Algorithms for Massive Datasets - Notes from prof. Dario Malchiodi video lectures 2020

Index

Chapter 1
HDFS, MapReduce and Spark

When do we say that we have enough data so that we can say we are dealing with the so called *big data* paradigm? Different people, and especially people from different fields, will tell you a different threshold that allows us to say if we're dealing with big data or not. From a computing point of view, the idea is that resources that traditionally were able to process data suddenly are no more suitable.

For instance, think about a very simple problem where we have to compute the product $A \cdot x$, where A is a $n \times n$ matrix, and x is a vector of n elements. If we take A as the matrix representing the connections among web pages, then this product will be huge. An estimation of the number of web pages is in the order of 10^9. Therefore, we would have 10^{18} elements. Moreover, if each element is a float, then A will require $8 \cdot 10^{18}$ bytes to be stored, that is about 8 exabytes. This means that we would not be able to store this matrix using a standard computer, a computer having memory of the magnitude of some terabytes.

The first solution that comes to mind is to build a computer with that quantity of memory that we need, but this would be really expensive. Having a unique hard disk with that kind of memory is very costly, and if it breaks, then the investment, and most importantly data, are gone. This is not a feasible solution.

The way to go is not to build a single supercomputer, but to think in a distributed fashion: use many "simple" and cheap computers (*commodity hardware*) so that we can buy a huge number of them. By doing this, we are also able to duplicate data so that if we encounter a failure in one of the hardware, we do not lose data since there is a copy on another machine.

1.1 HDFS

The Hadoop Distributed File System dates back to 2007. It is not only a file system but more of a distributed processing environment that also allows to distribute the processing of files, which are all stored on a distributed FS. This kind of systems are not so different in the architectural point of view from standard FS.
Files are divided into the equivalent of blocks, although now each distributed file system has its own nomenclature; what in a standard Unix-like FS is called *block*, now it is referred to as a *chunk*. The full choice for a chunk is 64 Megabytes. Each chunk as its own replica, which is one or more copies of that chunk, and such copies typically are stored in different places. The various commodity hardware that make up cluster on which Hadoop is installed are organized hierarchically.
The standard default behavior is that of replicating a chunk once in the same rack, and once in another rack. This means that if the information contained in a chunk disappears, the first thing we try to do is locating the replica in the same rack so that we can access it fast. But if we have a network problem, for instance overall rack has been cut off, we are still able to find the same information in another rack.
There exists a *Write Once - Read Many* policy: a file is either already obtained from somewhere else, or is simply written once and then it is only accessed through reading it and this is not a reading in a classical style, it's reading by typically doing a big pass on the overall file.

FOR THE PART ON WRITING, READING, REPLICAS AND FAILURE HANDLING IN HADOOP, PLEASE REFER TO THE PDF SLIDES.

1.2 MapReduce

The Hadoop FS is composed by two main actors: one is the distributed file system and the other is the distributed computational framework which is called *MapReduce*. The idea is that it process a file which is contained in the distributed FS and transforms it into another file that at the end of the process is saved in the same distributed file system.
Let's say we have a file in HDFS that has been divided into several chunks. The content of each chunk should be interpreted as a series of key-value pairs. We are able to divide the overall file into some "atoms" and we want to process those atoms. For instance, if we a have text file, we could say that an atom is a line of the text. The truly important part here is that once we subdivide the overall file in several lines, we cannot have a line which is partially included in one input chunk and partially included in another one. If the input is not already available in key-value format, we can use an artificial key. For instance, we could say that each line is a value and the corresponding key is the number of the line.
The idea is that those input chunks are all stored in one or possibly several data nodes and these nodes are actually computers, so they are also able process the data they carry. A first processing stage can occur where data resides. The mantra here is that we don't want to send data where computation occurs, we want to make the computation where data resides. Thus, on each node, we will launch a task called *map task*. The map tasks operating in parallel process each of those atom and each time they output zero or more key-value pairs. Be aware that these are not key and values in the sense which we would use if we were speaking about a dictionary: we might have the same key which is associated to different values in different places.
After the map task we have the *shuffling*, in which all these pairs are considered together. This process allows us, for any value of key that occurs at least once in the overall set of key-value

pairs outputted by our map tasks, to group all the values which are associated to that key.
Then, we run the *reduce* task. Reduce takes those complex information as input and outputs again zero or more key-value pairs that is gathered together into a new file that goes back into the distributed file system.

