\{x_1, .., x_N\}$, a set of K output clusters $\{C_1, .., C_K\}$. We represent with C_k the generic k-th cluster and with Θ_k the **representative of the cluster** C_k (a single data point represented like a vector that sort of represents the whole cluster). We can express the goodness of a cluster algorithm using an **Objective Function** (that works as a Loss Function):

$$L(A, \boldsymbol{\Theta}) = \sum_{n=1}^{N}\sum_{k=1}^{K} \alpha_{n,k}\delta(\mathbf{x}_n, \boldsymbol{\theta}_k) \qquad \forall n\ \exists! k \text{ such that } \alpha_{n,k} = 1\ \wedge\ \alpha_{n,k'} = 0\ \forall k' \neq k$$ (Hard Clustering)

The function above takes an NxK matrix and $\Theta=\{\Theta_1, .., \Theta_K\}$ set of cluster representatives in input. The entry of the A matrix is a binary value $\alpha_{n,k}$ where $\alpha_{n,k}=1$ iff the n-th element x_n is assigned to the k-th cluster C_k, 0 otherwise. Notice that we talk about **Hard Clustering**: **each data point belongs to one and only one cluster**. We have two summations where the first one sums over the N data points and the second one sums the K clusters. $\boldsymbol{\delta}(x_n, \Theta_k)$ is a function that measures the distance between the data point x_n and the cluster representative Θ_k.
Our objective is to find the best assignment matrix A* and the best clustering representatives Θ* such that the value of the Loss Function is minimized (the distance between all data points and cluster representatives is minimized):

$$A^*, \boldsymbol{\Theta}^* = \text{argmin}_{A,\boldsymbol{\Theta}} \underbrace{\sum_{n=1}^{N}\sum_{k=1}^{K} \alpha_{n,k}\delta(\mathbf{x}_n, \boldsymbol{\theta}_k)}_{L(A,\boldsymbol{\Theta})}$$

Again, the exact solution must explore the exponential search space S(K, N) ~ $O(K^N)$ so the problem is **NP-Hard**. Moreover, the function L(A, Θ) is **non-convex** due to the discrete assignment matrix A so, when minimizing the function, we could have **multiple local minima** (e.g. like in the case of Gradient Descent) that is not necessarily equal to the global minimum (so we are not guaranteed to reach the best solution).
To sum up, NP-hardness doesn't allow us to compute the exact solution and Non-convexity doesn't allow us to rely on the nice property of convex optimization with numerical methods (unique global minimum).
To find an approximate solution, we can use the **Lloyd-Forgy Algorithm** that is a **2-step iterative solution**: the **assignment step** and the **update step**.

The **Assignment Step** consists in a function L that takes in input the assignment matrix A and the clustering representatives Θ. The idea is to **minimize L**, **varying A** and **keeping Θ fixed**. We can write the Loss function L as:

$$L(A|\boldsymbol{\Theta}) = L(A; \boldsymbol{\Theta}) = L \text{ is a function of } A \text{ parametrized by } \boldsymbol{\Theta}$$

Intuitively, given a set of fixed representatives, L is minimized if each data point is assigned to the closest cluster representative according to $\boldsymbol{\delta}$ (since Θ is fixed, we have to assign the data points to the cluster representative with the lowest distance). Remember that L is just the summation of all the distances from each data point to its assigned representative. We can evaluate the elements of A in this way:

$$\alpha_{n,k} = \begin{cases} 1 & \text{if } \delta(\mathbf{x}_n, \boldsymbol{\theta}_k) = \min_{1\leq j\leq K}\{\delta(\mathbf{x}_n, \boldsymbol{\theta}_j)\} \\ 0 & \text{otherwise} \end{cases}$$

In simple words, for each row of the matrix, we choose the k-th element (the k-th cluster representative) with the lowest distance and we assign the value 1 to that element (all the others will be 0).

The **Update Step** consists in a function L that takes in input the assignment matrix A and the clustering representatives Θ. The idea is to **minimize L**, **varying Θ** and **keeping A fixed**:

$$L(\boldsymbol{\Theta}|A) = L(\boldsymbol{\Theta}; A) = L \text{ is a function of } \boldsymbol{\Theta} \text{ parametrized by } A$$

Notice that before we couldn't take the gradient of L w.r.t. A since A is discrete but now we don't have this problem anymore. We can minimize L by taking the gradient of L w.r.t Θ (i.e. the vector of partial derivatives), set it to 0 and solve it for Θ:

$$\nabla L(\boldsymbol{\Theta}; A) = \left(\frac{\partial L(\boldsymbol{\Theta}; A)}{\partial \boldsymbol{\theta}_1}, \dots, \frac{\partial L(\boldsymbol{\Theta}; A)}{\partial \boldsymbol{\theta}_K} \right) = \left(\frac{\partial L(\boldsymbol{\theta}_1 \dots \boldsymbol{\theta}_K; A)}{\partial \boldsymbol{\theta}_1}, \dots, \frac{\partial L(\boldsymbol{\theta}_1 \dots \boldsymbol{\theta}_K; A)}{\partial \boldsymbol{\theta}_K} \right)$$

$$\frac{\partial L(\boldsymbol{\theta}_1 \dots \boldsymbol{\theta}_K; A)}{\partial \boldsymbol{\theta}_j}$$

The general j-th partial derivative

∇ **(nabla)** represents the vector of all K partial derivatives of L with respect to Θ.
We want to set the gradient to 0 (notice that the 0 is bold in the formula since we mean a vector of 0):

$$\nabla L(\boldsymbol{\Theta}; A) = \mathbf{0} \Leftrightarrow \frac{\partial L(\boldsymbol{\theta}_1, \dots, \boldsymbol{\theta}_K; A)}{\partial \boldsymbol{\theta}_j} = 0 \; \forall j \in \{1, \dots, K\}$$

$$\frac{\partial L}{\partial \boldsymbol{\theta}_j}$$ To make the notation easier!

We can substitute the general formula of the Loss function with the one given before, so:

$$\frac{\partial L}{\partial \boldsymbol{\theta}_j} = \frac{\partial}{\partial \boldsymbol{\theta}_j} \left[\sum_{n=1}^{N} \sum_{k=1}^{K} \alpha_{n,k} \delta(\mathbf{x}_n, \boldsymbol{\theta}_k) \right] = \frac{\partial}{\partial \boldsymbol{\theta}_j} \left[\sum_{n=1}^{N} \alpha_{n,j} \delta(\mathbf{x}_n, \boldsymbol{\theta}_j) \right] = 0$$

Notice that when computing the partial derivative w.r.t. Θ_j any other term Θ_k of the inner summation is treated as constant so we **only need to consider all the terms that refer to Θ_j**. This means that we can get rid of the inner summation. So we must **solve for each Θ_j independently** and the **result depends on the distance function δ**.

4.1.1 K-Means

What we have seen so far is a general framework and now we see a special case that is **K-Means**. In K-Means each cluster representative is its center of mass (i.e., **centroid**) that is **the mean of the instances** assigned to that cluster. So we basically take all the elements in a cluster and compute the average: the result is the centroid of that cluster. What we can do is (Re)Assign the instances to clusters, based on distance/similarity to the current cluster centroids. The basic idea is constructing clusters so that the total within-cluster **Sum of Square Distances (SSD)** is minimized.

Let's assume we have a set of N input data points $\{x_1, .., x_N\}$, a set of K output clusters $\{C_1, .., C_K\}$. We represent with C_k the generic k-th cluster and Θ_k the **representative of the cluster** that is now the average $\boldsymbol{\mu}_k$ of the k-th cluster. So to compute Θ_k, we take all the values of the data points that are assigned to the k-th cluster and divide it by the (absolute) number of data points that are assigned to the k-th cluster (remember that $\alpha_{n,k}$=1 only when the n-th data point is assigned to the k-th cluster and in this case will contribute to the summation, otherwise 0).

$$\boldsymbol{\theta}_k = \frac{\sum_{n=1}^{N} \alpha_{n,k} \mathbf{x}_n}{\sum_{n=1}^{N} \alpha_{n,k}} = \boldsymbol{\mu}_k = \frac{1}{|C_k|} \sum_{n \in C_k} \mathbf{x}_n \qquad \text{where } |C_k| = \sum_{n=1}^{N} \alpha_{n,k}$$

The **objective function** is exactly the one seen before but here we replace the generic distance function $\boldsymbol{\delta}(x_n, \Theta_k)$ with the **Euclidean Space**:

$$L(A, \boldsymbol{\Theta}) = \sum_{n=1}^{N} \sum_{k=1}^{K} \alpha_{n,k} \underbrace{(||\mathbf{x}_n - \boldsymbol{\theta}_k||_2)^2}_{\delta(\mathbf{x}_n, \boldsymbol{\theta}_k)} = \sum_{n=1}^{N} \sum_{k=1}^{K} \alpha_{n,k} (\mathbf{x}_n - \boldsymbol{\theta}_k)^2$$

This is called **Sum of Square Distances (SSD)**. Notice that we went from left to right because:

$$\delta(\mathbf{x}_n, \boldsymbol{\theta}_k) = (||\mathbf{x}_n - \boldsymbol{\theta}_k||_2)^2 = \left[\sqrt{(\mathbf{x}_n - \boldsymbol{\theta}_k)^2} \right]^2 = (\mathbf{x}_n - \boldsymbol{\theta}_k)^2$$

Now, we can proceed like before defining the Assignment Step and the Update Step.
The **Assignment Step** consists in a function L that takes in input the assignment matrix A and the centroids Θ. The idea is to minimize L, varying A and keeping Θ fixed. Intuitively, given a set of fixed centroids, L is minimized if each data point is assigned to the centroid with the smallest SSD (L is just the SSD from each data point to its assigned centroid). We can evaluate the elements of A in this way:

$$\alpha_{n,k} = \begin{cases} 1 & \text{if } (\mathbf{x}_n - \boldsymbol{\theta}_k)^2 = \min_{1 \le j \le K}\{(\mathbf{x}_n - \boldsymbol{\theta}_j)^2\} \\ 0 & \text{otherwise} \end{cases}$$

The **Update Step** consists in a function L that takes in input the assignment matrix A and the clustering representatives Θ. The idea is to **minimize L**, **varying Θ** and **keeping A fixed**:

$$\boldsymbol{\Theta}^* = \text{argmin}_{\boldsymbol{\Theta}} \underbrace{\left\{ \sum_{n=1}^{N} \sum_{k=1}^{K} \alpha_{n,k} (\mathbf{x}_n - \boldsymbol{\theta}_k)^2 \right\}}_{L(\boldsymbol{\Theta}; A)}$$

Again, we must **compute the gradient w.r.t. Θ**, **set it to 0** and **solve it for Θ**. It is exactly like before but we replaced the distance function with the Euclidean one. In order to make the gradient equal to 0, we want that each k-th partial derivative is set to 0. Executing the formula below, we'll find the new K centroids. Again, we can remove the inner summation and we can solve the derivative of the element into brackets getting:

$$\frac{\partial L}{\partial \boldsymbol{\theta}_k} = \frac{\partial}{\partial \boldsymbol{\theta}_k} \left[\sum_{n=1}^{N} \alpha_{n,k} (\mathbf{x}_n - \boldsymbol{\theta}_k)^2 \right] = \sum_{n=1}^{N} -2\alpha_{n,k} (\mathbf{x}_n - \boldsymbol{\theta}_k) = 0 \qquad \forall k \in \{1, \dots, K\}$$

Basically we want to find the value of Θ_k that plugged in that expression gives 0 as a result:

$$\text{Find } \boldsymbol{\theta}_k^* \text{ s.t. } \sum_{n=1}^{N} -2\alpha_{n,k} (\mathbf{x}_n - \boldsymbol{\theta}_k^*) = 0$$

We can factor things out (just sum up and take one of the factor to the right of the equation changing the sign):

$$\sum_{n=1}^{N} -2\alpha_{n,k} (\mathbf{x}_n - \boldsymbol{\theta}_k^*) = 0 \Leftrightarrow 2 \sum_{n=1}^{N} \alpha_{n,k} \boldsymbol{\theta}_k^* = 2 \sum_{n=1}^{N} \alpha_{n,k} \mathbf{x}_n$$

Now, Θ^*_k doesn't depend on N so we can take it out of the summation (also divide everything by 2 to remove the multiplicative 2). If we divide everything by the summation on the right we go back from what we started:

$$\boldsymbol{\theta}_k^* \sum_{n=1}^{N} \alpha_{n,k} = \sum_{n=1}^{N} \alpha_{n,k} \mathbf{x}_n \qquad \boldsymbol{\theta}_k^* = \frac{\sum_{n=1}^{N} \alpha_{n,k} \mathbf{x}_n}{\sum_{n=1}^{N} \alpha_{n,k}} = \boldsymbol{\mu}_k = \frac{1}{|C_k|} \sum_{n \in C_k} \mathbf{x}_n$$

So, the cluster centroid (i.e. mean) minimizes the objective (for a fixed assignment A).

As said before, it is possible to use the **Lloyd-Forgy Algorithm** to solve the K-means problem. It works like this:

1. Specify the number of output clusters K (the value K can be given, otherwise we must devise it);
2. Select K observations at random from the N data points as the initial cluster centroids (this is only one of the possible initialization that we can choose and the easiest one);
3. **Assignment step**: Assign each observation to the closest centroid based on the distance measure chosen;
4. **Update step**: For each of the K clusters update the centroid by computing the new mean values of all the data points now in the cluster;
5. Iteratively repeat steps 3-4 until a **stopping criterion** is met.

So, starting from the initial cluster centroids we have to keep on assigning the N data points to the closest centroid and, since we change the observations in each cluster, we also have to recompute the centroid. We have to keep doing the assignment step and the update step until a certain point. There are several options we can choose from as a **stopping criterion**:

- **Fixed number of iterations**: As soon as we reach a certain number of iterations, we stop the algorithm;
- **Cluster assignments stop changing** (beyond some threshold): If no data points (or few of them) change at each iteration, it means we can stop the algorithm;

- **Centroid doesn't change** (beyond some threshold): Same as before but the center of mass stays still (or changes a little).

Note: We are guaranteed that in each step we improve the objective or, at most, we stay still. It is impossible that we do worse so **there exists a state in which we are sure we reach convergence**. The K-means algorithm is an example of **Expectation Maximization (EM)** that is known to converge (although not necessarily to a global optimum). The relationship with this general framework and the algorithm described before is that with EM we have an **E-step** in which each object is assigned to the closest centroid monotonically decreasing the SSD (it corresponds to the assignment step in which we choose the most likely cluster). We also have a **M-step** in which the model is updated and this monotonically decreases each SSD_k that is the sum of square distance of each cluster (it corresponds to the update step in which centroids are updated).

Note: For what concerns the **complexity analysis of the Lloys-Forgy algorithm**, computing the distance between two d-dimensional data points (d is the number of dimensions) takes O(d) (assume we have 2 vectors and we have to compute the Euclidean distance between the two). (Re)Assigning clusters (E-step) takes O(KN) distance computations or O(KNd) (for every K cluster centroids and for every N data points we have to compute the distance). Computing centroids (M-step) takes O(Nd) as there are O(N) average computations since each data point is added to a cluster exactly once at each iteration, each one taking O(d). Putting all together, the algorithm takes O(RKNd) assuming the 2 steps above are repeated R times.

Now, the **convergence rate** (how fast we reach convergence) and clustering quality (how well we group the data together) depends on the **selection of the initial centroids**. Starting the algorithm with different seeds doesn't guarantee to reach the same final result so we should find the seed that optimizes the outcome of the algorithm. Forgy method randomly chooses K data points as the initial means while random Partition method randomly assigns a cluster to each observation. The problem with the a**pproaches that use randomness is that we could lead to suboptimal clusterings**. To mitigate this problem we should execute several runs of the Lloyd-Forgy algorithm with **multiple random initiation seeds** and possibly **choose the best one**.
There is another (more efficient) method to carefully select initial centroids that is called **K-means++** that is:

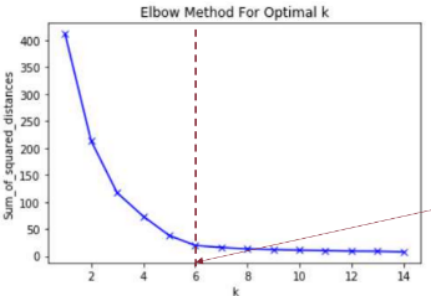
1. Choose one center uniformly at random from among the initial data points;
2. For each data point x, compute D(x) as the distance between x and the nearest center that has already been chosen;
3. Choose one new data point at random as a new center with probability proportional to $D(x)^2$;
4. Repeat steps 2 and 3 until K centers are chosen, then run Lloyd-Forgy.

So, at the beginning we randomly choose a data point k_1 (among the N data points) to be the initial center. Then, we compute the distance D(x) between the center k_1 and each remaining data point x (that are now N-1). Once we have this distance, we can construct a probability distribution based on $D(x)^2$ from which to pick the second centroid k_2 (we have to choose it among the N-1 data points). So basically, the more distant the data point with respect to the already selected centroids, the more likely to be chosen. Repeat steps 2 and 3 until K centers are chosen, then run Lloyd-Forgy algorithm as soon as the initial clusters have been chosen.
Note: The difference with “vanilla” K-means and K-means++ is that the first one may produce clusters that are **arbitrarily worse than optimum** (remember that in K-means we find an approximation of the optimum solution) while the second one provides an upper-bound to the approximation obtained w.r.t. the optimal solution in fact at most, clusters obtained with K-means++ initialization are O(log K) worse than the optimal partitioning.

Another task in K-means is to understand **the number of clusters K** to choose. In some cases it is a predetermined number (it’s kind of given by the problem we have to solve, e.g. if we have to recognize the digit in an image the trivially the number of clusters is 10). In most of the cases it isn’t possible to know K in advance so we have to find a **trade-off between having too few and too many clusters**. There are two approaches:

- **Total Benefit**: Given a clustering (some data points already grouped in a certain amount of clusters), we define the benefit b_i for a data point x_i to be the similarity to its assigned centroid (how close is the data point to the assigned centroid). We define the total benefit B to be the sum of the individual benefits. Notice that there is always a clustering whose total benefit is B=N (where N is again the number of data points) that is when we have exactly N clusters (one data point per cluster, so the data point is the cluster itself for the assigned centroid). Of course this is not a good solution since we don’t want to create so many clusters: we can assign a cost everytime we create a cluster;
- **Total Cost**: Assign a cost p to each cluster, thereby a clustering with K clusters has a total cost P=Kp. So we can define the value V of a clustering to be total benefit-total cost (V=B-P). The goal is to find the

clustering which maximizes V, over all choices of K (B increases with larger values of K, but P allows to stop that).



There is a more straightforward and empirical operation that is the **Elbow method** that is a trade-off between total benefit and total cost and **try multiple values of K and look at the change of the SSD**. Notice that as K increases, SSD sharply decreases up to a certain value. We should take K when the SSD stops decreasing substantially.

Note: So far we have focused on the Euclidean Distance ($\boldsymbol{\delta}$ = L^2-Norm). It is possible to use the hard clustering framework with other distance functions such as the cosine distance (Euclidean distance on normalized input points) or the correlation coefficient (Euclidean distance on standardized input points). In these two cases, the centroids would still be the minimizers for those distances. Other distances like the Manhattan distance ($\boldsymbol{\delta}$ = L^1-Norm) requires specific minimizers like the median (in this case we talk about **K-medians**).

Note: Some variances of K-means are the K-medoid and the BFR K-means. **K-medoids** is similar to K-means yet chooses input data points as centers (medoids). So the medoid is the closest object to any other point in the cluster (similar to the center of mass concept but it enforces the fact that the object must be a real point of the data set). **BFR K-means** instead is thought for large datasets, works better in high-dimensional Euclidean space and there is a strong assumption on the shape of clusters: Data is normally distributed around the centroid and there must be independence between data dimensions.

4.2 Measures of Clustering Quality

In order to measure the quality of a cluster, it is possible to do it internally or externally.
In **internal evaluation**, clustering is evaluated based on the data that was clustered itself. A good clustering will produce high quality clusters with a high intra-cluster similarity (the data points in a cluster are very similar to each other) and a low inter-cluster similarity (the clusters are well separated to each other). The measured quality of a clustering depends on the data representation and the similarity measure that we choose. There are some similarity measures that are:

- **Davies-Bouldin Index**: K is the number of clusters, $\boldsymbol{\mu}_k$ is the centroid of the cluster C_k, $\boldsymbol{\sigma}k$ is the average distance of all elements of the cluster C_k from its centroid $\boldsymbol{\mu}_k$ and $\boldsymbol{\delta}(\boldsymbol{\mu}_i, \boldsymbol{\mu}_j)$ is the distance between the centroids of C_i and C_j. The smaller the better since we want clusters to be very similar internally but well apart from each other.

$$DB = \frac{1}{K}\sum_{i=1}^{K} \max_{j \neq i}\left(\frac{\sigma_i + \sigma_j}{\delta(\boldsymbol{\mu}_i, \boldsymbol{\mu}_j)}\right)$$

- **Dunn Index**: K is the number of clusters, $\boldsymbol{\delta}(C_i, C_j)$ is the distance between the centroids of C_i and C_j (the distance between centroids) and $\boldsymbol{\delta}'(C_k)$ is the intra-cluster distance of the cluster C_k (that is the maximum distance between any pair of objects). Here we have the opposite situation so the higher the better.

$$D = \frac{\min_{1 \leq i < j \leq K} \delta(C_i, C_j)}{\max_{1 \leq k \leq K} \delta'(C_k)}$$

- **Silhouette Coefficient**: We take one data point i in a cluster C_i and compute:

$$a(i) = \frac{1}{|C_i| - 1}\sum_{j \in C_i, j \neq i} \delta(i,j) \qquad b(i) = \min_{k \neq i} \frac{1}{|C_k|}\sum_{j \in C_k} \delta(i,j)$$

Where a(i) is the mean distance between i and all other data points in the same cluster C_i and b(i) is the smallest mean distance of i to all points in any other cluster C_k != C_i. Finally we compute s(i) that is a score: the higher, the better.

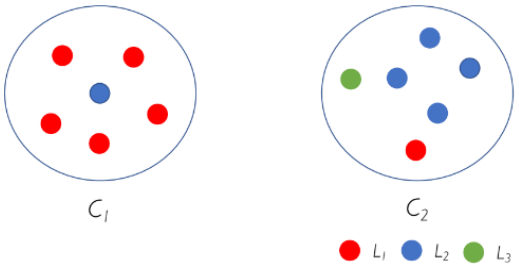
$$s(i) = \begin{cases} 1 - a(i)/b(i) & \text{if } a(i) < b(i) \\ 0 & \text{if } a(i) = b(i) \\ b(i)/a(i) - 1 & \text{if } a(i) > b(i) \end{cases}$$

In **external evaluation**, clustering is evaluated based on data that was not used for clustering, yet pre-classified. In this case, the quality is measured by the ability to discover some or all of the hidden patterns in the labeled data but of course it is difficult to achieve since in clustering we usually don't have labeled data. Some approaches are:

- **Purity**: Let's assume to have C_1, .., C_K clusters, L_1, .., L_J labels, $n_{i,j}$ the number of items with label L_j clustered in C_i and n_i the total number of elements in cluster C_i. The purity is computed as;

$$\text{purity}(C_i) = \frac{1}{n_i}\max_{j\in\{1,\dots,J\}} n_{i,j} \qquad \text{purity} = \frac{1}{K}\sum_{i=1}^{K}\text{purity}(C_i)$$

EX. Let's assume to have this clustering:



$purity(C_1) = 1/6 * max\{5, 1, 0\} = 5/6$

$purity(C_2) = 1/6 * max\{1, 4, 1\} = 4/6 = 2/3$

$purity(C_3) = 1/5 * max\{2, 0, 3\} = 3/5$

$purity = 1/3 * purity(C_1)+purity(C_2)+purity(C_3) = 7/10$

- **Rand Index**: It measures the level of agreement between clustering and ground truth. We basically have to compute the **confusion matrix** where:
  - True Positives (TP): Same cluster in ground-truth and clustering;
  - True Negatives (TN): Different clusters in ground-truth and in clustering;
  - False Positives (FP): Different clusters in ground-truth and same cluster in clustering;
  - False Negatives (FN): Same cluster in ground-truth and different clusters in clustering.

  Once we have the confusion matrix, we can compute measures like the precision, the recall or the F_1-score. Notice that we can also compute the F_β-score that balances the contribution of false negatives by weighting recall through a parameter β.

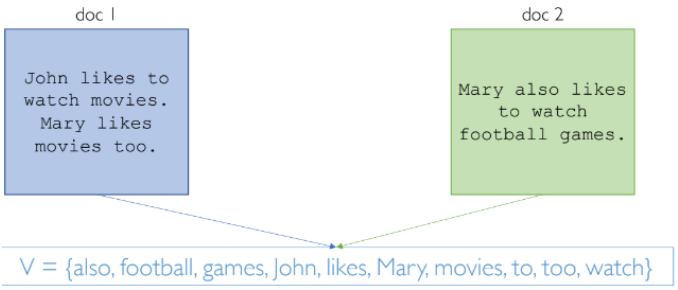
$$P = \frac{TP}{TP+FP} \quad R = \frac{TP}{TP+FN} \quad F_\beta = \frac{(\beta^2+1)\cdot P\cdot R}{\beta^2\cdot P+R} \quad F_1 = \frac{2\cdot P\cdot R}{P+R}$$

4.3 Document Clustering

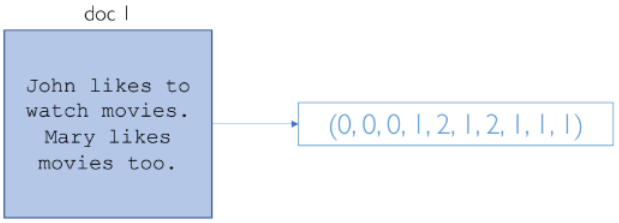
An example of clustering is **document clustering** in which the goal is to group together documents on the same topic. The basic idea is that documents with similar sets of words may be about the same topic. Of course this operation is not easy and depends on how we decide to **represent the documents** (so that they are machine readable) and we **measure their similarity**. There are different ways of representing documents (in space of words) that are:

- As a **set of words** (disregarding the order and multiplicity): We simply take the document, extract the words and put them in a set without considering the order and the fact that a word could appear several times. For example:

- As a **bag-of-words** (i.e., a multiset disregarding the order yet keeping multiplicity): We take the document, we extract the words and save how many times they appear in the document (think of a Python dictionary where the key is the word and the value is the number of occurrences). Bag-of-Words (BoW) model is just a preliminary step for more complex document representations in which we have to consider the **vocabulary** and a **weighing scheme**. In order to create the **vocabulary V**, we can take all the documents, extract all the words and put them in a set. For example:



For what concerns the **weighting scheme**, we can create a A |V|-dimensional vector, where the i-th component indicates the multiplicity of the i-th word of the vocabulary. For example:



Let's define it formally: Given D = {d_1, .., d_N} a collection of N documents, V = {w_1, .., $w_{|V|}$} the vocabulary of |V| words extracted from D, d_i = (f(w_1, i), .., f($w_{|V|}$, i)) the |V|-dimensional vector representing d_i and f:VxD → R a function that maps each word of a document to a real value (so basically the weighting scheme). The **One-Hot (binary) weighting scheme** allows us to simply understand if a vocabulary term w_j appears in the document d_i (i is the id of the document):

$$f(w_j, i) = \begin{cases} 1 & \text{if } w_j \text{ appears in } d_i \\ 0 & \text{otherwise} \end{cases}$$

In this case we don't take multiplicity into account so we can use the **Term-Frequency weighting scheme** where tf computes the number of times word w_j occurs in document d_i:

$$f(w_j, i) = tf(w_j, i)$$

A more advanced scheme is the **TF-IDF weighting scheme** that captures how important a word is with respect to a specific document. There might be some words that appear several times in different documents and these words might not be so relevant to the clustering operation. So we want to distinguish if a word is frequent only in a single document or in all documents therefore we want to lower down the score. We compute the word frequency as before but we lower it down by a factor idf(w_j) (*idf* stands for "inverse document frequency") (n_j is the number of documents in D containing the word w_j):

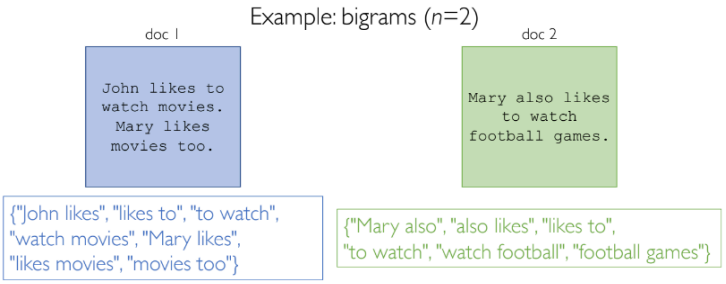
$$f(w_j, i) = tf(w_j, i) * idf(w_j) \qquad idf(w_j) = \log\left(\frac{N}{1 + n_j}\right)$$

Note: The +1 term in the denominator if put there so that if we want to use the vocabulary with some other documents, we are sure that the denominator is never equal to 0 (notice that if we use it with the documents that we used to build it then the denominator would be always at least 1 since that word appear at least one time in the document).

Some **limitations of the BoW model** are the **high dimensionality** (when creating the vocabulary and computing the weighting scheme on a document, it is likely that the result is composed by a lot of 0 due to sparseness) and **no sequential information nor semantics is included** (we simply select unigram of words in a document but we don't have any information on the order of the words → bag-of-n-grams).

- As a **bag-of-n-grams** (i.e., the more general case of bag-of-words in which we select n-grams): It is exactly like the previous case but instead of taking single words, we can group together a set of words.

For example:

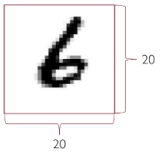


- More advanced representations derived from Neural Language Models (e.g. **word2vec**): word2vec represents each distinct word with a vector. The vectors are chosen carefully such that they capture the semantic and syntactic qualities of words, as such, a simple mathematical function (cosine similarity) can indicate the level of semantic similarity between the words represented by those vectors. For example, the words “king” and “queen” are distinct words w.r.t. the gender and they are similar to the words “man” and “woman”.

Once we found a way to represent documents, several similarity measures can be used such as:

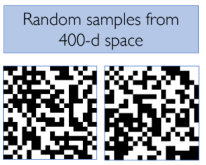
- Set of words → Jaccard coefficient (a simple method for a simple representation);
- One-hot bag-of-words → Euclidean distance (remember that when the space is sparse it could cause some problems);
- tf or tf-idf bag-of-words → Cosine similarity (we don’t care about the length of the document but the angle between them that describes the direction of those documents in the space, if the direction is the same then the documents are likely to be similar).

4.4 Dimensionality Reduction (PCA)

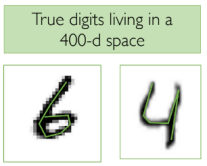


We know from the last chapter that clustering high-dimensional data may be problematic due to the curse of dimensionality but in most of the cases this problem is not real due to the way in which we observe/collect data (data is not uniformly distributed but there are some pattern we can look for that helps clustering the data).

Let’s first of all give an example: Let’s think of the digit recognition operation in which we have different images representing a digit from 1 to 9 and we need to train a model that allows us to understand what’s the digit in the image. We can represent each digit by a 20x20 bitmap (binary image), so a 400-dimensional vector. We can easily notice that digits aren’t random samples from the 400-d space but they have a specific pattern, so they just cover a tiny fraction of all the huge space (with small variations of the pen-stroke). This means that the **modeled dimensionality is different from the true dimensionality**.

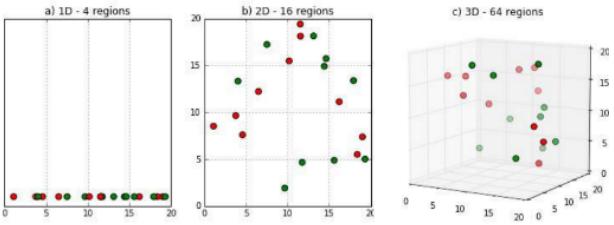


To sum up what we said so far, as dimensionality grows the fewer examples we have in each region of the feature space (assuming the number of examples is a constant). In another way, if we want to maintain the same density, the number of examples must grow exponentially with dimensionality. EX. In the example of the right it is possible to notice how the data gets sparser as the dimensionality grows larger and larger.



There are some ways to **deal with high dimensionality**:

- Feature Engineering: We can use the domain knowledge to extract some useful information (features) from the data. It requires a lot of effort and some techniques used for a certain domain may not work for another domain;
- Making Assumption: We can make assumptions on the data. For example, we can assume each dimension is independent from each other and count classes in each region along each dimension separately. Otherwise we can assume some smoothness on the data and propagate counts to neighboring regions or assume some symmetry between dimensions.



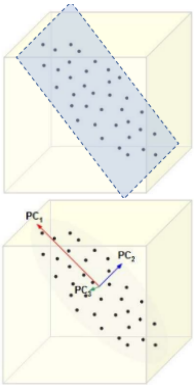
- **Reduce Dimensionality**: It is about creating a new set of dimensions (i.e. features).

The technique we focus on is the **dimensionality reduction** that allows us to unveil the actual dimensions of the data. It is no more than a **pre-processing step** for trying to represent the data with fewer features **preserving as much "structure"** of the data as possible in order not to lose too much information (with structure we mean **variance**). There are 2 main approaches to dimensionality reduction:

- Feature Selection: Assuming to have a vector of features x_1, .., x_d, it is about **picking a subset of the original dimensions** that are good predictors (so pick the most informative features) (e.g. it can done using some criterion such as information gain);
- **Feature Extraction**: It consists in building a new set of k < d dimensions as a (linear) combination of the originals e_1, e_2, .., e_k so $e_i = f(x_1, x_2, ..., x_d)$.

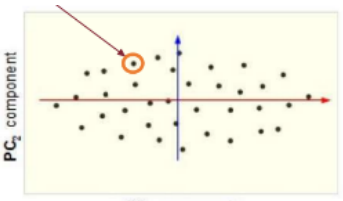
We focus on the latter, in particular on a dimensionality reduction technique based on feature extraction that is called **Principal Component Analysis (PCA)**. What usually happens in high-dimensional data is that it is embedded into some lower dimensional space. PCA defines a set of principal components as follows:

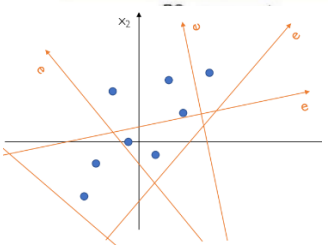
1. We choose the one with the direction of the greatest variance of data;
2. The take the perpendicular to the 1st and greatest variance of what's left;
3. And so on until d;

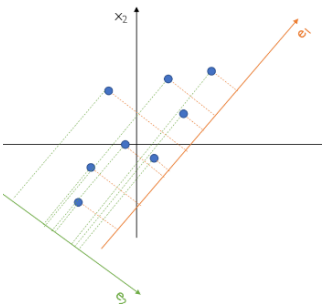


EX. On the right we can see a 3-d set of points embedded into a 2-d hyperplane. The data points can be represented by a new coordinate system that in this case sits in a 2-d hyperplane so k=2. So, PC_1 and PC_2 are the top-2 principal components and we can change the coordinates of every point according to the new dimensions (so it is like we change the origin and transform the data accordingly).

EX. Now, let's make another example to understand why we have to look for the greatest variance. Let's assume we have 2-dimensional data and we want to reduce it to 1-d. First of all, we center the points around the mean along x_1 and x_2 (we can simply compute the mean along x_1 and along x_2 and shift the points accordingly). Now, what we can do is project (x_1, x_2) to a **single dimension axis e**. Now, we don't actually know how to choose this axis since there are possible infinite mappings from (x_1, x_2) to a new axis e. So let's consider 2 different mappings e_1 and e_2 (almost perpendicular to the other). It is easy to notice that points projected onto e_1 look more spread-out than onto e_2 so the **variance along e_1 is larger than along e_2**. This means that if we take 2 points that are far away from each other, they end up very close if projected onto e_2 (while if projected onto e_1 they better preserve their distance).







Intuitively, we want to **minimize the chance that 2 points that are far in the original space end up close in the lower dimensional space**. So we should minimize the distances between points as measured on the (x_1, x_2) space and those measured on e that means that we have to pick e so that we **maximize the variance of projected data** (so that the distances in the original space and the points on the projected space are preserved and, as said before, we also minimize the chance that 2 points that are far away in the original space end up very close in the projected space).

Before going on, let's remember what a covariance matrix is. First of all the **variance of a random variable X** measures how far a set of (random) numbers are spread out from their mean value. Formally, it is the expected value of the squared deviation from its mean:

$$\mathrm{Var}(X) = \mathrm{E}[(X-\mu)^2] \quad \text{where } \mu = \mathrm{E}[X]$$

The **covariance of two random variables** is a measure of the joint variability of two random variables X and Y (basically what we want to understand if X and Y increase/decrease together or when one increases/decreases the other decreases/increases). Formally, it is the expected value of the product of their deviations from their individual means:

$$\mathrm{Cov}(X,Y) = \mathrm{E}[(X-\mu_X)(Y-\mu_Y)] \quad \text{where } \mu_X = \mathrm{E}[X] \text{ and } \mu_Y = \mathrm{E}[Y]$$

Note: $Cov(X, X) = E[(X - \mu_X)(X - \mu_X)] = E[(X - \mu_X)^2] = Var(X)$

Given a random vector $X = (X_1, ..., X_d)$ its **covariance matrix K** is a dxd square matrix with the covariance between each pair of elements:

$$K[i, j] = \text{Cov}(X_i, X_j)$$

Note: This means that in the matrix diagonal there are variances (i.e. the covariance of each element with itself).

Now, notice that the original set of dimensions is a random vector $X = (X_1, .., X_d)$ (in particular, in our previous example, d=2 and $X = (X_1, X_2)$ so basically the covariance matrix K is a 2x2 matrix). To ease the covariance computation, we can center each data point at zero: This can be done by **subtracting the mean of each attribute/dimension** (so we do this computation so that the mean of each dimension becomes 0). Let's assume we have a set of n data points $x_1, .., x_n$ and we use only 2 dimensions, then each data point is represented by a (x_1, x_2) pair (basically, $x_i = (x_{i,1}, x_{i,2})$). We can associate 2 random variables X_1, X_2 to each dimension and compute:

$$\mu_1 = \text{E}[X_1] = \frac{1}{n}\sum_{i=1}^{n} x_{i,1} \qquad \mu_2 = \text{E}[X_2] = \frac{1}{n}\sum_{i=1}^{n} x_{i,2}$$

So, we compute the average of the dimensions (summing all the data points of the first dimension and dividing the result by the number of data points) and finally use them for the subtraction operation:

$$\mathbf{x}_i = (x_{i,1} - \mu_1, x_{i,2} - \mu_2)$$

So, for each data point, we subtract the mean of the respective component. Basically we create a new vector where all the data points are scaled by the mean (the mean of the new vector is exactly 0). Formally we define the new data points as follows:

$$\mathbf{x}_i = (x'_{i,1}, x'_{i,2}) \text{ where: } x'_{i,1} = x_{i,1} - \mu_1; x'_{i,2} = x_{i,2} - \mu_2$$

Now, we can easily see that the **new mean is equal to 0**:

$$\mu_1^{\text{new}} = \text{E}[X_1] = \frac{1}{n}\sum_{i=1}^{n} x'_{i,1} = \frac{1}{n}\sum_{i=1}^{n}(x_{i,1} - \mu_1) = \frac{1}{n}\Big(\underbrace{\sum_{i=1}^{n} x_{i,1}}_{n\mu_1} - \underbrace{\sum_{i=1}^{n}\mu_1}_{n\mu_1}\Big) = 0$$

$$\mu_2^{\text{new}} = \text{E}[X_2] = \frac{1}{n}\sum_{i=1}^{n} x'_{i,2} = \frac{1}{n}\sum_{i=1}^{n}(x_{i,2} - \mu_2) = \frac{1}{n}\Big(\underbrace{\sum_{i=1}^{n} x_{i,2}}_{n\mu_2} - \underbrace{\sum_{i=1}^{n}\mu_2}_{n\mu_2}\Big) = 0$$

Basically, we just rewrite the mean of the new data points. Now, notice that μ_1 doesn't depend on i (the mean is just a constant) so we can decompose the summation. So, the second summation is simply $n*\mu_1$ and the first one can be seen as an inverse average formula (it is like we sum all the values but don't divide them by n so we remain with an n at the numerator). Finally, we subtract the terms (that are the same) and end up with 0. The same idea applies with μ_2.

Scaling data so as to have 0-mean on all dimensions allows **computing each entry of the covariance matrix much more easily** (the covariance matrix is also easier to compute):

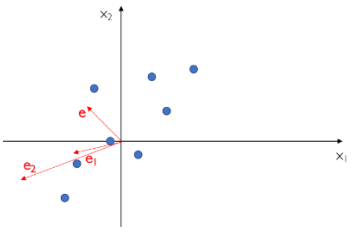
$$\text{Cov}(X_1, X_2) = \text{E}[(X_1 - \underbrace{\mu_1^{\text{new}}}_{=0})(X_2 - \underbrace{\mu_2^{\text{new}}}_{=0})] = \text{E}[X_1 X_2]$$

Now, let's assume the following is our 2-by-2 covariance matrix, this means we can compute the covariance as:

$$\begin{array}{c} \\ x_1 \\ x_2 \end{array}\begin{array}{c} \begin{array}{cc} x_1 & x_2 \end{array} \\ \begin{bmatrix} 2 & 4/5 \\ 4/5 & 3/5 \end{bmatrix} \end{array}$$

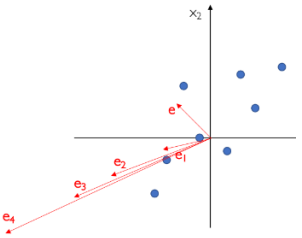
$$\text{Cov}(X_1, X_2) = \frac{1}{n}\sum_{i=1}^{n} x'_{i,1} * x'_{i,2}$$

$$\text{Cov}(X_2, X_2) = \text{Var}(X_2) = \frac{1}{n}\sum_{i=1}^{n}(x'_{i,2})^2$$



Notice that the elements in the main diagonal are the variance. Let's multiply our 2-by-2 covariance matrix K by a random vector e = (-1,1):

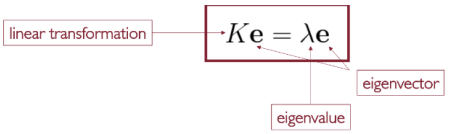
$$\underbrace{\begin{bmatrix} 2 & 4/5 \\ 4/5 & 3/5 \end{bmatrix}}_{K} \underbrace{\begin{bmatrix} -1 \\ 1 \end{bmatrix}}_{\mathbf{e}} = \underbrace{\begin{bmatrix} -6/5 \\ -1/5 \end{bmatrix}}_{\mathbf{e_1}}$$



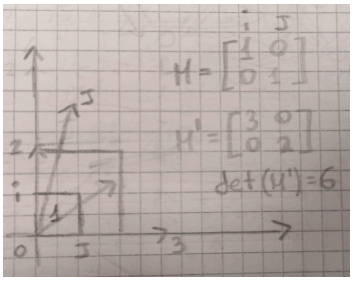
The new vector e_1 (that results from the multiplication between the covariance matrix K and random vector e) is turning towards the direction of greatest variance (that's what we want). The slope of e is 1/-1 = -1 while the new slope is -(1/5)/-(6/5) = 1/6. Let's repeat the previous step multiplying the covariance matrix K by e_1 = (-6/5, -1/5) (the slope becomes -(27/25)/-(64/25) = 27/64):

$$\underbrace{\begin{bmatrix} 2 & 4/5 \\ 4/5 & 3/5 \end{bmatrix}}_{K} \underbrace{\begin{bmatrix} -6/5 \\ -1/5 \end{bmatrix}}_{\mathbf{e_1}} = \underbrace{\begin{bmatrix} -64/25 \\ -27/25 \end{bmatrix}}_{\mathbf{e_2}}$$

If we keep doing this the resulting vector is **getting longer** and **turns towards the direction of the largest variance. The slope converges to the dimension where data is mostly spread off.** In linear algebra those vectors are called **eigenvectors**.