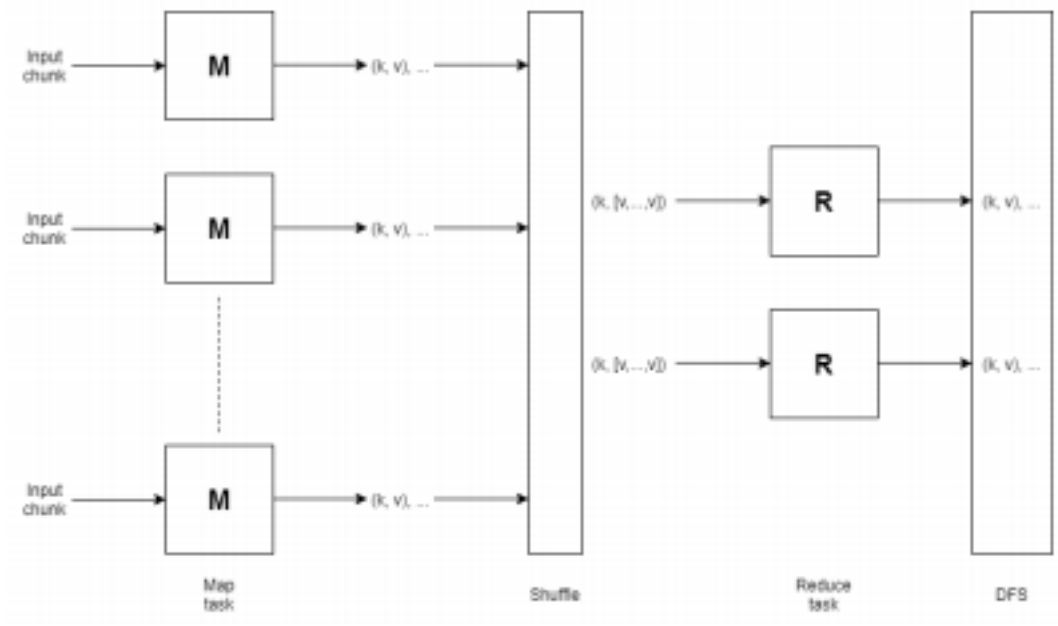


Figure 1.1: Schema of MapReduce

Let us see the classical example used in order to introduce the working of our MapReduce job. We have a corpus of files which we could think of as joined together into single big file. That big file is organized in HDFS in chunks and we further say that each chunk is logically subdivided in lines. Each line will be an atom which will be processed by a map task.
Actually, a map task need as input a pair having a key and value. The line itself will be the value. The key will be useless to us, so we could think that each line is coupled with the line number or an offsetting bytes from the beginning of file.
When the map task process the first, it tokenizes it and for each word it will emit a pair where the word is the key and the value is one. This process will be repeated for each line in the first chunk and, possibly in parallel, all the chunks of our big file.
For instance, let's say that the map task outputs (w, 1),(w^0, 1) and (w^{00}, 1). The key (i.e.: the word) may change, but the value, in this case, will always be one.
After the shuffling phase, for each of the words that were found, we will have each unique word coupled with all the values that occur somewhere with that word. Since those values will always be one, we will have a list variable length, but whose contents are always one: (w, [1, ..., 1]),(w^0, [1, ..., 1]),(w^{00}, [1, ..., 1])..... Each of this rich information will be fed to a reduce task that will simply compute the length of this list or, equivalently, it will sum up their content. Let's say the list contains 42 elements: this means that we have exactly 42 pairs where the key is w, which means that in the corpus of documents the word w occurs exactly 42 times. This will be done for each pair, again possibly in parallel, and all those pairs will be gathered up and they will form a new file in our distributed file system.

The computation of a MapReduce job is itself resilient. Failures might impact not only the dis tributed file system per se. So if, for instance, a data node has some kind of hardware failure, its contents will not be accessible anymore but actually, due to replication, the data node will be able to find the lost content somewhere else, most of the times.

Actually a hardware failure might occur in a node when it is making a computation. In this case, the overall computation is not lost because if a map task has to be shutdown, it will be simply sufficient to reprocess all the chunks that were contained on that data node. This works for a map task, and the idea is similar for a reduce task, even if the input of the reduce task is not contained in the distributed file system but it is stored temporarily in local storage.
The fact that we require the input of MapReduce job to be organized as a set of key-value pairs allows us to chain several MapReduce steps: as the output of the reduce phase is contained in a file that is organized in several key-value pairs, we want to take that output and use it as input for a further step of MapReduce.

Let us see another typical example of a MapReduce job. We want to compute the product of a square matrix A and a vector x of length n. The generic entry of the matrix A will be A_{ij} and that of the vector x will be x_j. Their product will be a vector p:

$A \cdot x = p$

of length n with generic entry p_i, where

$$p_i = \sum_{j=1}^{n}$$

What happens if the matrix is so big that it does not fit into a standard hard disk? If it does not fit the hard disk, it will not even fit the main memory. Nevertheless, we are still able to compute this product using MapReduce. However, in order to do that, we need to assume that the vector x can fit the main memory so that each map process might load in main memory the overall vector x.
The first thing we need to do is to properly encode the matrix A in order to save it on our distributed file system: for each non-zero entry of the matrix we will store a triplet (i, j, a_{ij}), that is, we are saving to disk the representation of a sparse matrix.
The overall file will contain all the triplets and it will be organized in chunks. Each chunk will contain several of these triplets. The atom that will be processed by each call to the map task will be a single triplet.
In order to make the map task work, we need to organize the triplet as key-value pair. We have several possibilities: we could say that the first value is a key and the second and third one are grouped together in pair as the value; or that the key is the first two values and the value is the third one in the example of; or again that we have a fake key and the value is the overall triplet.
A map task will receive the triplet (i, j, a_{ij}) and it will output a key-value pair $(i, a_{ij}x_j)$. The position of the vector which we need to access is contained in the second element (i.e.: j) of the triplet. After the shuffling we will have that a key i will be associated to $(i, [a_{i1}x_1, a_{i2}x_2, ..., a_{in}x_n])$. Note that we have no guarantee that this list will be sorted as we wrote here, but actually this is not important because our reduce task will simply compute a sum of the contents of this list.
The reduce task will output a pair $(i, \sum_{j=1}^{n} a_{ij}x_j)$ and because of equation 1.1 we have

$$\square$$
$$\square_{i,} \sum_{j=1}^{n}$$

This means that in the distributed file system we will find a set of pairs where the first element denotes the order of one component of the product and the second element is the value of that component. Therefore, we will simply need to sort those pairs for non-decreasing first component and then consider all the second components: they will give us our product vector p.

If we cannot ensure that the vector x can be stored in main memory, we will need to split both the

matrix and the vector. Let's split the matrix into two vertical hulls and call them A_L and A_R. Let's split also the vector x, this time horizontally, so that we will have x_{up} and x_{dw}. When we need to compute the product $A \cdot x$, we can instead compute the product $A_L \cdot x_{up}$ and similarly $A_R \cdot x_{dw}$. Therefore, we can organize two different MapReduce processes each one dealing with just one "sub-product".

6

Let's say that A is a 4×4 matrix and x is a vector of length 4 that does not fit into main memory. Even if we split the matrix and the vector as we discussed above, the result of the product $A_L \cdot x_{up}$ will be a vector of length 4, just as x. Thus, it will still not fit into main memory. In a case like this, where the split is not sufficient to make the vector fit into main memory, nothing prevents us from performing further subdivisions. For example, we could split the matrix A both horizontally and vertically.

1.3 Relations

Let us see how we can use map reduce in order to compute the basic operation of relational algebra. A relation can be written as $R(A_1, ..., A_n)$ where R is the relation and $A_1, ..., A_n$ are the elements (*fields*) the relation is built upon.
A relation groups some tuples $(a_1, ..., a_n)$ where a_1 is a possible element of this set A_1 and so on and so forth so that

$R(A_1, ..., A_n)\ 3\ (a_1, ..., a_n)$ (1.2)

in order to say that this tuple is a tuple of these relation.
Let's say that the relation R has been saved on HDFS and each of the tuple can be accessed in a file, so that a chunk of this file is divided into processable atoms where each atom is a tuple. There are several very simple operation that establish an algebra on relations.