Now, let's slightly go into more detail. When multiplying a matrix by an **eigenvector** e the resulting vector does not change its direction, but it is only scaled by a factor λ (**eigenvalue**). In other words, eigenvectors encapsulate all the relevant information to describe a linear transformation (in our case, represented by the covariance matrix K). We call **principal components** the **eigenvectors of the covariance matrix with the largest eigenvalues**.



Before going on, let's make an example. Let's assume we have a matrix M, we can apply a linear transformation so that we get a matrix M'. Notice that the area of M is just 1. It would be interesting to know how the area changes when applying the linear transformation and this can be achieved using the determinant. If we compute the determinant of M', we find out the det(M') = 6 that means that the area of M' is 6 times the original area (before the linear transformation) in fact the area of M' is exactly equal to 6. So, the determinant suggests how the area increases/decreases when applying some transformation.
We can solve now the following homogeneous system (just take everything to the right):

$$(K - \lambda I)\mathbf{e} = 0$$

Any homogeneous system always has a trivial solution, i.e. the zero vector (**e = 0**). **We don't want a trivial solution** and the only way for the homogeneous system above to have a non-trivial solution is for its matrix (K – λI) to be **non-invertible**, otherwise:

$$\cancel{(K - \lambda I)(K - \lambda I)^{-1}}\mathbf{e} = 0\cancel{(K - \lambda I)^{-1}}$$

So, a square matrix is **invertible** if and only if its determinant is not equal to 0. If the determinant of the matrix (K – λI) is equal to 0, it is non-invertible so the corresponding homogeneous system will have a non-trivial solution. So, again, **we have to find λ so that the determinant is equal to 0**.

How do we compute eigenvectors?
First of all we have to find the eigenvalues by solving for λ: det(K – λI) = 0.
EX.

$$\det\left(\underbrace{\begin{bmatrix} 2-\lambda & 4/5 \\ 4/5 & 3/5-\lambda \end{bmatrix}}_{K-\lambda I}\right) = 0$$

$$(2-\lambda)(3/5-\lambda)-(4/5)(4/5) = \lambda^2-13/5\lambda+14/25$$

$$\lambda^2 - 13/5\lambda + 14/25 = 0 \quad \text{characteristic equation of K}$$

$$\lambda_1 = \frac{13+\sqrt{113}}{10} \approx 2.36; \quad \lambda_2 = \frac{13-\sqrt{113}}{10} \approx 0.24$$

We compute the determinant of the matrix K-λI and get to the **characteristic equation** of K. We can solve it (it is just a quadratic equation where the variable is λ) and get the values of λ (**eigenvalues**). If we plug one of the two into the matrix, we get a new matrix whose determinant is equal to 0 and therefore, we know that the matrix is

non-invertible. In this case, the resulting system of equations has infinitely many solutions. Now, we can plug each eigenvalue in to find the corresponding eigenvector:

$$\underbrace{\begin{bmatrix} 2 & 4/5 \\ 4/5 & 3/5 \end{bmatrix}}_{K} \underbrace{\begin{bmatrix} e_{1,1} \\ e_{1,2} \end{bmatrix}}_{\mathbf{e_1}} = \lambda_1 \underbrace{\begin{bmatrix} e_{1,1} \\ e_{1,2} \end{bmatrix}}_{\mathbf{e_1}} \qquad \underbrace{\begin{bmatrix} 2 & 4/5 \\ 4/5 & 3/5 \end{bmatrix}}_{K} \underbrace{\begin{bmatrix} e_{2,1} \\ e_{2,2} \end{bmatrix}}_{\mathbf{e_2}} = \lambda_2 \underbrace{\begin{bmatrix} e_{2,1} \\ e_{2,2} \end{bmatrix}}_{\mathbf{e_2}}$$

There is a relationship with the two solutions and we'll get to that. Let's see what happens for λ_1:

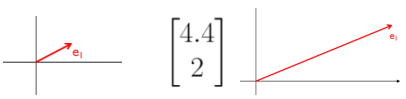
$$\begin{cases} 2e_{1,1} + 4/5e_{1,2} = \frac{13+\sqrt{113}}{10} e_{1,1} \\ 4/5e_{1,1} + 3/5e_{1,2} = \frac{13+\sqrt{113}}{10} e_{1,2} \end{cases}$$

Just replacing $\lambda_1 \sim 2.36$

$$\begin{cases} 2e_{1,1} + 0.8e_{1,2} = 2.36e_{1,1} \\ 0.8e_{1,1} + 0.6e_{1,2} = 2.36e_{1,2} \end{cases}$$

By solving the system we get that $e_{1,1} \approx 2.2e_{1,2}$ so basically, **the system has infinitely many solutions**.
This means that any vector that satisfies the relationship above ($e_{1,1} \approx 2.2e_{1,2}$) works, so all the vectors that lays along the same slope:

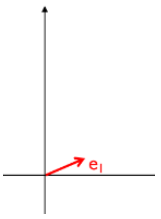
$$\begin{bmatrix} 2.2 \\ 1 \end{bmatrix} \begin{matrix} e1,1 \\ e1,2 \end{matrix} \qquad \begin{bmatrix} 1.1 \\ 0.5 \end{bmatrix} \qquad \begin{bmatrix} 4.4 \\ 2 \end{bmatrix}$$



By convention, we restrict to $||e_1|| = 1$ (where $||e_1|| = \text{sqrt}(e_{1,1}^2 + e_{1,2}^2)$).
The second eigenvector e_2 can be found by plugging in the smaller eigenvalue λ_2. In order to do so we can follow exactly the same procedure as before or by noticing that it is just **orthogonal to the previously found e_1**.

$$\begin{bmatrix} 0.91 \\ 0.41 \end{bmatrix}$$

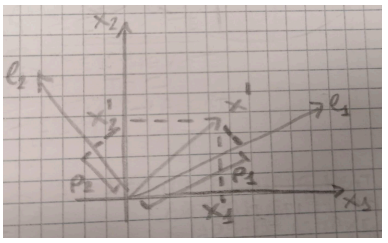
$$e_{1,1} \approx 2.2e_{1,2}$$



Note: For any dxd real symmetric matrix (like K), the eigenvalues are real and eigenvectors can
be chosen real and orthonormal.
To sum up:

- e_1 is the 1st principal component as it is the **eigenvector corresponding to the largest eigenvalue**;
- e_2 is the 2nd principal component as it is the **eigenvector corresponding to the second largest** (in the previous example the smallest) **eigenvalue**.

$$\mathbf{e_1} = \begin{bmatrix} 0.91 \\ 0.41 \end{bmatrix} \quad \mathbf{e_2} = \begin{bmatrix} -0.41 \\ 0.91 \end{bmatrix}$$



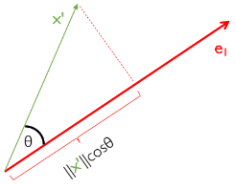
So, e_1 and e_2 identify our new coordinate system (**principal components**) that are both 2-dimensional vectors. Let x = (x_1, x_2) be a point (i.e., a vector) in the original (x_1, x_2)-space, then we want to represent x in the new (e_1, e_2)-coordinate system (so we are basically interested in P1 and P2 in the image on the right).

To get to our solution, first of all let's **center x around the mean of each dimension**:

$$\mathbf{x}' = \mathbf{x} - \boldsymbol{\mu} = (x_1 - \mu_1, x_2 - \mu_2)$$

Now, we can project x' on each dimension e_1 and e_2:

$$\mathbf{x}' = \underbrace{(x'_1, x'_2)}_{\text{coordinates of } \mathbf{x}' \text{ in the } (\mathbf{e}_1,\mathbf{e}_2)\text{-space}} = (\mathbf{x}'^T\mathbf{e}_1, \mathbf{x}'^T\mathbf{e}_2)$$



Note: We use the dot product because of some cosine properties that are:

$\cos\theta = (x'e_1)/||x'||||e_1||$ $\quad ||x'||\cos\theta = x'e_1/||e_1||$ $\quad ||x'||\cos\theta = x'e_1$

Noticing that the $||e_1||=1$ by convention (as we defined before).

The **new coordinates of the original data point x according to the eigenvectors e_1 and e_2** are as follows:

$$\mathbf{x}' = \begin{bmatrix} x'_1 \\ x'_2 \end{bmatrix} = \begin{bmatrix} \mathbf{x}'^T\mathbf{e}_1 \\ \mathbf{x}'^T\mathbf{e}_2 \end{bmatrix} = \begin{bmatrix} (x_1 - \mu_1)e_{1,1} + (x_2 - \mu_2)e_{1,2} \\ (x_1 - \mu_1)e_{2,1} + (x_2 - \mu_2)e_{2,2} \end{bmatrix}$$

Now, so far we have assumed that we have 2-dimensional points. Let's see how to behave if we have **d-dimensions** data points:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix}$$

Original d-dimensional data point

$$\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_k \qquad k \ll d, \ \mathbf{e}_i \in \mathbb{R}^d$$

k << d principal components

Let's assume we choose k << d, what we have to do is to apply mean centering again (by subtracting the mean from all coordinates as we did before). Then we project the data points to the k principal components by taking the dot product of the centered vector times the corresponding i-th eigen vector:

1. Mean centering

$$\mathbf{x}' = \mathbf{x} - \boldsymbol{\mu} = \begin{bmatrix} x_1 - \mu_1 \\ x_2 - \mu_2 \\ \vdots \\ x_d - \mu_d \end{bmatrix}$$

2. Projection to principal components

$$\mathbf{x}' = \begin{bmatrix} x'_1 \\ x'_2 \\ \vdots \\ x'_k \end{bmatrix} = \begin{bmatrix} (\mathbf{x} - \boldsymbol{\mu})^T \mathbf{e}_1 \\ (\mathbf{x} - \boldsymbol{\mu})^T \mathbf{e}_2 \\ \vdots \\ (\mathbf{x} - \boldsymbol{\mu})^T \mathbf{e}_k \end{bmatrix} = \begin{bmatrix} (x_1 - \mu_1)e_{1,1} + (x_2 - \mu_2)e_{1,2} + \ldots + (x_d - \mu_d)e_{1,d} \\ (x_1 - \mu_1)e_{2,1} + (x_2 - \mu_2)e_{2,2} + \ldots + (x_d - \mu_d)e_{2,d} \\ \vdots \\ (x_1 - \mu_1)e_{k,1} + (x_2 - \mu_2)e_{k,2} + \ldots + (x_d - \mu_d)e_{k,d} \end{bmatrix}$$

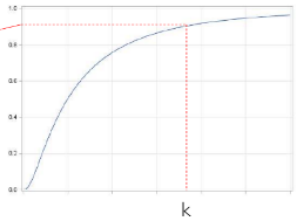
So we basically reduce the initial d-dimensional space into another system of coordinates that is k-dimensional (where k << d) in which to represent the data.

We have only "visually proved" that eigenvector e turns towards the direction of the greatest variance. To actually prove it, we need to show that e maximizes the variance among all possible projections. Moreover, we pick the eigenvector with the **largest eigenvalue λ** because λ is exactly the variance of the data along that eigenvector.

Now, before we talked about k << d but we didn't specify how to actually find the right k (so the **right number of dimensions**). In a d-dimensional space we may have e_1, ..., e_d length-1 eigenvectors and we want to select k dimensions (k << d). We know that eigenvalue λ_i is the variance along e_i and the overall variance of the data is given by the sum of all the eigenvalues. What we can do is to pick the subset of k eigenvectors that has the most variance. So we take all the eigenvalues and **sort eigenvectors by eigenvalues** such that $\lambda_1 \geq \lambda_2 \geq .. \geq \lambda_d$. After that, we can simply pick the first k eigenvectors that explain x% of the total variance. In practice, we can sum all the k eigenvalues and divide the result by the sum of all the d eigenvalues. This is no more than a normalization so the result will be smaller or equal to 1. We have to choose k so that the result explains x% of the variance (e.g. x = 0.90 that means that the k-dimensional space captures 90% of the variance w.r.t. the initial d-dimensional space).

$$\frac{\sum_{i=1}^{k} \lambda_i}{\sum_{i=1}^{d} \lambda_i} \leq 1$$

e.g., x = 90÷95%

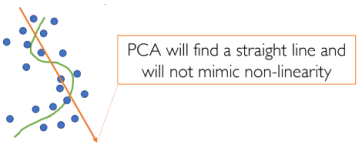


Some **practical issues of PCA** are:

- Covariance and variance are highly sensitive to large values so if one dimension takes on extremely large values (e.g. outliers) w.r.t. other dimensions the former will be considered as the principal component. The solution is to normalize each dimension to 0-mean and 1-std-deviation:

$$z = \frac{x - \mu}{\sigma}$$

- PCA assumes the projection subspace is linear (i.e. an hyperplane that in 1-d is straight line, in 2-d is a flat surface and so on and so forth). If data live in a low-dimensional but not linear space (i.e., manifold), PCA can still be applied but may not work nicely.



To make a summary:

- PCA is a dimensionality reduction technique which allows representing high-dimensional data into low-dimensional linear subspace (the original space is rotated to make dimensions uncorrelated (i.e. independent));
- Reduced size of data means faster processing and smaller storage;
- PCA can be very expensive for many very large-scale applications;
- If data do not live on a linear subspace PCA may not work well.

Note: Before moving on, let's make a definition. A d×d symmetric real matrix A is said to be **positive-semidefinite** (or non-negative-definite) iff $\mathbf{v}^T\mathbf{A}\mathbf{v} \geq 0 \ \forall \mathbf{v} \in \mathbf{R}^d$. The **correlation matrix K is positive-semidefinite**, let's prove it by absurdity. Let us assume that K is not positive-semidefinite. According to the definition, that means there should exist at least one vector $w \in R^d$, such that $w^T Kw < 0$. By using the fact that $K = Z^T Z$, we can rewrite this expression as follows:

$$\boldsymbol{w}^T K \boldsymbol{w} = \boldsymbol{w}^T Z^T Z \boldsymbol{w} = (Z\boldsymbol{w})^T (Z\boldsymbol{w})$$

Notice that Zw is the result of a n×d matrix (Z) multiplied by a d×1 column vector (w), i.e., an n×1 column vector:

$$Z\boldsymbol{w} = (\boldsymbol{z}_1^T \cdot \boldsymbol{w}, \boldsymbol{z}_2^T \cdot \boldsymbol{w}, \ldots, \boldsymbol{z}_n^T \cdot \boldsymbol{w})^T$$

where each $z_i^T \bullet w = \sum_{j=1}^{d} z_{i,j} \bullet w_j$. Let us define $\alpha_i = z_i^T \bullet w$ and, thus, $\alpha = (\alpha_1, \alpha_2, .., \alpha_n)^T$ then:

$$Z\boldsymbol{w} = (\alpha_1, \alpha_2, \ldots, \alpha_n)^T = \boldsymbol{\alpha}$$

Now, we can rewrite $(Zw)^T(Zw)$ as follows:

$$(Z\boldsymbol{w})^T(Z\boldsymbol{w}) = \boldsymbol{\alpha}^T \cdot \boldsymbol{\alpha} = \sum_{i=1}^{n} \alpha_i * \alpha_i = \sum_{i=1}^{n} \alpha_i^2$$

In other words, we have shown that:

$$\boldsymbol{w}^T K \boldsymbol{w} = (Z\boldsymbol{w})^T(Z\boldsymbol{w}) = \sum_{i=1}^{n} \alpha_i^2$$

Since $\sum_{i=1}^{n} \alpha_i^2$ is a sum of squares, it turns out it cannot be negative and this contradicts our initial assumption.

Therefore, the correlation matrix K is indeed positive-semidefinite:

$$\sum_{i=1}^{n} \alpha_i^2 \geq 0 \Rightarrow \boldsymbol{w}^T K \boldsymbol{w} \geq 0$$

Note: Another definition. Any **positive-semidefinite d×d real matrix A has non-negative eigenvalues**.
The proof is as follows: If A is a positive-semidefinite then the following must hold true: $v^TAv \geq 0 \ \forall v \in R^d$. Moreover, by definition of eigenvectors/eigenvalues: $Av = \lambda v$. If we multiply both sides of the equation above by v^T, we obtain: $v^TAv = v^T\lambda v = \lambda v^Tv$. Since $v^Tv \geq 0$, then it must also be $\lambda \geq 0$ in order to satisfy $v^TAv \geq 0$.

5. Supervised Learning

Contrarily to unsupervised learning, in **Supervised Learning** the data that we use to build a model is labeled. What we do is create a model that is trained with some labeled data and this model (also called **Learning Algorithm**) is capable of giving us the class/value of an unknown data point. So, given an input x and a function f that is **learned by the learning algorithm** from a (large) **set of labeled data**, the function outputs a value y (y = f(x)). Now, supervised learning (as well as unsupervised learning and reinforcement learning) is a part of a larger component that is **machine learning**. There are few definitions of machine learning and one of them is: "(machine learning is) the field of study that gives computers the ability to learn without being explicitly programmed". So our goal is to **teach machines to learn** (i.e. extract knowledge) from data.
Concerning supervised learning, there are two types of prediction we can do:

- **Regression**: The target y we want to predict is a **continuous real value** (e.g. y = the price of a house);
- **Classification**: The target y we want to predict is a **discrete value** (e.g. y = spam/non-spam).

There are few stages of Supervised Learning:

1. Be sure **your problem needs actually to be tackled using Machine Learning techniques** (i.e. there is no point in adopting any fancy ML solution if it can be solved “directly”);
2. **Data collection**: Get data from your domain of interest;
3. **Feature engineering**: Represent raw data we collect in a “machine-friendly” format;
4. **Model training**: Build one (or more) learning models;
5. **Model selection/evaluation**: Pick the best-performing model according to some quality metrics.

The **Data Collection** step requires retrieving the data. Supervised Learning requires labeled data which may be even harder to get, so this is not an easy task and might involve combining multiple and heterogeneous sources. Once the data has been collected, it has to be encoded with a machine-readable format and this is done in the **Feature Engineering** step. What usually happens is that each domain object is translated into a n-dimensional vector of features (so every data point has n vector where the j-th element represents a feature). In general, Each feature is a **property of an instance** of our domain (e.g. number_of_bedrooms in the case our domain objects are “houses”), it can be either derived locally from an instance (e.g. annual_income of a person) or it can be the result of more complex computation involving the whole data collection (e.g. tf-idf of a word of a document w.r.t. a corpus). Traditionally this feature engineering step is done manually by human experts that have in-depth

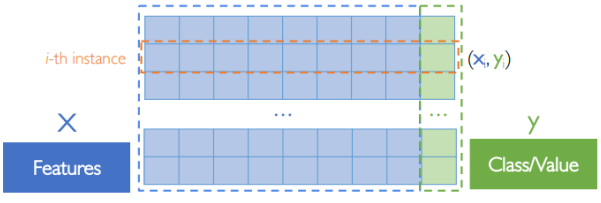
knowledge of the specific domain of the application but this is quite time consuming so there are techniques to automatically learn data representation (i.e. features), in particular:

- K-means clustering, PCA, autoencoders (unsupervised);
- Neural Networks (supervised).