1.3.1 Selection

The simplest operation is the *selection* $\sigma_c(R)$, where R is a relation and c is a condition. The idea is that selection acts like a filter and returns a new relation that is a subset of the original relation containing only the tuples of R that satisfy the condition c. The condition c is something that can either be true or false and depends on n values.
For each tuple t in R, we pass t to a map process that is aware of the condition c and thus can easily understand if this tuple satisfies or not the condition c. The output of the map task will be different according to the fact that the condition c computed on t evaluates true or otherwise. Recall that map should always output key-value pairs and, actually, also the inputs of map should be composed by key-value pairs. Again, like in the case of word count, the key we associate to our tuple is not really important as we are in the first step and we will only have one map process followed by a reduce process just because we are not joining this MapReduce job with a further MapReduce step. If the condition c on t evaluates true, the map process will output the pair (t, t), otherwise it will output nothing. The reduce task will actually do nothing because if we consider all the tuples that are output by map, we obtain only those of R that satisfy the condition c, so the reduce step will be equivalent to the identity function. Note also that given a tuple t, we will not have another equal tuple. Thus, the reduce task will take as input $(t, [t])$ and will output (t, t).

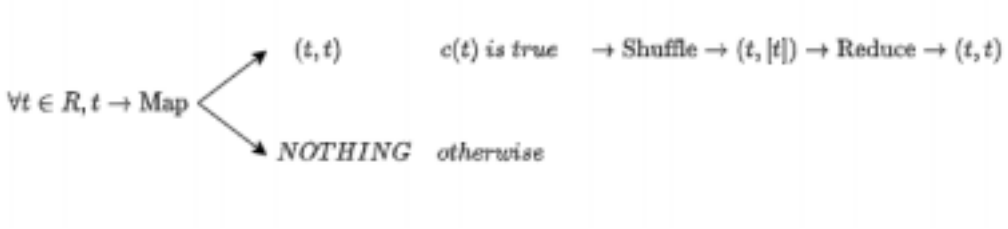


Figure 1.2: Selection relation

1.3.2 Projection

While the selection operations filters out rows (i.e.: tuples) in a sort of tabular representation, the *projection* operation filters out the columns that are the attributes (i.e.: fields) of the relation. It can be written as $\Pi_s(R)$, where *s* is a subset of the fields of the relation *R*.
Again, each tuple *t* in *R* is given as input to a map process and it can simply build a new tuple (t^0, t^0) by excluding all the components that are not in *s*. The reduce step is similar to the one performed in the selection. However, we could have in principle two different tuples t_1 and t_2 in *R*; once we have removed the attributes that we don't want to consider, we obtain two tuples t^0_1 and t^0_2 which happen to be equal. This means that for a given reduced tuple, we might have one or more keys associated: (t^0, [t^0, ..., t^0]). Regardless of the number of elements the reduce task finds in this list, it will simply output (t^0, t^0).

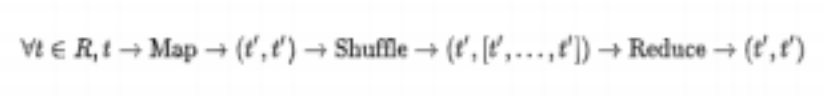


Figure 1.3: Projection operation

1.3.3 Union

We have two relations *R* and *S* and we want to build a union of those two relations, that is $R \cup S$, intended as a new relation that contains all the tuples of *R* and all the tuples of *S* without repetition. Each tuple *t* in *R* will be passed to a map process both as key and value, like (*t, t*). We will do the same for each tuple in *S*. We now have two possibilities: either *t* just belongs exactly to one of the two considered relations, so the map task will output again (*t,* [*t*]) and reduce (*t, t*); otherwise if that tuple belongs to the intersection of the two relations, it will be output twice from the map task, like (*t,* [*t, t*]), and the reduce task will output (*t, t*).