Now, as said before, collecting (raw) data is not an easy task and many challenges need to be addressed before taking any further step down to the ML pipeline seen before: this step is done with **Data Preprocessing**. The challenges and the possible solutions are:

- **Missing values**: A feature value may not be available for one or more instances. The solution is to replace missing values with the median (continuous) or the mode (most frequent category) (categorical) of the existing values. So we replace the missing value with a common one;
- **Sparsity**: Most of the instances contain just a small subset of the features: In other words, for a given data point only few features have a relative value while the others have no information at all. If we represent the data point as an n vector (n is the number of features) we would end up having a vector in which most of the values are 0 (so we waste a lot of space). A solution is to use "sparse-friendly" data structures (e.g. dictionary of keys where we only store values that are available);
- **Outliers**: One or more instances have out-of-range values for one or more features that could affect the accuracy of the model. The solutions are **retention** (we basically do nothing and consider them when creating the model) and **exclusion**. For what concerns the exclusion, we can cut them off using techniques such as **trimming** or we can limit extreme values in the statistical data to reduce the effect of possibly spurious outliers and this technique is called **winsoring**. So, in a trimmed estimator, the extreme values are discarded while in a winsorized estimator, the extreme values are instead replaced by certain percentiles (the trimmed minimum and maximum);
- **Mix of continuous and discrete values**: It may happen that the feature set contains both numerical and categorical values (e.g. number of bedrooms is a continuous value, while swimming pool can be a discrete (binary) value that tells us if the house has it or not). A possible solution is to transform categorical features using one-hot encoding (so, assuming to have c categories, we can substitute the categorical value with a c-dimensional vector that contains all zeros except where the category is, e.g. if a have 3 categories such as windy, cloudy and sunny and the value is "sunny" then we can substitute "sunny" with a vector [0, 0, 1]). The problem with this approach is sparsity, so again, we create vectors in which most of the values are 0. There are more advanced techniques to tackle this problem;
- **Multiple feature magnitudes**: Feature set contains a very wide range of values (this leads to giving a higher importance to features with a higher magnitude and we want features to have more or less the same weight). A solution is **standardization** in which we make features spanning the same values. Some standardization techniques are min-max or z-scores;
- **Class imbalance**: Instances labeled with the class of interest represents a tiny fraction of the total (e.g. we may have a dataset of emails that we can use to build a model that aims to understand if an email is spam or non-spam but most of the emails are non-spam). Some solutions are **oversampling** (we sample more with minority class samples, e.g. we add some spam emails), **undersampling** (we remove some of the majority class samples, e.g. we remove some non-spam emails) and **cost-sensitive learning** (we distinguish between different classes and give a different weight to errors depending on the class, e.g. we can assign a higher weight to errors regarding spam emails);
- **Strong multicollinearity**: There might be a linear relationship between one or more features so features that are highly correlated to each other and give more or less the same information and it is useless to keep both features. Two features are usually correlated when one increases/decreases the other increases/decreases (positive correlation) or when one increases the other decreases (negative correlation). A solution is to apply **dimensionality reduction (PCA)**.

Now, let's go into more details and give some notation. Let's assume that $X \subseteq R^n$ is the input feature space and Y is the output space. Now, we already know that if we have a regression problem then the output is a real number (so $Y \subseteq R^n$) while if we have a (k-ary) classification problem then the output is a discrete value (so Y = {1, .., k} where k is the number of classes). For what concerns the input space X, each instance is a labeled pair (x_i, y_i) where $x_i = (x_{i,1}, .., x_{i,n}) \in X$ the n-dimensional feature vector of the i-th instance and $y_i \in Y$ the label of the i-th instance. Putting all together we have that $D = \{(x_1, y_1), .., (x_m, y_m)\}$ is the dataset of m i.i.d. (independent and identically



distributed labeled instances). Again, each instance comes with the **class label (classification)** or the **value (regression)** we want to predict.

There is an **unknown target function** f which puts in a relationship elements of X with elements of Y. We don't know the joint probability distribution between any element of X with any element of Y but the intuition is that we can approximate the function f. So cannot write down an algorithm which just implements f because, if otherwise, we wouldn't need any data (ML is useless if we can come up with an algorithm that immediately gives us the precise output). Now, let's try to formalize it more: Learning f means "finding" **another function h*** which best approximates f using the data we observed where h* is chosen among a **family of functions H** called **hypothesis space** by specifying two components:

- **Loss function**: It measures the error of using h* instead of the true f;
- **Learning algorithm**: It explores the hypothesis space to pick the function which minimizes the loss on the observed data (basically h*).

So f is like the "perfect" function that is the one that always gives us the right output, but we know that we can't find f because we can't find such an algorithm: this means that we have to find an approximation. Now, in order to create a model, we train it using a dataset D for which we know the labels so we know the values given by the function f (only for that instances, not the new ones we have to predict!) and so we can actually compute the loss between the true output and the predicted one (the one found using our model). Let's assume that we have H that is a set of functions and we pick h* that is the best function in that set (the one that minimizes the loss on the observed data). So again, H is the set of functions the learning algorithm will search through to pick the **hypothesis h* which best approximates the true target f**. The larger the hypothesis space:

- The larger will be the set of functions that can be represented;
- The harder will be for the learning algorithm to pick h*.

So to keep the problem simple, we can apply some **trade-off** and put some constraints on H (e.g. limit the search space to linear functions, we may have some restrictions but choosing a smaller H it maybe would be easier to find the best approximation h* even with this limitation).

The **Loss Function** measures the error we would make if a hypothesis h (not necessarily h*, the best one) is used instead of the true (yet unknown) mapping f. As said before, the loss can be computed only on the data we observed, therefore it depends on the hypothesis and the dataset:

$$L : \mathcal{H} \times \mathcal{D} \mapsto \mathbb{R}$$

This **in-sample error** (a.k.a. empirical loss) is an estimate of the **out-of-sample error** (a.k.a. expected loss or risk). So, it is called the empirical loss because we compute from the dataset we observe and it is an estimate of the true expected loss we have for the new unknown instances (the instances for which we don't know the label and try to predict it). We have to find a way to minimize the (estimate of the) out-of-sample error.

The **Learning Algorithm** defines the strategy we use to search the hypothesis space H for picking our best hypothesis h*. Again, "best" means the hypothesis that minimizes the loss function on the observed data (**Empirical Risk Minimization**). In other words, among all the hypotheses specified by H the learning algorithm will pick the one that minimizes L:

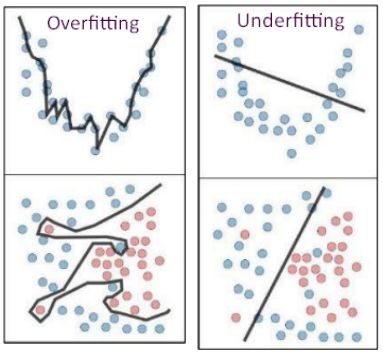
$$h^* = \text{argmin}_{h \in \mathcal{H}} L(h, \mathcal{D})$$

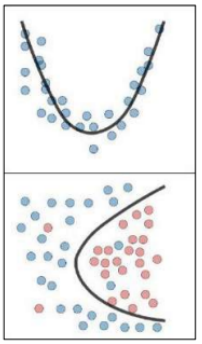
So, learning f is an optimization problem and this is done by plugging in different loss functions h combined with various hypothesis spaces we must solve a specific optimization problem. Those choices (the functions that we choose) are usually "mathematically convenient" (e.g. **convex objective functions** are guaranteed to have a unique global minimum while we may have different complex functions in which there are a lot of local minima and it is not guaranteed that we reach the one that is also the global and this mainly depends on the starting point). Remember that to understand if we have reached a minimum we have to look for the derivative of the loss function (the gradient) to be equal to 0. Even though closed-form solutions to the optimization problem rarely exist, there are numerical methods which work (e.g. **gradient descent**).
Minimizing the loss function on the observed data D (the labeled dataset for which we know the groundtruth) just limits the insample error (we have the instances from D, we train a model, we test it and we get the insample error so the error that we can actually quantify). Our ultimate hypothesis is to pick h* which is able to generalize to **unseen instances** (i.e. we want to minimize the out-of-sample error so the error that we could get when applying the model on unseen instances). If we pick a hypothesis which just memorizes all the training instances, we will obtain a 0 in-sample error but **this is not learning** (we lose generalization power so we may experience

overfitting) and, at the same time, we do not want h* to perform poorly on D (so have a large error on the testing set also called underfitting).

As already anticipated, 2 major problem of supervised learning are:

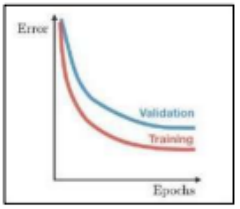
- **Overfitting**: The model is too complex and the residual error is low (or null) but this situation might bring strange deep curves in the cost function (as described by the image on the right). The hypothesis h* is not learning the true f but it mimics its noise. In order to understand if we are experiencing overfitting, we can notice that we may have a low in-sample (training) error and a high out-of-sample (validation/testing) error. Some solutions are:
    - Apply some regularization;
    - Get more data (the model is too complicated and a larger dataset might allow it to train it more adequately).
- **Underfitting**: The model is too simple and the residual error is too big. Formally, underfitting occurs when a mathematical model cannot adequately capture the underlying structure of the data. An under-fitted model is a model where some parameters or terms that would appear in a correctly specified model are missing. Under-fitting would occur, for example, when fitting a linear model to non-linear data. Such a model will tend to have poor predictive performance. So, the hypothesis h* is too "simple" for approximating the true f. In order to understand if we are experiencing overfitting, we can notice that we may have a high in-sample (training) error and a high out-of-sample (validation/testing) error. Some solutions are:
    - Increase the model complexity (e.g. choose a nonlinear function rather than a linear one);
    - Add more features.





What we should achieve is a trade-off between bias and variance with a low in-sample (training) error and a low out-of-sample (validation/testing) error: The hypothesis h* is just right in fact it is the simplest one explaining the data.
Measuring the generalization (i.e. out-of-sample) performance online may be too risky (e.g. We don't want to deploy a new spam classifier in production knowing only its training (i.e., in-sample) performance. The solution is to estimate the generalization performance using the training set (so again, we train the model and we test it to see how it performs). As long as it holds true the assumption that training and test instances are both drawn from **the same probability distribution** (**i.i.d. assumption**).

In order to do what we just said, it is necessary to **split the data**. In general what we do is starting from the original dataset D, we do a random shuffling and then divide it. There are few approaches:

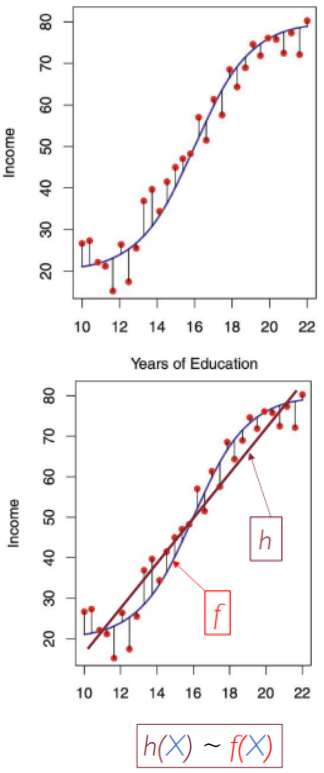


1. We can divide D in **training set** (about 80% of D) to learn h* and the **testing set** (the remaining 20%) to evaluate h*. The problem with this approach is that we don't know how the model will perform on new data;
2. We can divide D in **training set** (about 70% of D) to learn h*, the **validation set** to evaluate the model (about 10% of D) (the hyperparameter that we choose) and the **testing set** (the 20% of D) to evaluate h* (only once we find the best hyperparameters);
3. **K-Fold Cross Validation**: A generalization of the training/test splitting seen before. We have to pick a value for K (e.g., K=5 or 10) and divide the dataset D into K distinct folds. Then we perform K rounds where h* is learned from K-1 training folds and evaluated on 1 remaining test fold. The estimate of generalization error is the average across the K test folds of all the K rounds.

EX. Select which value of k = {2, 5, 10} of a kNN gives the best performance:

1. Train a separate model for each value of k on the training set (e.g., 70%);
2. Measure the error of each model on the validation set (e.g., 10%);
3. Select the model whose value of k gives the best performance on the validation set (e.g., k = 5);
4. Re-train only this model on the training + validation set;
5. Measure the performance on the test set (e.g., 20%).

6. Linear Regression



In order to introduce this topic let's assume that we have a set of observations (X, Y) where X is the input (e.g. years of education) and Y is the output (e.g. income) and we want to know what is the relationship between the two (e.g. how the yearly income changes depending on the years of education). In order to describe this relationship we can come out with a function Y=f(X)+ε. Now, f(X) is the function that best approximates the relationship between X and Y but we still have some error due to randomness/variation on the data so we have to consider the error ε (the error is not captured by f(x) so we have to add it).
So, f is some **fixed** but **unknown function** of X. **ε** is a **random error term**, which is independent of X and has **0-mean** (this means that if we consider the average error on all data, it is equal to 0). In this formulation, f represents the systematic information that X provides about Y (so f tells us the inherent relationship between X and Y). Our goal is to find an **approximation h** of the **true relationship f** by choosing it from a specific **hypothesis space H** (i.e. in this chapter we consider linear functions). In order to do this, we can start from a **dataset D of observations** that i.i.d.
The **hypothesis space H** is defined as follows:

$$\mathcal{H} = \{h_{\boldsymbol{\theta}} : \mathcal{X} \mapsto \mathcal{Y} \mid h_{\boldsymbol{\theta}}(\mathbf{x}) = \theta_0 x_0 + \theta_1 x_1 + \ldots + \theta_n x_n\}$$

So it is a set of functions that are parametrized by some coefficients Θ. The coefficients can be represented by an **n+1 dimensional vector** (e.g. in the case of one dimension and we only have a feature x_1 then we have the coefficient Θ_0 that represents the intersect with the y-axes and Θ_1 represents the slope). Since we use Θ_0 to represent the intersection with the y-axes, in order to stick with the formula we can multiply it for $\mathbf{x_0}$ **that can be set to 0 by convention**. In the case of a single dimension it is easy to visualize what happens but it is not straightforward with more dimensions, still the same idea applies.
Among all the possible instantiations of Θ the **learning algorithm selects Θ* as the one which minimizes a loss function measured on D**.
Note: When talking about linearity, we mean a linear combination in the parameters of the model so may combine features together (e.g. we could square them) and the function would be still linear because the linearity is not in the features but in the parameters (where the parameters/coefficients are Θ).

Now let's go one step ahead. We can define with $\mathbf{y_i = f(x_i)+\varepsilon_i}$ the i-th observation and with $\hat{\mathbf{y}}_\mathbf{i} = \mathbf{h_\Theta(x_i)}$ the **i-th prediction** (what our model predicts for the i-th observation) that basically approximates $f(x_i)$. The hypothesis $h_\Theta(x_i)$ on the i-th observation can be represented as:

$$\hat{y}_i = h_{\boldsymbol{\theta}}(\mathbf{x}_i) = \theta_0 x_{i,0} + \theta_1 x_{i,1} + \ldots + \theta_n x_{i,n}$$

That is no more than the linear combination seen before for the i-th observation. Now, we define the **error $\boldsymbol{\varepsilon_i}$** (the discrepancy between the i-th observation and the prediction on the i-th observation also called **residual**) with just the difference between y_i and $\hat{y}_i$:

$$e_i = \hat{y}_i - y_i = h_{\boldsymbol{\theta}}(\mathbf{x}_i) - \underbrace{y_i}_{f(\mathbf{x}_i)+\epsilon_i} \quad i\text{-th residual}$$

Now, if we consider all instances in our dataset, we can consider the total residual by computing the **Residual Sum of Squares (RSS)** (that tells us how good/bad out model is) as:

$$\mathrm{RSS}(h_{\boldsymbol{\theta}}, \mathcal{D}) = \sum_{i=1}^{m} e_i^2 = \sum_{i=1}^{m} (\hat{y}_i - y_i)^2 = \sum_{i=1}^{m} (h_{\boldsymbol{\theta}}(\mathbf{x}_i) - y_i)^2$$

Remember that the supervised learning problem can be generally defined as the following optimization problem:

$$h^* = \mathrm{argmin}_{h \in \mathcal{H}} L(h, \mathcal{D}) \quad h^* = h_{\boldsymbol{\theta}^*} = \mathrm{argmin}_{\boldsymbol{\theta}} L(h_{\boldsymbol{\theta}}, \mathcal{D})$$

Ordinary Least Square (OLS) is the usual approach to fit (i.e., find the optimal set of parameters of) linear regression models. So, in our case, h* corresponds to h_Θ* (when changing Θ, we also change h basically and our goal is to find the parameters so that we minimize the loss).

OLS uses **Mean Squared Error** as the loss function to perform the minimization. MSE measures the **average error when the true f is substituted with a hypothesis h_Θ in H** (in-sample error):

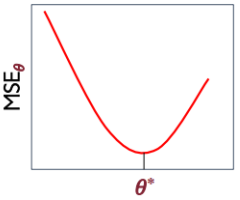
$$\text{MSE}(h_{\boldsymbol{\theta}}, \mathcal{D}) = \frac{1}{m}\text{RSS}(h_{\boldsymbol{\theta}}, \mathcal{D}) = \frac{1}{m}\sum_{i=1}^{m}(h_{\boldsymbol{\theta}}(\mathbf{x}_i) - y_i)^2$$

Again, we want to minimize the mean squared error so OLS aims at solving this optimization problem:

$$h^* = h_{\boldsymbol{\theta}^*} = \text{argmin}_{\boldsymbol{\theta}}\text{MSE}(h_{\boldsymbol{\theta}}, \mathcal{D}) = \text{argmin}_{\boldsymbol{\theta}}\left[\frac{1}{m}\sum_{i=1}^{m}(h_{\boldsymbol{\theta}}(\mathbf{x}_i) - y_i)^2\right]$$

We can prove that this function is **convex** and we know that any **local minimum (maximum) of a convex (concave) function** is also a **global minimum (maximum)**. If the function is convex (concave), finding the global minimum (maximum) can be done just by **computing the first derivative and setting it to 0.** In the case of a multivariate function (it a function of Θ that is an n-dimensional vector and not a scalar), this generalizes to compute the **gradient (∇)** of the function and set it to 0. The gradient of an n-variable function is the n-dimensional vector of the **partial derivatives of the function** w.r.t. each of its variable:

$$f : \mathbb{R}^n \mapsto \mathbb{R} \qquad \nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n}\right)$$



Solving ∇f = 0 means finding the n-dimensional vector x such that:

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n}\right) = \underbrace{(0, 0, \ldots, 0)}_{n} = \mathbf{0}$$

Now, let's go back to the initial problem. We want to minimize the mean squared error where, in the formula, we have that the observations y_i and features x_i can be thought of as fixed constants. Each term of the summation is a **multivariate linear function** of the model parameters Θ. **Linear functions are convex** and so is any sum of those and convex functions have a **unique local minimum**, which therefore happens to be the **global minimum**.

Now, let's try to compute the gradient of MSE:

$$\nabla\text{MSE}(h_{\boldsymbol{\theta}}, \mathcal{D}) = \nabla\left[\frac{1}{m}\sum_{i=1}^{m}(h_{\boldsymbol{\theta}}(\mathbf{x}_i) - y_i)^2\right]$$

$\frac{\partial}{\partial t}f(\alpha t) = \alpha\frac{\partial}{\partial t}f(t), \alpha \in \mathbb{R}\ (\text{constant})$ — scalar multiple rule

$\frac{\partial}{\partial t}f\left(\sum t\right) = \sum\left(\frac{\partial}{\partial t}f(t)\right)$ — sum rule

We can take the multiple scalar (1/m) out of the summation and also consider the fact that the derivative of a function that is a sum of terms is just the sum of the derivatives so we can take the gradient inside the sum:

$$\nabla\text{MSE}(h_{\boldsymbol{\theta}}, \mathcal{D}) = \frac{1}{m}\left[\sum_{i=1}^{m}\nabla(h_{\boldsymbol{\theta}}(\mathbf{x}_i) - y_i)^2\right]$$

What we did so far is just rewrite the equation above using some rules. Now, in order to understand what happens we can assume that the dataset D contains a single instance (x, y). Here the error is easy to compute since is just the difference between the hypothesis and the real value squared:

$$\nabla\text{MSE}(h_{\boldsymbol{\theta}}, \mathcal{D}) = \nabla(h_{\boldsymbol{\theta}}(\mathbf{x}) - y)^2$$

$\frac{\partial}{\partial t}t^n = nt^{n-1},\ n \in \mathbb{N}$ — power rule

$\frac{\partial}{\partial t}f(g(t)) = \frac{\partial}{\partial g(t)}f(g(t)) * \frac{\partial}{\partial t}g(t)$ — chain rule

Let's again try to transform the equation by applying the power rule and the chain rule altogether and get:

$$2(h_{\boldsymbol{\theta}}(\mathbf{x}) - y)\nabla(h_{\boldsymbol{\theta}}(\mathbf{x}) - y)$$

Now, we want to find the gradient but, how can we do it? Let's expand it:

$$\nabla(h_{\boldsymbol{\theta}}(\mathbf{x}) - y) = \nabla(\theta_0 x_0 + \theta_1 x_1 + \ldots + \theta_n x_n - y) =$$

So the gradient is just the linear combination of all the parameters minus y. Now, every component can be seen as the partial derivatives of all the components with respect to each parameters Θ. Now, when computing the j-th partial derivative, the Θ's (except for Θ_j) are constants so we do only care about Θ_j. This means that for the derivative perspective, the j-th partial derivative is no more than the j-th coefficient (the one that multiply Θ_j) so end up having the n+1 dimensional vector x = (x_0, x_1, .., x_n):

$$= \underbrace{\left(\frac{\partial(\theta_0 x_0 + \theta_1 x_1 + \ldots + \theta_n x_n - y)}{\partial \theta_0}, \ldots, \frac{\partial(\theta_0 x_0 + \theta_1 x_1 + \ldots + \theta_n x_n - y)}{\partial \theta_n} \right)}_{n+1} = (x_0, x_1, \ldots, x_n) = \mathbf{x}$$

So if we plug in x (the gradient that we solved) in the previous formula, we get:

$$\nabla \text{MSE}(h_{\boldsymbol{\theta}}, \mathcal{D}) = \underbrace{2(h_{\boldsymbol{\theta}}(\mathbf{x}) - y)}_{\text{scalar}} \cdot \underbrace{\mathbf{x}}_{(n+1)\text{-dimensional vector}}$$

Now, the left part is just a scalar (we know all those values that are no more than the difference between the hypothesis and the real value multiplied by 2) while the second part is our vector x. Now, the hypothesis $h_\Theta(x)$ can be rewritten as the linear combination we have seen before and, that is $\boldsymbol{\Theta}^T\mathbf{x}$. This means that the formula above can be synthetized in:

$$\nabla \text{MSE}(h_{\boldsymbol{\theta}}, \mathcal{D}) = 2(\boldsymbol{\theta}^T \cdot \mathbf{x} - y)\mathbf{x}$$

and expanding the equation we get (in the right part we simply remove x_0 since it is equal to 1 by convention):

$$= \begin{bmatrix} 2(\boldsymbol{\theta}^T \cdot \mathbf{x} - y)x_0 \\ 2(\boldsymbol{\theta}^T \cdot \mathbf{x} - y)x_1 \\ \vdots \\ 2(\boldsymbol{\theta}^T \cdot \mathbf{x} - y)x_n \end{bmatrix} = \begin{bmatrix} 2(\boldsymbol{\theta}^T \cdot \mathbf{x} - y) \\ 2(\boldsymbol{\theta}^T \cdot \mathbf{x} - y)x_1 \\ \vdots \\ 2(\boldsymbol{\theta}^T \cdot \mathbf{x} - y)x_n \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \mathbf{0} \qquad 2(\boldsymbol{\theta}^T \cdot \mathbf{x} - y)x_j = 0 \ \forall j \in \{0, 1, \ldots, n\}$$

The resulting gradient is an (n+1)-dimensional vector as expected and we want it to be equal to 0 (we would like the error to be equal to 0). So we basically need to solve a system of n+1 linear equations with n+1 variables.

Now, so far we have seen the simple case in which we only consider one instance in the dataset D. In **the general case where the dataset D contains a m instances** we have:

$$\nabla \text{MSE}(h_{\boldsymbol{\theta}}, \mathcal{D}) = \frac{2}{m}\left[\sum_{i=1}^{m} \Big(h_{\boldsymbol{\theta}}(\mathbf{x}_i) - y_i \Big) \nabla \Big(h_{\boldsymbol{\theta}}(\mathbf{x}_i) - y_i \Big) \right]$$

$$\nabla \text{MSE}(h_{\boldsymbol{\theta}}, \mathcal{D}) = \frac{2}{m}\left[\sum_{i=1}^{m} \underbrace{\Big(h_{\boldsymbol{\theta}}(\mathbf{x}_i) - y_i \Big)}_{\text{scalar}} \underbrace{\mathbf{x}_i}_{n+1\text{-dimensional vector}} \right]$$

It is the same but, instead of looking at a single datapoint, each component of the gradient vector has to **take into account all the m data points**. So, if we expand the equation we have:

$$\nabla \text{MSE}(h_{\boldsymbol{\theta}}, \mathcal{D}) = \begin{bmatrix} \frac{2}{m}(\boldsymbol{\theta}^T \cdot \mathbf{x}_1 - y_1)x_{1,0} + \ldots + \frac{2}{m}(\boldsymbol{\theta}^T \cdot \mathbf{x}_m - y_m)x_{m,0} \\ \frac{2}{m}(\boldsymbol{\theta}^T \cdot \mathbf{x}_1 - y_1)x_{1,1} + \ldots + \frac{2}{m}(\boldsymbol{\theta}^T \cdot \mathbf{x}_m - y_m)x_{m,1} \\ \vdots \\ \frac{2}{m}(\boldsymbol{\theta}^T \cdot \mathbf{x}_1 - y_1)x_{1,n} + \ldots + \frac{2}{m}(\boldsymbol{\theta}^T \cdot \mathbf{x}_m - y_m)x_{m,n} \end{bmatrix}$$

Note: Again, we can remove $x_{i,0}$ since it is equal to 1 by convention.
Again, we need to solve a system of n+1 linear equations with n+1 variables:

$$\frac{2}{m}\Big[(\boldsymbol{\theta}^T \cdot \mathbf{x}_1 - y_1)x_{1,j} + \ldots + (\boldsymbol{\theta}^T \cdot \mathbf{x}_m - y_m)x_{m,j} \Big] = 0 \ \forall j \in \{0, \ldots, n\}$$

Now, we can write a **vectorized form of the gradient of MSE** that enables us to compute it in a more straightforward way. First all let's write again the data that we have in a vectorized form:

$$\mathbf{X} = \underbrace{\begin{bmatrix} x_{1,0} & x_{1,1} & \ldots & x_{1,n} \\ x_{2,0} & x_{2,1} & \ldots & x_{2,n} \\ \vdots & \vdots & \vdots & \vdots \\ x_{m,0} & x_{m,1} & \ldots & x_{m,n} \end{bmatrix}}_{m \times n+1 \textbf{ feature matrix}} = \begin{bmatrix} -\mathbf{x}_1^T- \\ -\mathbf{x}_2^T- \\ \vdots \\ -\mathbf{x}_m^T- \end{bmatrix} \quad \mathbf{y} = \underbrace{\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}}_{m\text{-dimensional } \textbf{target vector}} \quad \boldsymbol{\theta} = \underbrace{\begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_n \end{bmatrix}}_{n+1\text{-dimensional } \textbf{parameter vector}}$$

X has m rows that correspond to the number of data points and each data point has n features (+1 because of the intercept). For each data point we have an observation that described by the vector y (that is m-dimensional) and finally we have Θ that is an n+1 dimensional vector (one Θ for each feature).
Using this vectorial representation of the data, the MSE can be rewritten as:

$$h^* = h_{\theta^*} = \text{argmin}_{\boldsymbol{\theta}} \left[\underbrace{\frac{1}{m} ||\mathbf{X} \cdot \boldsymbol{\theta} - \mathbf{y}||^2}_{\text{MSE}(h_{\boldsymbol{\theta}}, \mathcal{D})} \right]$$

Again, we want to minimize the MSE so we have to compute the gradient. We don't see all the steps but, computing the gradient of the MSE we get:

$$\nabla \text{MSE}(h_{\boldsymbol{\theta}}, \mathcal{D}) = \frac{2}{m} \mathbf{X}^T (\mathbf{X} \cdot \boldsymbol{\theta} - \mathbf{y}) \qquad \frac{2}{m} \mathbf{X}^T (\mathbf{X} \cdot \boldsymbol{\theta} - \mathbf{y}) = \mathbf{0}$$

We want it to be equal to 0. Since 2/m is a constant we can remove it (we don't care about it). Let's now resolve the equation by multiplying the elements inside brackets and putting one of the elements on the right:

$$\mathbf{X}^T \mathbf{X} \cdot \boldsymbol{\theta} = \mathbf{X}^T \cdot \mathbf{y}$$

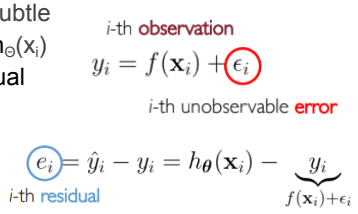
We want to find Θ so we can multiply both side for the inverse of X^TX. We can define **$\Theta = X^ty$** and:

$$\mathbf{X}^\dagger = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \text{ is the } \textbf{pseudo-inverse} \text{ of } \mathbf{X}$$

So in order to be able to compute the best optimal parameters we can simply use this formula and come up with a solution (there is no need to use methods like gradient descent that use iteration) and this solution is called **one-step** (or one-shot) **learning**. The problem is that, in general, the feature matrix X is non-squared therefore **non-invertible**. **X^TX** is instead square (n-by-n) and **very likely invertible**. The chance of a randomly generated squared matrix is invertible approaches 1 and **to be non-invertible**, **the determinant must be 0** (linearly dependent columns). Typically, the number m of rows (instances) are way larger than the number n of columns (features) so X^TX (the resulting matrix) is smaller than X.

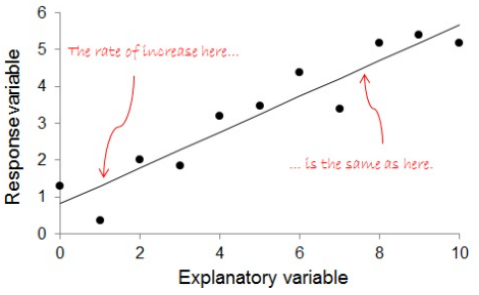
OLS is also known as **one-step learning** as there exists a closed-form (i.e. analytical) solution to the convex optimization problem. However, other choices of loss functions (even if convex) may need an iterative approach to get to a (local) minimum. Though in general n << m, computing the inverse of an n-by-n matrix is still a costly operation ($O(n^3)$ time complexity that can be reduced to $O(n^{2.376})$ using the **Coppersmith-Winograd algorithm**).

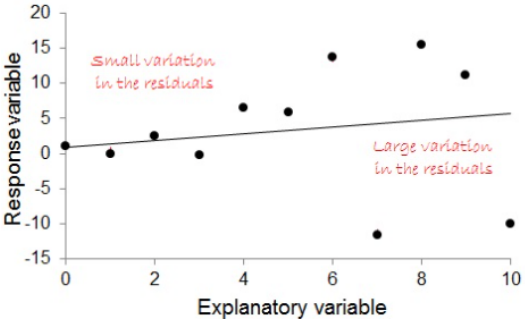
Now, so far we talked about error and residual like it is the same thing. Actually there is a subtle **difference between errors and residuals**. Given $y_i = f(x_i)+\varepsilon_i$ the i-th observation and $\hat{y}_i = h_\Theta(x_i) \sim f(x_i)$ the i-th prediction, we can define **ε_i as the i-th unobservable error**. The e_i i-th residual can be computed as the difference between $\hat{y}_i$ and y_i (the prediction that we make and the observation) so it is like in the residual we also consider the unobservable error that is included in the observation. **MSE is computed from residuals**, not unobservable errors so when computing MSE we are trying to reduce the residual on each prediction.



There are some basic assumptions of linear regression:

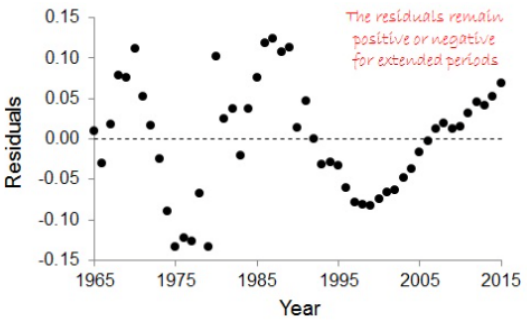
- **Weak exogeneity**: Features can be treated as error-free constants. The set of data points that we have are random variables mainly because they are generated from a process that we don't know so the are treated like they are error-free constants;
- **Linearity**: There is a linear relationship between the features and the response. For each unit increase in the explanatory variable (x), the mean of the response variable (y) increases by the same amount, regardless of the value of the explanatory variables;
- **Normality**: We may think (erroneously) that the normal distribution assumption of linear regression applies to their data. We might plot their response variable as a histogram and examine whether it differs from a normal distribution. Another assumption is that the explanatory variable must be normally-distributed. Neither is required. The normality assumption relates to the distributions of the residuals. This is assumed to be normally distributed, and the regression line is fitted to the data such that the mean of the residuals is zero. To examine whether the residuals are normally distributed, we can compare them to what would be expected and this can be done by binning the values in classes and examining a histogram, or by constructing a kernel density plot. In most of the





cases the explanatory and response variables are far from normally distributed – they are much closer to a uniform distribution (in fact the explanatory variable conforms exactly to a uniform distribution);

- **Equal variance**: Linear regression assumes that the variance of the residuals is the same regardless of the value of the response or explanatory variables – the issue of **homoscedasticity**. If the variance of the residuals varies, they are said to be heteroscedastic. The residuals in the image above are not heteroscedastic. If they were, they might look more like the example in the second image.

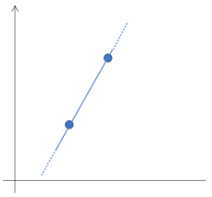


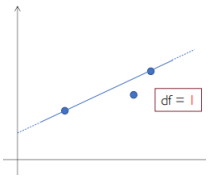
- **Independence**: The **residuals should be independent** of each other. In particular, it is worth checking for serial correlation. Correlation is evident if the residuals have patterns where they remain positive or negative for a certain amount of data points. In our first example, the residuals seem to randomly switch between positive and negative values – there are not disproportionately long runs of positive or negative values.

One important topic regarding every ML model is how to **assess its quality**. Suppose we have fit a linear regression model to some dataset of observations $D = \{(x_i, y_i)\}_{i=1..m}$ (in other words, we estimated the vector of parameters Θ^* using OLS). In order to measure how good our model there are various methods and two of them are:

- **Residual Standard Error (RSE)**: Recall that every observation of the target variable y_i is associated with an error term ε_i. Even if we were able to find the exact parameters of the true f, we would not be able to perfectly predict y_i from x_i. **RSE is an estimate of the standard deviation of ε**. The formula to compute the RSE is the following:

$$\mathrm{RSE}(h_{\boldsymbol{\theta}}, \mathcal{D}) = \sqrt{\frac{1}{\underbrace{m-n-1}_{\text{degrees of freedom}}} \underbrace{\sum_{i=1}^{m}(\hat{y}_i - y_i)^2}_{\text{RSS}}}$$



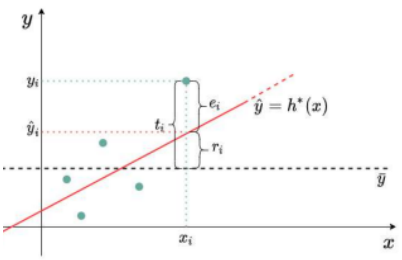


We simply take the Sum of Squared Error and multiply it by the degree of freedom. The lower RSE, the better. Now, one important component in this formula is the **degree of freedom**. Now, let's assume we are in the 1-D case: It is easy to notice that we need at least 2 data points to be able to fit a perfect line (we can't obviously do with only one point). The problem is that my fitted line may drastically change depending on where the second point is located. If we want my model to be more "flexible" we need at least 3 points which leave us with 1 degree of freedom. If we add another point (remember that we are always in the 1-D case) we have a degree of freedom of 2.
Now, Given the same number of observations m, **we lose 1 degree of freedom for each variable we add**. So, the formula to compute the degree of freedom is:

$$\mathrm{df} = \underbrace{m}_{\#\text{observations}} - \underbrace{n}_{\#\text{features}} - \underbrace{1}_{\text{intercept}}$$

The higher degree of freedom, the better. This is another way of looking at overfitting: If we have too many features it is likely that the model is too complex so we can both reduce the number features (e.g. using feature selection) or adding more data points;

- **R^2 Statistic**: Let's suppose that we have the plot on the right and we fit that regression line. The horizontal line represents the average value of the response that is the expectation of y. For each data point we can calculate t_i that is the total variance of the data point (so given the observations, we compute the distance between the observation and the average value and make square the result): We compute the sum of all t_i^2 and this is the **TSS**. Then, for each data point we can also compute the error e_i that is basically the squared distance between the observation and prediction (as we know): We compute the sum of all e_i^2 and this is the **RSS**. **R^2** is computed like:



$$t_i = y_i - \bar{y}$$
$$e_i = y_i - \hat{y}_i$$
$$r_i = \hat{y}_i - \bar{y}$$

$$TSS = \sum_{i=1}^{m}(y_i - \bar{y})^2 = \sum_{i=1}^{m} t_i^2$$
$$RSS = \sum_{i=1}^{m}(y_i - \hat{y}_i)^2 = \sum_{i=1}^{m} e_i^2$$

$$R^2 = 1 - \frac{RSS}{TSS} = \frac{TSS - RSS}{TSS} = 1 - \frac{\sum_{i=1}^{m}(\hat{y}_i - y_i)^2}{\sum_{i=1}^{m}(y_i - \bar{y})^2}$$

This tells us how much variance is captured by the regression line that we fit over the total. To sum up, **TSS measures the total variance in the response** Y before the regression takes place. **RSS measures the amount of variability that is left unexplained** after performing the regression. **R^2 measures the proportion of variability in Y that can be explained using X**. An R^2 statistic that is close to 1 indicates that a large proportion of the variability in the response has been explained by the regression so, **the larger the better**.
Note: R^2 is easier to interpret than RSE as it always ranges between 0 and 1. Fixing the sample size m, RSS decreases (or, at worst, it stays the same) as more variables are added to the fitted model. R^2 always increases as more variables are added (as df decreases) and this a one limitation of R^2. We need a way to adjust for that, otherwise we could get a better model by simply adding useless features to it and this can be done by slightly modifying the formula:

$$R^2_{\text{adj}} = 1 - \frac{\frac{\text{RSS}}{m-n-1}}{\frac{\text{TSS}}{m-1}}$$

Maximizing the adjusted R^2 is equivalent to minimizing RSS/(m-n-1). We know RSS may decrease if the number of variables in the model increases. RSS/(m-n-1) may increase or decrease, due to the presence of n in the denominator. We may need to increase the sample size m to compensate for the increasing RSS due to the inclusion of more features n.

Another important topic in ML (also valid for Linear Regression) is Regularization. The absolute value of learned parameters Θ should not be very large, otherwise, a small change in an input feature may cause a high difference in the output predicted value. This is an indication of **overfitting**: The learned model is highly "training set dependent" and does not generalize. **Regularization** is about putting some constraint on the **optimization problem so as to limit the values of the learned parameters**. We consider a far more general optimization framework called **Elastic Net Framework**, which OLS is just a special case of:

$$\boldsymbol{\theta}^* = \text{argmin}_{\boldsymbol{\theta}} \left[\frac{1}{m} ||\mathbf{X} \cdot \boldsymbol{\theta} - \mathbf{y}||^2 + \lambda \Big(\alpha |\boldsymbol{\theta}| + (1-\alpha) ||\boldsymbol{\theta}||^2 \Big) \right]$$

We have now different cases:

- λ ≥ 0 (**regularization parameter**): When λ = 0 we go back to OLS (no regularization at all) since we remove completely the right part;
- α ∈ [0, 1] (**tradeoff parameter**): α is used to weight the regularization penalties;
- λ > 0; α = 1 (**Least Absolute Shrinkage and Selection Operator** or **LASSO**): L1-regularization only;
- λ > 0; α = 0 (**Ridge**): L2-regularization only;

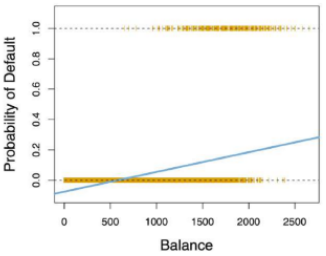
Depending on the kind of regularization we decide to apply, we implicitly constraint the values of Θ to be small enough.

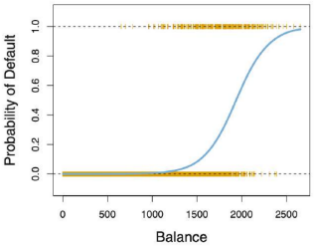
7. Logistic Regression

Very often, the response variable to predict is **qualitative** (categorical). **Classification** (as opposed to regression) deals with predicting **categorical responses**. Classification methods **may first predict the probability of each category** of a qualitative response to make in turn a decision.
EX. Some examples of categorical prediction are the spam vs. non spam emails or probability to click vs. non-click on a web page or an advertisement.

Why can't we deal with a classification problem as a regression one?
Suppose we want to predict the health condition of a patient arriving in the ER on the basis of her symptoms. Imagine there are only the following 3 possible diagnoses: stroke, drug overdose, and epileptic seizure. We may encode the above values as a categorical response variable Y so: Y=1 if stroke, Y=2 if drug overdose and Y=3 is epileptic seizure. With the previous encoding we can fit a linear regression model using OLS from a set of n features x_1, ..., x_n but this type of coding would imply an **ordering of the outcomes**, in fact: Drug overdose is in between stroke and epileptic seizure and the difference between stroke and drug overdose is the same as the difference between drug overdose and epileptic seizure. In practice, there is no particular reason to choose the encoding above and **different** (and still legitimate) **encodings will produce different models**.

Note: The encoding above would work if the response variable values take on an **equally-spaced natural ordering** (e.g., mild, moderate, and severe - there is a natural ordering between them) but this is a rare case and in general (most of the cases) there is no natural way to convert a K-ary (K > 2) response into a quantitative response that is ready for the linear regression framework we saw in the last chapter. For a binary response with a 0/1 encoding (so only 2 values), linear regression by OLS does anyway make sense because we would be able to predict 1 if the outcome is > 0.5, 0 otherwise. In any case, it is still preferable to use a classification method which works by design.





So, in order to introduce the topic let's consider a binary response Default(Y) taking on two values: Yes or No. Basically it tells us if a certain customer of a bank is in default or not. Suppose we want to predict the value of Y from the value of Balance(X) (the balance of his bank account). We can model it directly via linear regression (i.e., predicting its value) but we can use **Logistic Regression** instead, which **models the probability that Y belongs to one of the two possible outcome values**. With linear regression (first image) we can fit a line to understand if the customer will default or not: since Y can be only 0 or 1 we have that all the points lie in 2 horizontal lines. We fit a line that tries to minimize the error but, since the range of linear regression is the whole real line, it may happen that some estimated probabilities are negative. In the case of Logistic Regression we model the probability like a sigmoid distribution and all probabilities lie between 0 and 1.

In Logistic Regression, as well as other ML models, we have to specify:

- The model: It defines the space of representable hypotheses;
- The error measure (cost function): It measures the price of misclassification errors;
- The learning algorithm: It picks the best hypothesis exploring search space.