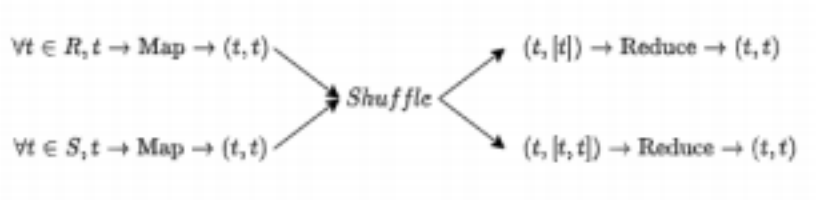


Figure 1.4: Union relation

1.3.4 Intersection

Intersection $R \cap S$ is a relation that contains only the tuples that are contained in both relations. The map step will be analogous to the one performed in the union, but the reduce step will accept only the case in which map outputs (*t,* [*t, t*]) because this ensures that the tuple *t* belongs to both relations *R* and *S*.

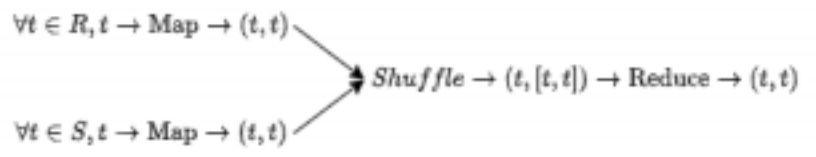


Figure 1.5: Intersection relation

1.3.5 Difference

We are interested in a relation containing all the tuples in *R* which are not contained in *S*. The difference is indicated as *R**S*. We will need two different map processes, each tailored on one relation only. Therefore, each tuple *t* in *R* will be passed to the map task Map_R and it will output (*t,* "*R*"); similarly, each tuple *t* in *S* will be passed to the map task Map_S and it will output (*t,* "*S*").
For the reduce task, we will have three cases:

1. We have a tuple *t* contained only in *R*. Thus, reduce will see (*t,* ["*R*"]).

2. We have a tuple *t* contained only in *S*. Thus, reduce will see (*t,* ["*S*"]).

3. We have a tuple *t* contained both in *R* and *S*. Thus, reduce will see (*t,* ["*R*", "*S*"])

However, we want to output something from reduce only in the first case because that case tells us that *t* is contained only in the relation *R* and not in *S*. So, reduce will output (*t, t*). In the other two cases, reduce will output nothing.

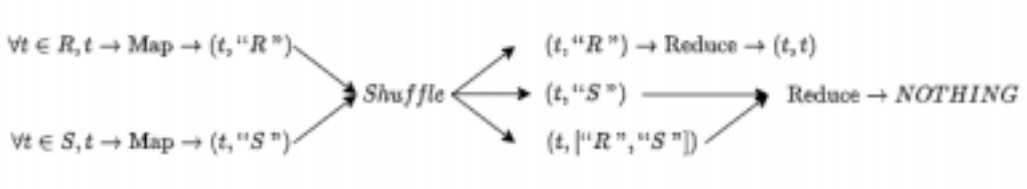


Figure 1.6: Difference relation

1.3.6 Join

Let us say that we have a relation $R(A, B)$ and a relation $S(B, C)$. We want to compute the join $J(A, B, C) = R(A, B) \bowtie S(B, C)$. The join between those two relations is a new relation having three attributes A, B and C and $(a, b, c) \in J \Leftrightarrow (a, b) \in R$ and $(b, c) \in S$.

Again, we will have a first stage with personalized map steps. Each tuple (a, b) in R will be passed to the map task Map_R that will output $(b, (a, \text{“}R\text{”}))$, that is, the common element b and a couple signaling that the element a belongs to R. Similar reasoning for each tuple (b, c) in S: (b, c) will be the input of Map_S that will output $(b, (c, \text{“}S\text{”}))$.

Let us consider a generic element b for the attribute B. It will be sent to reduce after the shuffling phase and it will be paired with a list containing several pairs of the first type and several pairs of the second type: $(b, [(a_1, \text{“}R\text{”}), ..., (a_n, \text{“}R\text{”}), (c_1, \text{“}S\text{”}), ...(c_m, \text{“}S\text{”})])$. Note that there is no guarantee that the elements will be sorted in the order displayed above. However, the reduce step does not care about it since it will only need to know how to tell that an element belongs to R rather than to S. This can be done by simply sorting these pairs on the second element.