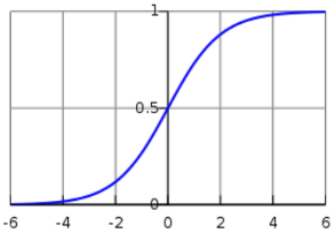
Let's start with the **model**. Let $x = (x_0, x_1, ..., x_d)$ be the (d+1)-dimensional input (the set of data points). Let F be the family of real-valued functions parametrized by θ so that $\theta^T = (\theta_0, \theta_1, ..., \theta_d)$:

$$\mathcal{F} = \{f_\theta : \mathbb{R}^{d+1} \longmapsto \mathbb{R} \mid f_\theta(\mathbf{x}) = \boldsymbol{\theta}^T\mathbf{x} = \sum_{i=0}^{d} \theta_i x_i\}$$

In other words F is the set of functions that are parametrized by θ that take the d+1 points in that space and map them to a real value. The definition is that the function f parametrized by θ of x is just the linear combination of θ with x that is, again, the sum of the products of the vector θ with the vector x. Each function in F outputs a real number (i.e. a scalar) as a linear combination of the input x with the parameters θ. The function $\mathbf{f_\theta(x)}$ is also called a **(linear) signal**. The signal alone is not enough to define the hypothesis space H. Usually the signal is passed through a "**filter**", i.e., another real-valued function g (it is like a composition), so we have:

$$\mathcal{H} = \{h_\theta : \mathbb{R}^{d+1} \longmapsto \mathbb{R} \mid h_\theta(\mathbf{x}) = g(f_\theta(\mathbf{x})) = g(\boldsymbol{\theta}^T\mathbf{x}) = g\big(\textstyle\sum_{i=0}^{d} \theta_i x_i\big)\}$$

$\mathbf{h_\theta(x) = g(f_\theta(x))}$ is our hypothesis space. Clearly, the set of possible hypotheses H changes depending on the parametric model f_θ and on the **thresholding function g.** We have different threshold functions and one of them is the **logistic function** (also called **sigmoid function**). The z in the sigmoid formula is basically the output of the function $f_\theta(x)$ that is $\theta^T x$. When $z = \theta^T x$ we are applying a **non-linear transformation** to our linear signal. This means that the output can be genuinely interpreted as a probability value.
Let's rewrite the whole formula by injecting the sigmoid function and we get:



$$h_\theta(\mathbf{x}) = \ell(f_\theta(\mathbf{x})) = \ell(\boldsymbol{\theta}^T\mathbf{x}) = \frac{e^{(\boldsymbol{\theta}^T\mathbf{x})}}{1+e^{(\boldsymbol{\theta}^T\mathbf{x})}} \qquad \ell(z) = \frac{e^z}{1+e^z}$$

Note: Describing the set of hypotheses using the logistic function is not enough to state that the output can be interpreted as a probability. All we know is that the logistic function always produces a real value between 0 and 1. Other functions may have the same property (e.g. $1/\pi \arctan(x) + ½$). The key points here are that the output of the logistic function can be interpreted as a probability even during learning and, moreover, the logistic function is mathematically convenient.

Now, let's introduce some concepts to understand why what we said so far is related to probability. Let **p** be the probability of success of an event and **q=1-p** the probability of failure. We can define the **odds** of success as odds(success) = p/1 = p/(1-p). Respectively we compute the odds of failure as odds(failure) =q/p = 1/(p/q) = 1/odds(success). If we take the natural logarithm of the odds of success we get the **logit**:
logit(p) = ln(odds(success)) = ln(p/q) = ln(p/1-p) = ln(p) - ln(1-p)
Notice that Logistic Regression is in fact an ordinary linear regression where the logit is the response variable:

$$\text{logit}(p) = \ln\left(\frac{p}{1-p}\right) = \theta_0 + \theta_1 x_1 + \ldots + \theta_d x_d = \boldsymbol{\theta}^T \mathbf{x}$$

So, instead of predicting the probability of the response variable, which may not be the case for linear regression because it takes on the whole real line of numbers. We say that our response variable indeed is the login of the probability: This response is the linear combination of our parameters θ and the features. **The coefficients of logistic regression are expressed in terms of the natural logarithm of odds**. Odds are defined on the range [0, +inf]. Logits (i.e., natural log of odds) are defined on the range [-inf, +inf]. Therefore we can use the "standard" regression equation above. For any value of the regression coefficients and features a valid value for the odds are predicted. On the contrary, probabilities are only defined on the range [0, 1] so it would need very complicated constraints on the regression coefficients to work with probability. Now, starting for the formula above, the try to find out how we can get the probability of success:

$$e^{\text{logit}(p)} = e^{\ln\left(\frac{p}{1-p}\right)} = \frac{p}{1-p} = e^{(\boldsymbol{\theta}^T \mathbf{x})}$$

So we start from the equation above and we put all the terms in the exponential (both right and left part of the equation). Notice that the logarithm cancels out.

$$p = e^{(\boldsymbol{\theta}^T \mathbf{x})}(1-p) = e^{(\boldsymbol{\theta}^T \mathbf{x})} - e^{(\boldsymbol{\theta}^T \mathbf{x})} p$$

Now we simply highlight p by multiplying right and left terms by (1-p) then we multiply the 2 terms.

$$p + e^{(\boldsymbol{\theta}^T \mathbf{x})} p = e^{(\boldsymbol{\theta}^T \mathbf{x})} \qquad p(1 + e^{(\boldsymbol{\theta}^T \mathbf{x})}) = e^{(\boldsymbol{\theta}^T \mathbf{x})}$$

Let's move the second term on the right to the left and collect the value p. Finally, let's highlight p we we get:

$$p = \frac{e^{(\boldsymbol{\theta}^T \mathbf{x})}}{1 + e^{(\boldsymbol{\theta}^T \mathbf{x})}} = \frac{1}{e^{-(\boldsymbol{\theta}^T \mathbf{x})} + 1}$$

Using (log) odds rather than actual probabilities provides an easier interpretation of the model's coefficients learned:

$$\ln\left(\frac{p}{1-p}\right) = \theta_0 + \theta_1 x_1 + \ldots + \theta_d x_d = \boldsymbol{\theta}^T \mathbf{x}$$

$$\left(\frac{p}{1-p}\right) = e^{\theta_0 + \theta_1 x_1 + \ldots + \theta_d x_d} = e^{\boldsymbol{\theta}^T \mathbf{x}}$$

$$\mathbf{x} = (x_1, \ldots, x_i, \ldots, x_d)$$
$$\mathbf{x}' = (x_1, \ldots, x_i + 1, \ldots, x_d)$$

x' is just the same as x where the *i*-th predictor/feature is increased by 1 unit

Suppose we want to measure the effect of a unit increase in one of the predictors to the output response. So, we want to measure the impact of increasing by one unit a feature on the output. Let's measure the ratio between the odds computed at a certain input x and the odds computed at a different point x' (the one in which we changed the feature value). More formally, if we assume x to have a value x then we simply get x' by increasing the value of the feature x_i by 1. The **odds ratio is** simply the ratio between the odds obtained by using x' and the one obtained by using x.

$$\frac{e^{\boldsymbol{\theta}^T \mathbf{x}'}}{e^{\boldsymbol{\theta}^T \mathbf{x}}}$$
Odds Ratio

Let's try to expand it:

$$\frac{e^{\boldsymbol{\theta}^T \mathbf{x}'}}{e^{\boldsymbol{\theta}^T \mathbf{x}}} = \frac{e^{\theta_0 + \theta_1 x_1 + \ldots + \theta_i (x_i + 1) + \ldots + \theta_d x_d}}{e^{\theta_0 + \theta_1 x_1 + \ldots + \theta_i x_i + \ldots + \theta_d x_d}} = \frac{e^{\theta_0 + \theta_1 x_1 + \ldots + \theta_i x_i + \ldots + \theta_d x_d} * e^{\theta_i}}{e^{\theta_0 + \theta_1 x_1 + \ldots + \theta_i x_i + \ldots + \theta_d x_d}} = e^{\theta_i}$$

We simply write it using the definition above. In the third term we simply take out the "+1" in the middle of the formula. Finally we simplify the formula and remain with $\mathbf{e^{\theta_i}}$ that is the ratio of the odds for 1-unit increase in x_i. In other words $\boldsymbol{\theta_i}$ **is the ratio of the natural log(odds) for 1-unit increase in** $\mathbf{x_i}$. This **ratio is constant** in fact it does not change according to the value of the other x_j because they cancel out in the calculation. Notice that if

we used probability rather than odds this wouldn't be constant in fact the effect of x_i on the probability of success p is different depending on the value of x_i.
EX. An odds ratio of 1.08 will give an 8% increase in the odds at any value of x_i.

Now, as with any other supervised learning problem we are given a finite set D of m i.i.d. labeled examples which we can try to learn from, and each y_i is a binary variable taking on two values instead of a real one (e.g. {-1,+1}):

$$\mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)\}$$

That means we **do not have access to the individual probability** associated with each training sample (so, again, we can't observe the actual probability of default but we only understand from historical observation whether a certain instance has a default equal 1 or a default equal 0). The data we observe from D is actually **generated by an underlying and unknown probability function** (**noisy target**) which we want to estimate:

$$P(y|\mathbf{x}) = \begin{cases} \phi(\mathbf{x}) & \text{if } y = +1 \\ 1 - \phi(\mathbf{x}) & \text{if } y = -1 \end{cases}$$

Now, let's focus on the difference between:

- **Deterministic function**: Given x as input it always outputs either y = +1 or y = -1 (mutually exclusive);
- **Noisy target function**: given x as input it always outputs both y = +1 and y = -1, each with a "degree of certainty" associated.

The goal is to find an **estimate φ* which best approximates φ** where φ: $R^{d+1} \rightarrow [0,1]$ is the unknown noisy target which generates our examples (so can substitute φ with φ* in the previous formula). We claim that the best estimate φ* of φ is **$h^*_\theta(x)$**, which in turn is picked from the set of hypotheses defined by logistic function (so the approximation of φ is picked from the logistic function applied to the linear signal):

$$\phi^*(\mathbf{x}) = h^*_{\boldsymbol{\theta}}(\mathbf{x}) = \ell(\boldsymbol{\theta}^T \mathbf{x}) \approx \phi(\mathbf{x})$$

Now, in order to **approximate $h^*_\theta(x)$** we can use the same general framework introduced for the supervised learning problem. We already fixed the set of hypothesis functions to select from and still need to set a training set D and an error measure (cost function) to minimize.

Let's start with the **cost function**. To **find the best hypothesis**, we would like to find the posterior hypothesis so the probability of getting an hypothesis observing a specific data D. We can split it in 3 components using the **Bayes rule** and it becomes:

$$\overbrace{P(h_{\boldsymbol{\theta}} \mid \mathcal{D})}^{\text{posterior}} = \frac{\overbrace{P(\mathcal{D} \mid h_{\boldsymbol{\theta}})}^{\text{likelihood}} \times \overbrace{P(h_{\boldsymbol{\theta}})}^{\text{prior}}}{\underbrace{P(\mathcal{D})}_{\text{evidence}}}$$

Starting from here, there are two main ways to find the estimate of the best hypothesis parameters θ*:

- **Maximum Likelihood Estimate (MLE)** or frequentist approach;
- **Maximum A Posteriori (MAP)** or bayesian approach.

MLE returns the set of parameters that **maximize the likelihood**:

$$h^*_{\boldsymbol{\theta}} = h^{\text{MLE}}_{\boldsymbol{\theta}} = \texttt{argmax}_{h_{\boldsymbol{\theta}} \in \mathcal{H}} P(\mathcal{D} \mid h_{\boldsymbol{\theta}})$$

So it is about picking the parameters that best explain the data.
MAP returns the set of parameters that **maximize the posterior** (the posterior is the whole numerator):

$$h^*_{\boldsymbol{\theta}} = h^{\text{MAP}}_{\boldsymbol{\theta}} = \texttt{argmax}_{h_{\boldsymbol{\theta}} \in \mathcal{H}} P(h_{\boldsymbol{\theta}} \mid \mathcal{D}) = \texttt{argmax}_{h_{\boldsymbol{\theta}} \in \mathcal{H}} \frac{P(\mathcal{D} \mid h_{\boldsymbol{\theta}}) \times P(h_{\boldsymbol{\theta}})}{P(\mathcal{D})} = \texttt{argmax}_{h_{\boldsymbol{\theta}} \in \mathcal{H}} P(\mathcal{D} \mid h_{\boldsymbol{\theta}}) \times P(h_{\boldsymbol{\theta}})$$

MLE is just a special case of MAP where **priors are uniform** (i.e. every hypothesis is equiprobable). Both MLE and MAP are **point estimators**: they return a single value for the optimal parameter vector θ*.
Note: A full Bayesian estimation is also possible, where the **full posterior distribution** (i.e. probability density/mass function) is estimated, although this turns out to be often **computationally intractable**.

Now, let's focus on the **MLE** that is about maximizing the **likelihood function**. We measure the error we are making by assuming that $h^*_\theta(x)$ approximates the true noisy target φ. The question is the following: How likely is that the observed data D have been generated by our selected hypothesis $h^*_\theta(x)$?
We have to find the hypothesis which maximizes the probability of the observed data D given a particular hypothesis:

$$h^*_{\theta} = \text{argmax}_{h_{\theta} \in \mathcal{H}} \; P(\mathcal{D} \mid h_{\theta})$$

Given the generic training example (x, y) and assuming it has been generated by a hypothesis $h_\theta(x)$ the likelihood function is (where φ has been replaced with our hypothesis):

$$P(y|\mathbf{x}) = \begin{cases} h_{\boldsymbol{\theta}}(\mathbf{x}) & \text{if } y = +1 \\ 1 - h_{\boldsymbol{\theta}}(\mathbf{x}) & \text{if } y = -1 \end{cases}$$

If we assume the hypothesis is the logistic function (so $l(\theta^T x)$) and by noticing that logistic function is symmetric, (i.e. l(-z) = 1-l(z)), the likelihood for a single example is:

$$P(y \mid \mathbf{x}) = \ell(y\boldsymbol{\theta}^T\mathbf{x})$$

So, knowing that the logistic function is symmetric we can rewrite the two cases above into a single case. Now, having access to a full set of m i.i.d. training examples D, then the overall likelihood function is computed as:

$$\prod_{i=1}^{m} P(y_i \mid \mathbf{x_i}) = \prod_{i=1}^{m} \ell(y_i\boldsymbol{\theta}^T\mathbf{x_i})$$

Let's try to see what happens to the likelihood function with respect to the sign of the actual observation y_i and the signal $\theta^T x_i$. If the label is **concordant** with the signal (either positively or negatively) then $l(y_i\theta^T x_i)$ approaches 1: This means that the **prediction agrees with the true label**. If the label is **discordant** with the signal then $l(y_i\theta^T x_i)$ approaches 1: This means that the **prediction disagrees with the true label**.

	$\theta^T x_i > 0$	$\theta^T x_i < 0$
$y_i > 0$	≈ 1	≈ 0
$y_i < 0$	≈ 0	≈ 1

So, we want to find the vector of parameters θ such that the likelihood function is maximum:

$$\text{argmax}_{\boldsymbol{\theta}} \left(\prod_{i=1}^{m} P(y_i \mid \mathbf{x_i}) \right) = \text{argmax}_{\boldsymbol{\theta}} \left(\prod_{i=1}^{m} \ell(y_i\boldsymbol{\theta}^T\mathbf{x_i}) \right)$$

Given a hypothesis h_θ and a training set D of m labeled samples we are interested in measuring the **"in-sample" (i.e. training) error**. Theoretically we have to find the maximum likelihood (so, as said before, we have to properly choose the parameters θ such that the likelihood function is maximum) but it might be better to transform this problem into something that is similar to an error measure like we did with linear regression:

$$E_{\text{in}}(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^{m} e(h_{\boldsymbol{\theta}}(\mathbf{x_i}), y_i)$$

where **e() measures how "far" the chosen hypothesis is from the true observed value**. Now, how can we "transform" MLE to the "in-sample" error above? Let's introduce the **negative log-likelihood**.

So we basically applied some **mathematical transformation to try to minimize the negative log likelihood** instead of maximizing the likelihood:

$$\text{argmax}_{\boldsymbol{\theta}} \left(\prod_{i=1}^{m} \ell(y_i\boldsymbol{\theta}^T\mathbf{x_i}) \right) \qquad \text{argmax}_{\boldsymbol{\theta}} \left(\frac{1}{m} \ln\left(\prod_{i=1}^{m} \ell(y_i\boldsymbol{\theta}^T\mathbf{x_i}) \right) \right)$$

The first formula on the left is the maximum likelihood estimation so it is the argmax of θ of the product of the logistic functions applied to the i-th signal multiplied by the i-th observations. We can apply the highlighted transformation and, logically speaking, nothing changes since the shape of the function is always the same (we simply squash the function but its behavior doesn't change).

$$\text{argmax}_{\boldsymbol{\theta}} \left(\frac{1}{m} \ln\left(\prod_{i=1}^{m} \ell(y_i\boldsymbol{\theta}^T\mathbf{x_i}) \right) \right) = \text{argmin}_{\boldsymbol{\theta}} \left(-\frac{1}{m} \ln\left(\prod_{i=1}^{m} \ell(y_i\boldsymbol{\theta}^T\mathbf{x_i}) \right) \right)$$

Now, maximizing the quantity on the left is exactly the same as minimizing the quantity on the right (we add a minus in front of the formula so that we have an opposite behavior). Now, let's decompose the formula above using the logarithms property. So basically, the logarithm of products is the sum of logarithms:

$$= \text{argmin}_{\boldsymbol{\theta}} \left(-\frac{1}{m} \ln\left(\ell(y_1\boldsymbol{\theta}^T\mathbf{x_1})\right) - \ldots - \frac{1}{m} \ln\left(\ell(y_m\boldsymbol{\theta}^T\mathbf{x_m})\right) \right)$$

as $k \ln(a \cdot b) = k(\ln(a) + \ln(b)) = k \ln(a) + k \ln(b)$.

$$= \text{argmin}_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} -\ln(\ell(y_i\boldsymbol{\theta}^T\mathbf{x_i})) \right) = \text{argmin}_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} \ln\left(\frac{1}{\ell(y_i\boldsymbol{\theta}^T\mathbf{x_i})} \right) \right)$$

as $-\ln(a) = \ln(\frac{1}{a})$.

Finally we rewrite the formula as a summation and use another property of the logarithm to remove the minus in front of the logarithm. Now, by noticing that logistic function can be rewritten as follows:

$$\ell(z) = \frac{e^z}{1+e^z} = \frac{1}{e^{-z}+1}$$

We can finally write the "in-sample" error to be minimized:

$$E_{\text{in}}(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^{m} \ln\left(e^{-y_i \boldsymbol{\theta}^T \mathbf{x_i}} + 1\right)$$

This is called **Cross-Entropy Error** and our goal is to minimize it. Now, 2 formulations of cross-entropy can be found depending on the labeling chosen for the (binary) response y:

$$\frac{1}{m}\sum_{i=1}^{m} \ln(e^{-y_i \boldsymbol{\theta}^T \mathbf{x}_i} + 1)$$

$y = \{-1, +1\}$

$$-\frac{1}{m}\sum_{i=1}^{m} y_i \ln(p) + (1-y_i)\ln(1-p) \qquad p = \frac{e^{\boldsymbol{\theta}^T \mathbf{x}}}{e^{\boldsymbol{\theta}^T \mathbf{x}}+1} = \frac{1}{1+e^{-\boldsymbol{\theta}^T \mathbf{x}}}$$

$y = \{0, 1\}$

Now, let's assume now that Y follows a Bernoulli distribution so we have a probability distribution p that tells us which is the probability of having 0 and which is the probability of getting 1. If we consider this assumption then we have a probability mass function of a Bernoulli-distributed random variable with known parameter p. This function tells us what is the probability of Y = 1 (Y = 0) given the parameter p. In the case of a Bernoulli variable this is quite easy in fact with probability p we get y = 1 and with probability q = 1 - p we get 0:

$$f_Y(y \mid p) = f_Y(Y = y \mid p) = \begin{cases} p & \text{if } y = 1 \\ q = 1 - p & \text{if } y = 0 \end{cases}$$

In other words, we plug in a value y for Y and f_Y tells us the **probability of observing that value given the parameter p**. The likelihood is the same but from the reverse perspective so we still use the probability mass function but this time we need to ask ourselves what is the likelihood of an observed Bernoulli distributed random variable is equal to y when the parameter p is unknown.

$$\mathcal{L}_Y(p \mid y) = f_Y(y \mid p) = f_Y(Y = y \mid p) = \begin{cases} p & \text{if } y = 1 \\ q = 1 - p & \text{if } y = -1 \end{cases}$$

So, the highlighted element is the likelihood of an observed Bernoulli-distributed random variable Y = y when the parameter p is unknown. We plug in a value p for the parameter of the distribution and f_Y tells us the likelihood of the observed Y = y. The likelihood function does not specify the probability that p is the truth, given the observed sample Y = y.