Now, we can write a table having the elements from the first relation as columns and the elements

from the second relation as row. Each entry in this tabular structure will be a possible pair having one element from the first relation and one element from the second relation that will be contained in our join.

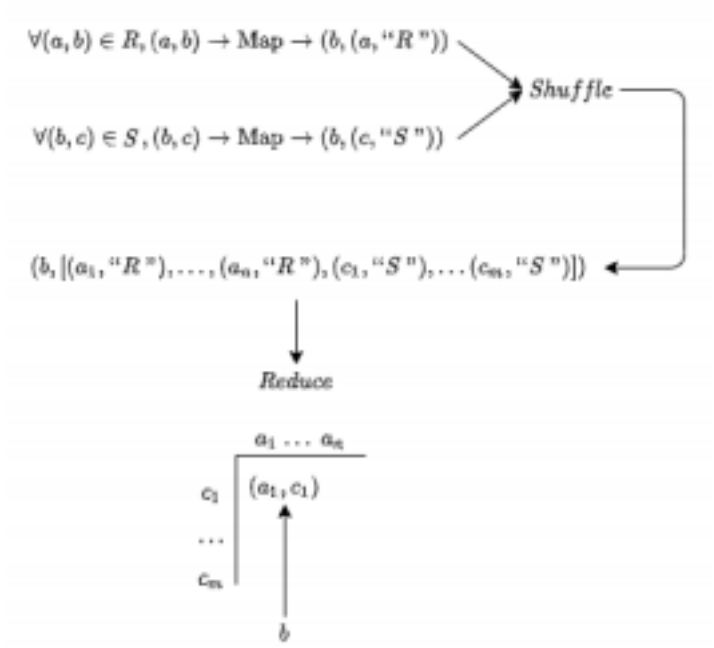


Figure 1.7: Join relation

1.3.7 Matrix multiplication

Being able to join relations helps us write a MapReduce job that is able to compute the product between two matrices. Let *A* and *B* be two matrices such as

$A = [a_{ik}]_{m \times n}$
$B = [b_{kj}]_{n \times o}$

and let their product be
$P = A \cdot B = [p_{ij}]_{m \times o}$
where the generic entry of *P* is

By abuse of notation, let us introduce a relation *A*(*I, K, V*) linked to the matrix *A*, such that *A* ≡ *A*(*I, K, V*), where *I* and *K* is the set of all possible values for the indexes *i* and *k* respectively, while *V* is the set of all possible values for the entries of the matrix *A*. Moreover, the generic tuple contained in the relation will be (*i, k,* a_{ik}), so that

A ≡ *A*(*I, K, V*) *3* (*i, k,* a_{ik})

Let's do the same reasoning for *B*, so that we end up with

B ≡ *B*(*K, J, W*) *3* (*k, j,* b_{kj})

Let's now compute the join *A ./ B* (on the *k* attribute)

$A \,./\, B\ 3\ (i, k, j, a_{ik}, b_{kj})$

This join contains the two components a_{ik} and b_{kj} that will lead us to the generic entry p_{ij} of the product P. The simplest way to obtain the product is to use MapReduce twice. That is, we will process our matrices with the first MapReduce step, it will produce a result that would be further processed with a second MapReduce operation.
The specialized map task for A will have as input (i, k, a_{ik}) and will output $(k,(\text{“}A\text{”}, i, a_{ik}))$; similar reasoning for the map task for B, that will take as input (k, j, b_{kj}) and output $(k,(\text{“}B\text{”}, j, b_{kj}))$:

$\forall i = 1, ..., m,\ \forall k = 1, ..., n : (i, k, a_{ik}) \rightarrow \text{Map}_A \rightarrow (k,(\text{“}A\text{”}, i, a_{ik}))$
$\forall k = 1, ..., n,\ \forall j = 1, ..., o : (k, j, b_{kj}) \rightarrow \text{Map}_B \rightarrow (k,(\text{“}B\text{”}, j, b_{kj}))$