The likelihood function of m i.i.d. observations of Y is the following:

$$\mathcal{L}_Y(p \mid y_1, \ldots, y_m) = \prod_{i=1}^{m} p^{y_i}(1-p)^{1-y_i}$$

Here the unknown is the parameter p and we use the observations y_1, ..., y_m (we know yi and they can be either 0 or 1) to find p so as to maximize the likelihood:

$$p^* = \text{argmax}_p \left\{ \prod_{i=1}^{m} p^{y_i}(1-p)^{(1-y_i)} \right\}$$

Again, maximizing the log likelihood is equivalent to minimizing the **Negative Log-Likelihood**:

$$p^* = \text{argmin}_p \left\{ -\ln\left[\prod_{i=1}^{m} p^{y_i}(1-p)^{(1-y_i)} \right] \right\}$$

Again, let's apply some transformation like we did before and we get:

$$p^* = \text{argmin}_p \left\{ -\sum_{i=1}^{m} \ln\left[p^{y_i}(1-p)^{(1-y_i)} \right] \right\} = \text{argmin}_p \left\{ -\sum_{i=1}^{m} \ln(p^{y_i}) + \ln\left((1-p)^{(1-y_i)}\right) \right\}$$

$$= \text{argmin}_p \left\{ -\sum_{i=1}^{m} y_i \ln(p) + (1-y_i)\ln(1-p) \right\}$$

Notice that except for the 1/m factor this is **exactly the second formulation we gave for the cross-entropy error**. Now, let's substitute p that is our logistic function applied to the linear signal:

$$-\sum_{i=1}^{m} y_i \ln(p) + (1-y_i)\ln(1-p)$$

$$-\sum_{i=1}^{m} y_i \ln\left(\frac{e^{\boldsymbol{\theta}^T \mathbf{x}_i}}{e^{\boldsymbol{\theta}^T \mathbf{x}_i}+1}\right) + (1-y_i)\ln\left(1-\frac{e^{\boldsymbol{\theta}^T \mathbf{x}_i}}{e^{\boldsymbol{\theta}^T \mathbf{x}_i}+1}\right)$$

$$-\sum_{i=1}^{m} y_i[\ln(e^{\boldsymbol{\theta}^T \mathbf{x}_i}) - \ln(e^{\boldsymbol{\theta}^T \mathbf{x}_i}+1)] + (1-y_i)[\ln(1) - \ln(e^{\boldsymbol{\theta}^T \mathbf{x}_i}+1)]$$

$$-\sum_{i=1}^{m} y_i[\ln(e^{\boldsymbol{\theta}^T \mathbf{x}_i}) - \ln(e^{\boldsymbol{\theta}^T \mathbf{x}_i}+1)] + (1-y_i)[\ln(1) - \ln(e^{\boldsymbol{\theta}^T \mathbf{x}_i}+1)]$$

$$0$$

$$-\sum_{i=1}^{m} y_i \boldsymbol{\theta}^T \mathbf{x}_i - y_i \ln(e^{\boldsymbol{\theta}^T \mathbf{x}_i}+1) - \ln(e^{\boldsymbol{\theta}^T \mathbf{x}_i}+1) + y_i \ln(e^{\boldsymbol{\theta}^T \mathbf{x}_i}+1)$$

And we finally get:

$$-\sum_{i=1}^{m} y_i \boldsymbol{\theta}^T \mathbf{x}_i - \ln(e^{\boldsymbol{\theta}^T \mathbf{x}_i}+1)$$

Now, we want to show the 2 formulations on the right lead to the same function to be minimized. Basically if we plug in -1 in the first formula and 0 in the second (the same for +1 and 1) we get the same binary cross entropy error. In fact:

$$\sum_{i=1}^{m} \ln(e^{-y_i \boldsymbol{\theta}^T \mathbf{x}_i}+1) \qquad y = \{-1, +1\}$$

$$-\sum_{i=1}^{m} y_i \boldsymbol{\theta}^T \mathbf{x}_i - \ln(e^{\boldsymbol{\theta}^T \mathbf{x}_i}+1) \qquad y = \{0, 1\}$$

$$\sum_{i=1}^{m} \ln(e^{\boldsymbol{\theta}^T \mathbf{x}_i}+1) \qquad y = -1$$

$$=$$

$$\sum_{i=1}^{m} \ln(e^{\boldsymbol{\theta}^T \mathbf{x}_i}+1) \qquad y = 0$$

For the other case we get:

$$\sum_{i=1}^{m} \ln(e^{-\boldsymbol{\theta}^T \mathbf{x}_i}+1) \qquad y = 1$$

$$\stackrel{?}{=}$$

$$-\sum_{i=1}^{m} \boldsymbol{\theta}^T \mathbf{x}_i - \ln(e^{\boldsymbol{\theta}^T \mathbf{x}_i}+1) \qquad y = 1$$

They might look different but in reality it is possible to show that they are actually the same:

$$\sum_{i=1}^{m} \ln(e^{-\boldsymbol{\theta}^T \mathbf{x}_i}+1) = \sum_{i=1}^{m} \ln\left(\frac{1}{e^{\boldsymbol{\theta}^T \mathbf{x}_i}}+1\right) = \sum_{i=1}^{m} \ln\left(\frac{1+e^{\boldsymbol{\theta}^T \mathbf{x}_i}}{e^{\boldsymbol{\theta}^T \mathbf{x}_i}}\right)$$

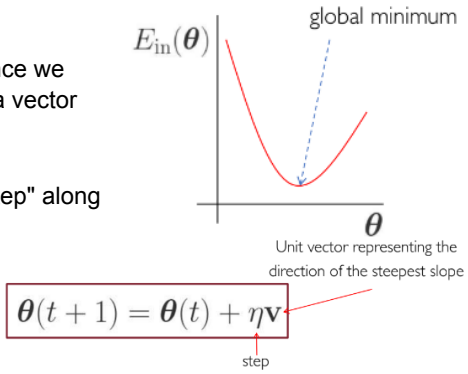
$$= \sum_{i=1}^{m} \ln(1+e^{\boldsymbol{\theta}^T \mathbf{x}_i}) - \ln(e^{\boldsymbol{\theta}^T \mathbf{x}_i}) = -\sum_{i=1}^{m} \boldsymbol{\theta}^T \mathbf{x}_i - \ln(1+e^{\boldsymbol{\theta}^T \mathbf{x}_i})$$

The only thing remaining to see is the actual **learning algorithm** that allows us to find the MLE. To actually select the best hypothesis, we have to pick the vector of parameters θ* so that the error measure is minimized:

$$E_{\text{in}}(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^{m} \left(\boldsymbol{\theta}^T \mathbf{x_i} - y_i\right)^2$$

Note: This is really similar to what we saw for linear regression with the only difference that here we cannot find a closed-form solution to the minimization problem. Yet, **Cross-Entropy is convex w.r.t. the parameters θ**. This means that we can apply an **iterative approach to find the solution**.
The general iterative method for any nonlinear optimization is the **(Batch) Gradient Descent** that is a method that guarantees the convergence to a local minimum (under specific assumptions on the objective function and learning rate). So once we reach the minimum we have no guarantee that it is also the global one, but **if we know that the objective function is convex** (like cross-entropy) **then the local minimum is also the global minimum**.
The main idea is the following:

1. At t = 0 initialize the (guessed) vector of parameters θ to θ(0) (so since we don't know anything, we can initialize θ as we want so randomly or a vector full of zeros for example);
2. Repeat until convergence:
    a. Update the current vector of parameters θ(t) by taking a "step" along the "steepest" slope: θ(t+1) = θ(t) + ηv (so to compute θ at time t+1, we take what we have in time θ and sum it to ηv where v is the direction and η is the step, also called learning rate);
3. Return to 2.



So, v is the **direction of the "steepest" slope** that means moving along the direction which mostly reduces the in-sample error function. Basically we should compute the difference of the errors at time t and t-1:

$$\Delta E_{\text{in}}(\boldsymbol{\theta}, t) = E_{\text{in}}(\boldsymbol{\theta}(t)) - E_{\text{in}}(\boldsymbol{\theta}(t-1))$$

We want ΔE_{in} to be as **negative as possible**, which means that we are actually reducing the error w.r.t. the previous iteration t-1 (so the more negative, the more we reduce the error at each step, the better we do). If we replace the first bit with the what was computed at the previous step, we get:

$$\Delta E_{\text{in}}(\boldsymbol{\theta}, t) = E_{\text{in}}(\boldsymbol{\theta}(t-1) + \eta\mathbf{v}) - E_{\text{in}}(\boldsymbol{\theta}(t-1))$$

Now, we basically want to find out what v is so let's first assume we are in the univariate case where θ = ϑ in R (so θ is just a scalar). We define f = E_{in}, x_0 = θ(t-1) and x = θ(t). Using this notation we have:

$$\delta f = \Delta E_{\text{in}} = f(x) - f(x_0) \qquad \delta x = x - x_0 = \boldsymbol{\theta}(t) - \boldsymbol{\theta}(t-1) = \eta\mathbf{v}$$

We can notice that if we take the first derivative of f in x_0, it can be approximated by the ration between $\boldsymbol{\delta}$f and $\boldsymbol{\delta}$x:

$$f'(x_0) = \lim_{\delta x \to 0} \frac{f(x_0 + \delta x) - f(x_0)}{\delta x} \qquad f'(x_0) = \lim_{x \to x_0} \frac{f(x) - f(x_0)}{x - x_0} \approx \frac{\delta f}{\delta x}$$

Where $\boldsymbol{\delta}$f is equal to (just apply some mathematics):

$$\delta f = f(x) - f(x_0) \approx f'(x_0)\delta x = f'(x_0)(x - x_0)$$

Now, we can calculate f(x) using the first-order Taylor approximation and the second-order error term:

$$f(x) = \underbrace{f(x_0) + f'(x_0)(x - x_0)}_{\text{First-order Taylor approximation}} + \underbrace{O((x - x_0)^2)}_{\text{Second-order error term}}$$

To summarize and generalize to the multivariate case of θ:

$$\delta f = f(x) - f(x_0) = \Delta E_{\text{in}} = \eta \nabla E_{\text{in}}(\boldsymbol{\theta}(t-1))^T \mathbf{v} + O(\eta^2)$$

The greek letter *nabla* indicates the gradient

So, let's start again from here:

$$\Delta E_{\text{in}} = \eta \nabla E_{\text{in}}(\boldsymbol{\theta}(t-1))^T \mathbf{v} + O(\eta^2)$$

The **unit vector v only contributes to the direction and not to the magnitude of the iterative step** (η is responsible for that). Also, the second-order approximation term ($O(\eta^2)$) is negligible (when the step size is small)

so we can remove the second term in the summation. Now, if we call the gradient a vector u, we can rewrite everything like:

$$\nabla E_{\text{in}}(\boldsymbol{\theta}(t-1))^T = \mathbf{u} \quad \Delta E_{\text{in}} = \eta \mathbf{u} \cdot \mathbf{v}$$

How do we calculate the dot product between u and v?

$$\mathbf{u} \cdot \mathbf{v} = ||\mathbf{u}|| \underbrace{||\mathbf{v}||}_{=1} cos(\alpha) = ||\mathbf{u}|| cos(\alpha)$$

Note: v is a unit length vector so its norm is equal to 1 and the -1 ≤ cos(α) ≤ 1.
This means that the multiplication is a value that is larger than minus the norm of u and smaller than the norm of u and if we also plug in the step (learning rate η) we get:

$$-||\mathbf{u}|| \leq \mathbf{u} \cdot \mathbf{v} \leq ||\mathbf{u}|| \qquad -\eta||\mathbf{u}|| \leq \underbrace{\eta \mathbf{u} \cdot \mathbf{v}}_{\Delta E_{\text{in}}} \leq \eta||\mathbf{u}||$$

Now, the most positive ΔE_{in} when cos(α) = 1 (i.e. α = 0°) (so the case on the right) is the case in which both error and step vectors have the same direction. Of course we are not interested in this case but in the left one where we have the most negative ΔE_{in} when cos(α) = -1 (i.e., α = 180°) and the **error and step vectors have opposite directions**.
At each iteration t, **we want the unit vector v which makes exactly the most negative ΔE_{in}**:

$$\eta \mathbf{u} \cdot \mathbf{v} = -\eta||\mathbf{u}||$$

With some algebraic steps we get:

$$\mathbf{u} \cdot \mathbf{v} = -||\mathbf{u}|| \quad \mathbf{u}^T \cdot \mathbf{u} \cdot \mathbf{v} = -||\mathbf{u}||\mathbf{u}^T$$

$$\mathbf{v} = -\frac{||\mathbf{u}||\mathbf{u}^T}{||\mathbf{u}||^2} = -\frac{\mathbf{u}^T}{||\mathbf{u}||} = -\frac{\nabla E_{\text{in}}(\boldsymbol{\theta}(t-1))}{||\nabla E_{\text{in}}(\boldsymbol{\theta}(t-1))||}$$

The denominator enables the normalization of the vector at the nominator to unit length.

Now, how does the step magnitude η affect the convergence?
Depending on η is too small we could reach convergence too slowly, if η is too large we may zig zagging around the surface (this is not ideal) or even diverge so the solution might be to use a **η that is variable**. So, the idea is to dynamically change η proportionally to the gradient (so we start with a large η and we minimize it depending on the value of the gradient: the smaller the gradient, the smaller the steps we take).
Remember that at each iteration the update strategy is:

$$\boldsymbol{\theta}(t+1) = \boldsymbol{\theta}(t) + \eta \mathbf{v} \qquad \mathbf{v} = -\frac{\nabla E_{\text{in}}(\boldsymbol{\theta}(t))}{\|\nabla E_{\text{in}}(\boldsymbol{\theta}(t))\|}$$

At each iteration t, the step η is fixed. Instead of having a fixed η at each iteration, use a variable η_t as function of η. So, let's take:

$$\boldsymbol{\theta}(t+1) = \boldsymbol{\theta}(t) + \eta_t \mathbf{v} \qquad \eta_t = \eta k \qquad \boldsymbol{\theta}(t+1) = \boldsymbol{\theta}(t) - \eta k \frac{\nabla E_{\text{in}}(\boldsymbol{\theta}(t))}{\|\nabla E_{\text{in}}(\boldsymbol{\theta}(t))\|}$$

To make k depending on t, is just make it equal to the actual norm of the gradient. This allows us to simply the equation getting:

$$\boldsymbol{\theta}(t+1) = \boldsymbol{\theta}(t) - \eta \cancel{\|\nabla E_{\text{in}}(\boldsymbol{\theta}(t))\|} \cancel{\frac{\nabla E_{\text{in}}(\boldsymbol{\theta}(t))}{\|\nabla E_{\text{in}}(\boldsymbol{\theta}(t))\|}} \qquad \boxed{\boldsymbol{\theta}(t+1) = \boldsymbol{\theta}(t) - \eta \nabla E_{\text{in}}(\boldsymbol{\theta}(t))}$$

This allows us to include the optimization we mentioned before inside of the updating rule.

What happens in the case of Cross-Entropy?
Let's plug in the binary cross entropy as the error function that we want to minimize:

$$\nabla E_{\text{in}}(\boldsymbol{\theta}) = \nabla \left[\frac{1}{m} \sum_{i=1}^{m} \ln(e^{-y_i \boldsymbol{\theta}^T \mathbf{x}_i} + 1) \right]$$

$$= \left[\frac{1}{m} \sum_{i=1}^{m} \nabla \ln(e^{-y_i \boldsymbol{\theta}^T \mathbf{x}_i} + 1) \right] = \left[\frac{1}{m} \sum_{i=1}^{m} \frac{1}{e^{-y_i \boldsymbol{\theta}^T \mathbf{x}_i} + 1} \nabla (e^{-y_i \boldsymbol{\theta}^T \mathbf{x}_i} + 1) \right]$$

chain rule of derivative

$$= \frac{1}{m} \sum_{i=1}^{m} \frac{-y_i \mathbf{x}_i e^{-y_i \boldsymbol{\theta}^T \mathbf{x}_i}}{e^{-y_i \boldsymbol{\theta}^T \mathbf{x}_i} + 1} = \boxed{-\frac{1}{m} \sum_{i=1}^{m} \frac{y_i \mathbf{x}_i}{1 + e^{y_i \boldsymbol{\theta}^T \mathbf{x}_i}}}$$

So, finally the algorithm to apply for gradient descent is the following:

1. At t = 0 initialize the (guessed) vector of parameters θ to θ(0);
2. For t = 0, 1, 2, ... until stop:
    a. Compute the gradient of the cross-entropy error:

$$E_{\text{in}}(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^{m} \ln(e^{-y_i \boldsymbol{\theta}^T \mathbf{x}_i} + 1)$$

$$\nabla E_{\text{in}}(\boldsymbol{\theta}(t)) = -\frac{1}{m} \sum_{i=1}^{m} \frac{y_i \mathbf{x}_i}{1 + e^{y_i \boldsymbol{\theta}(t)^T \mathbf{x}_i}}$$

    b. Update the vector of parameters: $\theta(t+1) = \theta(t) - \eta \nabla E_{in}(\theta(t))$;
    c. Return to 2;
3. Return the final vector of parameters θ(∞).

Now, we said before that we have to initialize the initial θ in some ways. Typically what we do is choose a random initialization and if the function is convex we are guaranteed to reach the global minimum no matter what is the initial value of θ(0). In general, we may get to the local minimum nearest to θ(0) even if the function is non-convex. So, **Gradient Descent can still be used to try to optimize non-convex objectives**. The problem is that **non-convex functions may have several local minima**. A bad initialization might cause GD to end up into a "bad" local minimum and miss "better" ones (or even the global minimum if it exists). The heuristic solution is repeating Gradient Descent 100 ÷ 1,000 times each time with a different θ(0) and this may reduce the chance the above issue occurs.
About the **stopping criterion**: If the function is convex GD reaches the global minimum when $\nabla E_{in}(\theta(t)) = 0$. • In general, we don't know if eventually the gradient gets to 0 therefore we can use several criteria of termination:

- Stop whenever the difference between two iterations is "small enough" → may converge "prematurely";
- Stop when the error equals to ε → may not converge if the target error is not achievable;
- Stop after T iterations;
- Combinations of the above in practice works.

Some advanced topics:

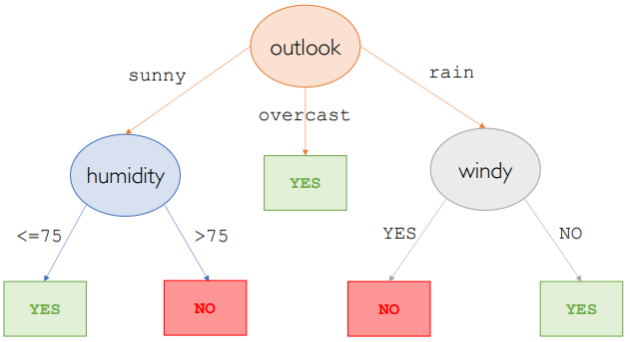
- Gradient Descent using **second-order approximation**: Better local approximation than first-order but each step requires computing the second derivative (Hessian matrix);
- **Stochastic** vs. **Mini-Batch Gradient Descent** (SGD vs. MBGD): At each iteration, compute the gradient only from one instance (SGD) or a sample of k instances (MBGD) rather than the full dataset;
- **Regularization**: Include the L1- or L2-norm of the vector of parameters θ in the cross-entropy error to avoid overfitting.

8. Decision Trees and Ensembles

We presented 2 linear models: linear regression and logistic regression and those hypotheses work well whenever there exists a linear relationship between the features (input) and the response (output).
Other methods that are suitable for both regression and classification tasks are the **tree-based methods**.

To introduce the topic we can say that these methods work by repeatedly splitting the input feature space into a number of regions. A prediction is computed by taking the **mean (regression)** or the **mode (classification)** of the points in the region in which the observation belongs. The set of splitting rules can be represented as a tree (the so-called **decision tree**). Notice that since we are dealing with trees, the models are also highly human-interpretable (we can understand better how they work with respect to the other models we saw).

EX. Let's start with a simple example. Imagine we have a bunch of data points and we want to use the values of the set of features we have to split the data so that each splitting represents a group of data that is similar. So, we can start with a **root** (in our case "outlook") and we **split** the data depending on the value of that feature (e.g. sunny, overcast, windy). If overcast, the prediction is "yes". If sunny we split the data again considering a **decision node** (in our case humidity). Now, originally humidity is a real value but we can define a splitting point so that if the value of the value is smaller or equal than 75 then the prediction is "yes", otherwise "no". In any case we reach a **leaf**.



Now, we have understood that we can build a decision tree from the data we have but, how do we do it?
We have to split the input feature space (i.e. the set of possible values observed for each feature x_i) into a set of non-overlapping regions R_1, R_2, …, R_J and for every observation that falls into the region R_j, we make the same prediction (i.e. the mean of the response values in R_j in case of a regression task).
EX. Let's suppose we split the input feature space in 2 regions (so the feature is binary): R_1 and R_2 and the response mean as computed from R_1 = 10 and R_2 = 20. For any x belonging to R1 (R2) will be predicted 10 (20). So, for every observation we see what the region is, and the final prediction for that specific x will be the same depending on the region: if x ends up in R_1 then we predict 10, otherwise 20.

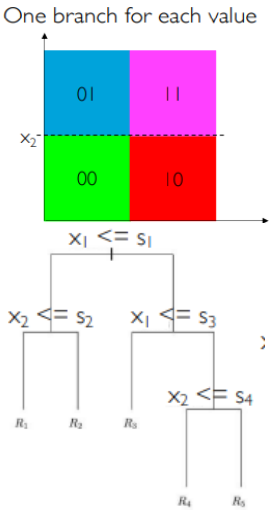
Note: If the data is linear then the model would take a lot of time to train (it would need a lot of partitioning and would not be so accurate) so it is better to use decision trees only when the data is not linear also because they enable to represent more complex situations.

In theory, partitions could have any shape but we choose to divide the input feature space into **hyper-rectangles** (i.e. high-dimensional rectangles) so that this way, the obtained model is easier to interpret. Now, in order to find the best partition what we could do is **minimizing the Residual Sum of Squares (RSS)**. Basically what we do is take all the points that end up in a certain region R_j and we calculate the difference between the actual observation and the mean of that region. Finally we take all the summation and sum them all (we want to minimize this quantity).

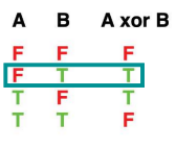
$$\text{RSS} = \sum_{j=1}^{J} \sum_{i \in R_j} (y_i - \hat{y}_{R_j})^2$$

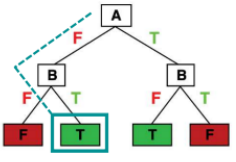
Note: Notice that if we take one region for data point and in this case the RSS would be equal to 0 but in this case we are not training a model but we simply memorize the training set. Since we have no generalization at all, this means we are terribly overfitting.