After shuffling, the reduce step will receive as input

$(k, [(\text{“}A\text{”}, 1, a_{1k}), ...,(\text{“}A\text{”}, m, a_{mk}),(\text{“}B\text{”}, 1, b_{k1}), ...,(\text{“}B\text{”}, o, b_{ko})])$

Like in the join example, the reduce step is gonna to tell all the elements coming from the matrix A from all the elements coming from the matrix B and it will build a tabular structure. Note once again that here we have written all those element elegantly sorted, but it will not be the case. So, the first operation that will be done by reduce is a proper sorting of those data, firstly on the first element of the triplet in order to have elements from the first matrix first, and then on the second element too in order to build the tabular structure that will look like:

$1,a_{1k}$ ... m,a_{mk}
$1,b_{k1}$ $(1,1,a_{1k}b_{k1})$
...
o,b_{ko}

The reduce step will output a key-value pair, where the key is the pair of the two indices and the value is the corresponding value in the tabular structure:

$\forall i, j : ((i, j), a_{ik}b_{kj})$

Now, we need to collect all the values output by this step and in the other reduce steps for the other values of k. If we gather together all those outputs for ranging k, but fixing a value of i and j, we will have at hand all the single products that need to be summed in order to get a value for the entry of the product. At the second step we only need reduce; therefore, map will output its input as is

$((i, j), a_{ik}b_{kj}) \rightarrow \text{Map}_2 \rightarrow ((i, j), a_{ik}b_{kj})$

so that reduce will receive as input all the values $a_{ik}b_{kj}$ for key (i, j):

$((i, j), [a_{i1}b_{1j}, ..., a_{in}b_{nj}]) \rightarrow \text{Reduce}_2 \rightarrow ((i, j), p_{ij})$

1.4 Spark

In order to process data in a distributed fashion, we refer to Apache Spark. Apache Spark can perform the same map reduce operation introduced by Hadoop and it also provides more complex tools in order to process data. Moreover, we will see that it is more efficient than Hadoop. Spark is

heavily based on the concept of *resilient distributed dataset* (RDD). It's somehow similar to the concept of chunk in Hadoop, but now what is divided into chunks is not a file on the file system, but it's an abstraction of a set of data which need to be processed. It is called distributed because

also Spark is a distributed framework that it is used in a network of computers and the computation is done in parallel by the nodes in this network.
It is also called resilient as a synonym of robust. If a failure happens in a node of the network which is used by Spark, most of the time you we will not need to restart the overall computation. The first big difference between the basic MapReduce and Spark is that MapReduce is built around the concept of key-value, while this is not the case in Spark. Spark always thinks about an abstract RDD which could be composed of pairs having (or not) a key and a value.
In order to understand how RDD are created, we have to start from actual data. The main entry point to Spark is called a sparkContext. So, the first thing which needs to be done is to obtain an instance of this sparkContext and then, if we wanted to create an RDD for instance, we could call the parallelize method on the sparkContext.

Regarding the architecture of Spark, we have three big actors:

• *Driver* : the driver is actually the main user of spark and it typically resides on a standard computer where we write the code (i.e.: the client) to access the sparkContext.

• *Cluster manager* : activated when we execute the code on the driver. It distributes the compu tation over a set of nodes that will be provided the data.

• *Nodes*: the servers of the network.

Inside each node we have an *executor* that contains the tasks to be performed, the cache and some RDD blocks. Needless to say that the cache is crucial to guarantee better performances, since it allows us to avoid writing the results of the tasks to the disk when they need to be used in another task. The presence of this cache is one of the differences that make Spark more efficient than Hadoop (recall that Hadoop always writes the results of map and reduce tasks on disk).
The operation that can be performed with Spark fall into big categories called respectively *transfor mations* and *actions*. A transformation is something that is applied to a RDD and creates another RDD. Map, for example, is a transformation. Actions process a RDD and bring the result into the main memory of the driver. The result of an action must be something of manageable size because otherwise the main memory of the driver will not be able to contain it.