Now, let's assume to have a bunch of features that can be both discrete or continuous. In the case of **discrete values**, we have one branch for each value the feature can take (e.g. in the example above, "windy" splits in two branches: "yes" and "no"). If we assume to have another binary feature, we have another possible binary splitting. If we combine them all then we have to consider all possible splitting combinations (e.g. 00 is the case in which both x_1 and x_2 are equal to 0).
In the case of **continuous inputs**, for each attribute, we have to find a splitting point s and create 2 branches (e.g. in the example above, "humidity" splits in two branches: "<=75" and ">75"). The same attribute may be further split in each subtree.



In case of **discrete inputs/outputs**, DecisionTrees can **express any function of the inputs**. Basically, it is like we create possible combinations and each of them maps to a **root-to-leaf path** on the tree (e.g. think of the boolean function).





In case of **continuous inputs/outputs**, DecisionTrees can **approximate any function arbitrarily closely**. Trivially, there exists a consistent decision tree for any training set and such a tree will have one dedicated root-to-leaf path for each training instance. Of course, this tree clearly overfits the training data and it will not generalize to unseen examples (it would need regularization).

x_1	x_1	...	x_n	y
0	0	...	0	
1	0	...	0	
1	1	...	0	
...	...	...	...	
...	...	...	...	
...	...	...	...	
1	1	1	1	

Now, how many distinct decision trees can be built out of n boolean attributes?
We saw we have to define many boolean functions to express the decision tree and it might be interesting to understand how many. Each boolean function of n boolean inputs is represented by a truth table with 2^n rows. For each input (each row), it can take values y=0 or y=1. A possible boolean function is the one which will output all 0s. Another one is the one which will output all 1s. Then we have to consider all the other cases. This means that overall, there are $2^{(2^n)}$ possible boolean functions, therefore decision trees and this is a **huge number** (e.g. with just 6 boolean attributes, there are 18,446,744,073,709,551,616 trees!). The hypothesis space defined by the decision tree is **very expressive** because there are a lot of different functions it can represent but this means that a larger hypothesis makes finding the best hypothesis harder (because of the larger space to explore).

Unfortunately, minimizing RSS would require exploring all the possible partitions of the feature space into J boxes and the problem is known to be **NP-Complete**, therefore computationally infeasible. The solution is to use a **Top-Down greedy heuristic Recursive Binary Splitting**.
The **Recursive Binary Splitting (RBS)** works this way:

- Initially, all the observations belong to the **root** of the tree;
- Split the input feature space into 2 subtrees (i.e. left and right);
- Recursively repeat the step above on both subtrees.

So it is a **top-down approach** because we start from a root, split the tree and keep on doing it until we reach a leaf. It is a **greedy strategy** because at each step, the best "local" split is made. Looking ahead might result in a different split, which leads to a better tree but since we use only the local information, we can't know it in advance. Intuitively, we **need to select the feature f and a splitting point s** (of f) and such a splitting will lead to the partitioning of the feature space into the following regions:

$$R_{\text{left}}(f,s) = \{\mathbf{x}_i \mid x_{i,f} \leq s\} \quad R_{\text{right}}(f,s) = \{\mathbf{x}_i \mid x_{i,f} > s\}$$

So, in the region R_{left} we put all values in which the f-th features are less or equal to s, while in R_{right} we put the ones greater than s. Among all the possible splittings, we **select the one which reduces RSS the most**.
For each feature f = 1, .., d we look at every possible splitting value s (i.e. all the values we observed in the training set for the feature f) and for each pair of feature and splitting point (f, s) we obtain the 2 regions described above. The goal is to find the pair (f, s) which **minimizing the sum of the RSS of the left region and the right region**:

$$\sum_{i:\ \mathbf{x}_i \in R_{\text{left}}(f,s)} (y_i - \hat{y}_{R_{\text{left}}})^2 + \sum_{i:\ \mathbf{x}_i \in R_{\text{right}}(f,s)} (y_i - \hat{y}_{R_{\text{right}}})^2$$

Finding the pair (f, s) which minimizes the quantity below can be done "easily", especially when the number of features d is not too large. At the next step, the same splitting strategy applies yet considering only the data falling into the previously identified regions (so, again, it is a greedy approach and we only take into account the local information). Each time we do this operation, we **reduce the RSS**.
We need a stopping criterion otherwise the tree will grow until each training instance falls into a leaf node and this is the case in which the RSS is **minimum** (RSS = 0) but this is the case in which **the tree is overfitted** and we don't want that. Possible **stopping criteria** are:

- No region contains more than N observations;
- Max depth of the tree is D;
- RSS is reduced by at least a threshold value t.

Once the tree is built using training instances, we will have **J regions**. In each of them, we have a fraction of the initial training examples. We can compute the mean of each region as the mean of the responses of the instances falling in that region. At test time, an unseen instance follows a root-to-leaf path on the tree and ends up into a region R_j and the **prediction for that test instance will be the mean of the region R_j**.

What we described so far is valid for regression trees (useful when we want to predict a real value).
Classification Trees are very similar to a regression tree and are used to predict a categorical response rather

than a numerical one. Tree building is still based on Recursive Binary Splitting algorithm but **RSS cannot be used as a criterion for splitting nodes**. A natural alternative to RSS minimization is to minimize the "**impurity**". The predicted label of a test instance is the **most frequent label (mode)** of the instances belonging to the region where it falls. What is impurity?
A node containing instances all with the same label is perfectly pure. We would like to grow a tree whose nodes are as purest as possible. There are some ways to measure the notion of node "impurity":

- Classification Error Rate (not the best method, we'll see why);
- Gini Index;
- Entropy.

It is often convenient to refer to the **information gain of a split** that tells us if it's convenient or not to do a splitting. If positive it is useful to split a node in left and right node, otherwise it is not convenient to do the splitting. Basically, we have a gain when we reduce the impurity.

Let's define some notation. Let's define D_R as the intersection between the initial dataset D and some region R such that it contains all the data points from D that ends up in R:

$$\mathcal{D}_R = \mathcal{D} \cap R = \{(\mathbf{x}_i, y_i) \in \mathcal{D} \mid \mathbf{x}_i \in R\}$$

Again, we can define the **(f, s)-split** to decide what data points go to the left or to the right:

$$R_{\text{left}}(f, s) = \{\mathbf{x}_i \mid x_{i,f} \leq s\} \quad R_{\text{right}}(f, s) = \{\mathbf{x}_i \mid x_{i,f} > s\}$$

Given D_R, we can compute the **impurity of the region I(D_R)** where the function I is computable by using the methods above (yet to be described). The **information gain obtained with an (f, s)-split IG(D_R, f, s)** tells us what is the gain we get by splitting the data that we have in D_R according to the splitting criterion (f, s) (so it tells us what the benefit of applying that splitting criterion). Of course impurity and information gain are correlated. In fact we start off with a certain impurity, we apply the splitting criterion, hopefully we get a gain and we measure what the impurity is afterward (we want to find the splitting criterion that reduces the impurity the most).
So, at the same time we want to **maximize the information gain**. At each step, the best (f, s)-split is found so as to **minimize the children nodes' impurity** and this corresponds to selecting the (f, s)-split that **maximizes the information gain**. How do we compute the information gain?

$$IG(\mathcal{D}_R, f, s) = \underbrace{I(\mathcal{D}_R)}_{\text{parent node's impurity}} - \underbrace{\left[\frac{|\mathcal{D}_{R_{\text{left}}(f,s)}|}{|\mathcal{D}_R|} I(\mathcal{D}_{R_{\text{left}}(f,s)}) + \frac{|\mathcal{D}_{R_{\text{right}}(f,s)}|}{|\mathcal{D}_R|} I(\mathcal{D}_{R_{\text{right}}(f,s)})\right]}_{\text{children nodes' impurity}}$$

We take the parent node's impurity (the one of the local root) and we subtract it to the sum of the impurity of the node and the right node. Notice that we ratios are just weighting factors so we weight the impurity of the resulting nodes by how many data points end up in that region. We want to reduce the impurity of the parent node so **we want the result to be positive** (if negative we increase the impurity so the given splitting criterion is not good).

Now, before we said that there are some ways to compute the node impurity. Let's see them more in details:

- **Classification error rate**: It is the fraction of the training observations in that region that are not labeled with the most common class.

  $$I_E(\mathcal{D}_{Rj}) = 1 - \max_k\{\hat{p}_{jk}\}$$

  The element inside brackets represents the proportion of training observations in the j-th region that are from the k-th class. Although this is the most intuitive notion of impurity we will see how this may not be a great choice. So again, we are in a certain region, we count the data points, we take the most common class among them once we have this information to compute the impurity. We want to reduce the impurity the most that is when most of the elements are of the same class.
- **Gini Index**: It measures the variance across the K classes.

  $$I_G(\mathcal{D}_{Rj}) = \sum_{k=1}^{K} \hat{p}_{jk}(1 - \hat{p}_{jk})$$

  A small value of Gini is obtained whenever all the proportions are either close to 1 or to 0. A small value indicates that a node contains predominantly observations from a single class (so it is better). The worst

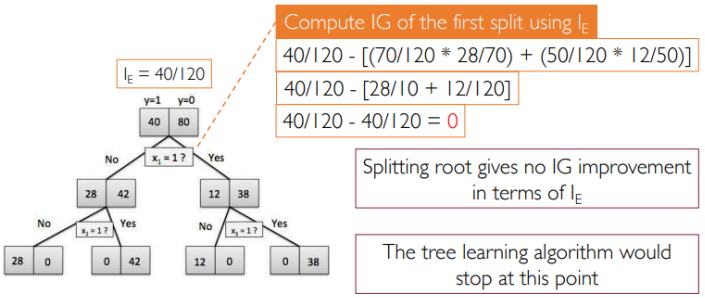
case (the most impure) is when we have half data points from a class and half data points from the other class (assuming to have a binary distribution).

- **Entropy**: An alternative that is similar to the Gini Index.

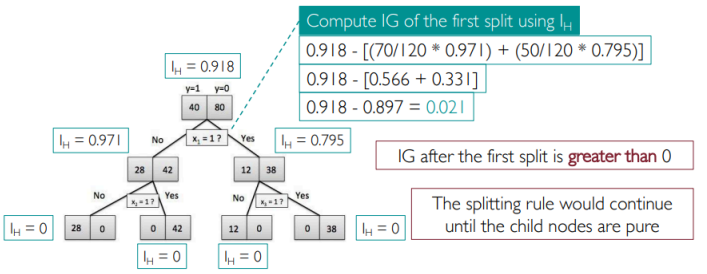
$$I_H(\mathcal{D}_{Rj}) = -\sum_{k=1}^{K} \hat{p}_{jk} \log_2(\hat{p}_{jk})$$

It ranges between [0, +infinity]. Like Gini, a small value of entropy is obtained whenever all the proportions are either close to 1 or to 0 (so also here the smaller, the better).

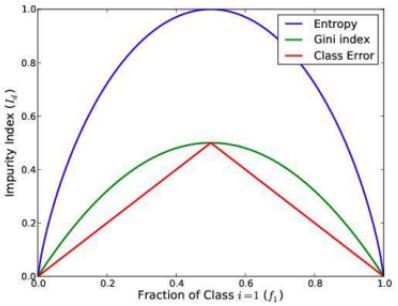
In practice, both entropy and Gini can be used to grow a tree. We can plug in one of these methods into the formula we described before.

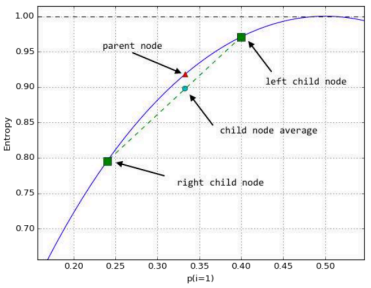


Note: Classification Error rate is not the best approach to compute the impurity. Let's assume there are 3 features to do the splitting. If we use this method what happens is that the information gain after the first split is exactly 0 so we would stop then (we can't continue) but this means that we can't reach the perfect tree we see in the example on the right in which each leaf has either class 1 or class 0. If otherwise we use the Entropy (or Gini) method this is possible because the information gain after the first split would possibly be greater than 0 (not 0!) and this would allow us to continue the splitting operation (we don't stop immediately at the first step) and this means that we can reach the perfect tree.



Gini and Entropy are "smoother" than classification error but this means that when computing the entropy, the result (assuming that it is the average between the left child node and the right child node) will always be lower than the parent node so we always have a gain. This happens thanks to the bell shaped function of the Entropy and the Gini index that in the Classification error we can't have because the risk is that everything is flat.





Suppose we have x_1, …, x_N N items where each of them is labeled either as positive (y=1) or negative (y=0). Let N+ (N-) be the number of positive (negative) elements. We define the ratio p = N+/N and q = N- /N. The entropy is defined as:

$$H = -\left[p \log_2(p) + q \log_2(q)\right] = -\frac{N^+}{N} \log_2\left(\frac{N^+}{N}\right) - \frac{N^-}{N} \log_2\left(\frac{N^-}{N}\right)$$

H is minimum (i.e. H=0) when either N+ or N- is equal to N and this is the best case of a pure set (all the elements have the same label). H is maximum (i.e. H=1) when N+ = N- = N/2 and this is the worst case (half of the elements labeled positive, half negative) because we have the highest level of confusion and we can't make the prediction. In the case x_i takes on a binary value then H ranges in [0, 1] independently of the size of the set. In the general case H ranges in [0, +∞].

Suppose now that we start with a node whose entropy is 1 (half +/half -). The worst split will occur if the two resulting children nodes will both contain half +/half - examples. In such a case, each child node will still have entropy = 1. To account for the number of elements, the **weighted average** of children entropies are computed:

$$H_{\text{split}} = \frac{N_{\text{left}}}{N} H_{\text{left}} + \frac{N_{\text{right}}}{N} H_{\text{right}}$$

In this case, **splitting can't do worse**.

Note: As already said before, if there is a strong linear relationship between input and output then linear models are better while if there is a highly non-linear relationship between input and output then decision trees are preferred. Notice that in the decision trees formula we have an indicator function: basically what happens is that we only consider the value of a certain region where the x belongs to. So the indication function returns 1 only if the x belongs to the region R_j, in all the other cases return 0. This means that we only consider c_j where the indicator function returns 1.

Linear Models

$$h_{\boldsymbol{\theta}}(\mathbf{x}) = \theta_0 + \sum_{i=1}^{n} \theta_i x_i$$

Decision Trees

$$h(\mathbf{x}) = \sum_{j=1}^{J} c_j \cdot \mathbf{1}_{R_j}(\mathbf{x})$$

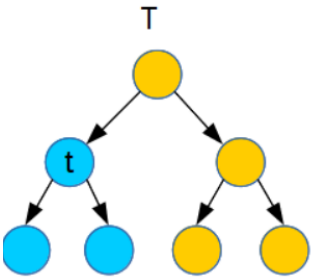
Now, using the greedy tree growing strategy is really easy to experience:

- Overfitting: If we keep splitting as long as there is an information gain;
- Underfitting: If we early-stop the splitting process;

A better strategy is to grow a very large tree T_0, and then prune it back in order to obtain a subtree: this technique is called **pruning**. In order to determine the **subtree**, we can take the one **with the smallest test error**. Given a subtree, we can estimate its test error using cross-validation or the validation set approach but the problem is that we have too many subtrees to consider (unfeasible!). We need a way to select a small set of subtrees for consideration with the **Cost Complexity Pruning** (a.k.a. Weakest Link Pruning).
The overall goal is to minimize the cost-complexity function:

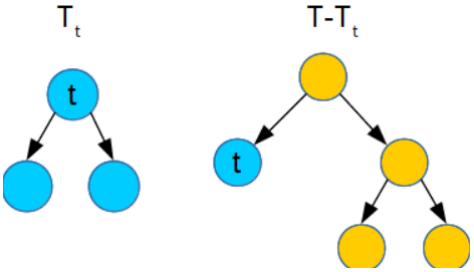
$$C_\alpha(T) = R(T) + \alpha|T|$$

where |T| is the number of leaves in tree T and R(T) a loss function calculated across these leaves. First step is to calculate a sequence of subtrees $T_0 \supseteq T_1 \supseteq .. \supseteq T_{n-1} \supseteq T_n$ where T_n is the tree consisting only of the root node and T_0 the whole tree. This is done by successively replacing a subtree T_t with root node t with a leaf (i.e. collapsing this subtree). In each step the subtree T_t is selected, which minimizes the decrease in the cost-complexity function and hence is the weakest link of the tree. So, we have to find the best subtree that minimizes the error, so we basically have to minimize the difference between $C_\alpha(T-T_t)-C\alpha(T)$. By applying some math we have that this is equivalent to minimizing:



$$\alpha = \frac{R(t) - R(T_t)}{|T_t| - 1}$$

So starting with the whole tree T_0 (and $\alpha_0 = 0$) in each step s the algorithm:



- Selects the node t which minimizes: $\frac{R(t)-R(T_t^{s-1})}{|T_t^{s-1}|-1}$
- Set: $T^s = T^{s-1} - T_t,\ \alpha^s = \frac{R(t)-R(T_t^{s-1})}{|T_t^{s-1}|-1}$

until the tree consists only of the root node. Hence as output we get a sequence of subtrees $T_0 \supseteq T_1 \supseteq .. \supseteq T_{n-1} \supseteq T_n$ alongside with the corresponding α-values $0 = \alpha_0 \leq \alpha_1 \leq .. \leq \alpha_{n-1} \leq \alpha_n$. Using these values one can define a mapping from α to a list of subtrees. Now, the cost-complexity function and so the loss / error function have been calculated on the training data, hence the danger of self-validation and overfitting is present. Because of this the final α is determined by cross validation. Calculating the sequence of subtrees of the tree trained on all the training-data (before optimization via inner cross validation) at least gives us an interval of possible α-values to select from.

Some **PROs of decision trees** are:

- Trees are very easy to explain (even easier than linear regression!);
- Some people think decision trees mimic human decision-making;
- Trees can be displayed graphically, and are easily interpreted even by a nonexpert (especially if they are small);
- Trees can easily handle categorical features without the need to create dummy variables (i.e., one-hot encoding).

Some **CONs of decision trees** are:

- Trees generally have **lower predictive accuracy** than regression and classification approaches;
- Trees may be **not very robust**: a small change in the data can cause a large change in the final estimated tree.

Some improvement can be done using some techniques such as **Bagging**, **Random Forest** and **Boosting**. These are **Ensemble methods**.

Let's see how **Bagging** works. Standard decision trees suffer from **high variance (overfitting)**. If we randomly split a training set in two halves and fit a decision tree on each, chances are we end up with 2 very different trees. Low-variance approaches, instead, are less sensitive to different training sets. **Bootstrap aggregation (Bagging)** is a general-purpose method to **lower the variance** of a statistical learning method. Given a set of n i.i.d. observations $Z_1, \dots, Z_n$, each with variance σ^2, the variance of the empirical mean is σ^2/n. The proof is the following:

$$\bar{Z} = \frac{1}{n}\left(Z_1 + \dots + Z_n\right)$$

$$\mathrm{Var}(\bar{Z}) = \mathrm{Var}\left[\frac{1}{n}\left(Z_1 + \dots + Z_n\right)\right] = \frac{1}{n^2}\left[\mathrm{Var}(Z_1) + \dots + \mathrm{Var}(Z_n)\right] = \frac{1}{n^2}\left(\underbrace{\sigma^2 + \dots + \sigma^2}_{n}\right) = \frac{n\sigma^2}{n^2} = \frac{\sigma^2}{n}$$

$$\mathrm{Var}(\alpha X) = \alpha^2 \mathrm{Var}(X)$$

$$\mathrm{Var}\left(\sum_{i=1}^{n} X_i\right) = \sum_{i=1}^{n} \mathrm{Var}(X_i) \quad \text{if } X_i \text{ are independent}$$

This means that if we **average a set of observations**, this **reduces the variance**.
What we can do is taking many training sets, build a separate prediction model on each, and average the resulting predictions:

$h^1(\mathbf{x}), \dots, h^B(\mathbf{x})$ — B predictive models learned from B training sets

$$h_{\text{avg}}(\mathbf{x}) = \frac{1}{B}\sum_{b=1}^{B} h^b(\mathbf{x})$$ — Final model obtained averaging the B predictions

Unfortunately, we generally do not have access to multiple training sets so what we can do is apply **Bootstrap**, which is about taking repeated samples from the same training set.
So, again, we generate B different bootstrapped samples from the original training set, we train a model on each bootstrapped sample and finally average the B predictions:

$$h_{\text{bagging}}(\mathbf{x}) = \frac{1}{B}\sum_{b=1}^{B} h^{b*}(\mathbf{x})$$

The improved prediction accuracy of bagging trees comes at the expense of the interpretability of a single tree. Still, one can obtain an overall summary of the importance of each feature using RSS (regression) or Gini index/Entropy (classification). Add up the total RSS/Gini index reduction obtained splitting on a certain feature and take the average over all the B trees.
Note: Note that bagging is a general-purpose framework and it can be used in combination with any model. When used with classification trees the final prediction is typically obtained via majority voting (instead of considering the average). So, the overall prediction is just the most common across the B models.

Let's see **random forests**. Random forests improve bagging trees through **decorrelating individual trees**. As in bagging, there will be B decision trees learned on bootstrapped samples of the original training set but for each individual tree, **every time it comes to splitting a node only a random sample of k < n features is considered**. Each split is allowed to use only one of those k features.
Note: More on random forest and boosting → slide.