We previously said that Spark is based around the concept of RDD and not on key-value pair. How ever, nothing prevents us to work with key-value pairs. Typically, we can do this by building tuple of two elements in Python. Actually, there are some operations in Spark that actually expect to process key-value pairs, like the functions reduceByKey, groupByKey and sortByKey of a RDD instance.

What happens under the hood when we run an operation in Spark? Spark organizes the overall computation as a DAG (directed acyclic graph), and it is able to spot if the computation we are doing can be somehow simplified; if so, the computation gets simplified, and we will refer always to

the simplest DAG that describes our computation. Then, the simplest DAG will be used in order to create the single tasks distributed to the overall nodes.

Chapter 2

Link Analysis

The term link analysis is often used in order to study relations, where a relation is a subset of the Cartesian product between two sets. That is, we have two sets of elements and we consider pairs of elements of this set so that a pair *AB* means that two elements *A* and *B* are in relation. Among the various formalism that allow us to describe our relation, we can use graphs if this relation especially when the relation involves pairs of element of the same set.

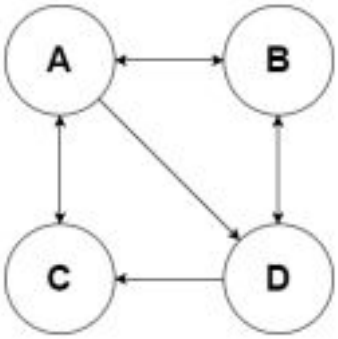


Figure 2.1: Simple directed graph

Here you see a very simple graph involving four nodes that will be the elements of a set on which we want to consider a relation. The edges allow us to find out which elements are in relation and, specifically, we can say that two nodes, for instance *A* and *B*, are in relation if there is an edge be tween them. Note that the graph is directed, so it doesn't necessarily means that one element being in relation with another one implies that we can reverse the relation. In this case, we have an arrow going from *A* to *B* and also an arrow going from *B* to *A*. This does not happen with *C* and the *D*: *D* in relation with *C*, but *C* is not in relation with *D*.

Despite the fact that link analysis can be applied to generic relations, the working example that we

will consider is about the web. Think that the nodes in our graph correspond to web pages and the arrows correspond to links; the relation we are interested in studying is the relation that links a page *A* to a page *B* if there exist at least a direct hyperlink going from the page *A* to page *B*.
The study of this kind of web relation is deeply linked to finding out a way in order to rank pages according to their importance. This is a problem which is crucial if we want to build a successful search engine such as Google, and indeed this particular technique that we are going to speak about is called *PageRank*.

13
Why does Google need to rank web pages? Even for a very complex search query, we typically have a huge number of web pages as a response, that can be of several orders of magnitude. Usually, the average user does not go further than second page of results. Thus, in order for a search engine to be successful, it's really important that the results of a search query are properly ranked and that the most important (i.e.: the most pertinent results) are shown first.

How this ranking problem of web pages has been addressed in the past? Originally, the importance of a page was found out by looking at the content of that page itself. This is not the best thing to do because this criterion is vulnerable to web spam. Web spam has to do with pages that are modified by their own owners in order to increase the importance of the page, regardless of the fact that page is important or not. The main reason that leads one to do so is to increase traffic to the web page: if we have commercial ads on our web page and we increase the number of visits, then we typically increase our revenue derived from ads.
For instance, think of a criterion where the importance of a page is related to its content. We write a query that contains the term *music*. The result pages are ranked and ordered by self-established content pertaining to music. As the owner of the page that is about music, we are for sure able to modify the content of the page and add several other terms that somehow might be of interest to a search engine, such as politics, movies and so on, even if they are not relevant to a music query. These triggering terms let the page show up first in the search results of queries that do not regard music. They may be written with a very very small font or in the same color of the background of the page, so that they are present but the user does not see them appear.

The problem is that search engines cannot avoid this form of web spam. Thus, the concept of im portance based on the content of the page does not work. The solution is to base the definition of importance of the page on some kind of external criterion (i.e.: external from the content of the web page).
Here is where PageRank comes to the rescue: it defines a web page as important if it is linked by other pages which are in turn important. Even though this is a recursive definition, we can still get something manageable by relying on a sort of fictious process where we imagine a huge number of users (*surfers*) that are placed on the web and, at given time, each surfer randomly chooses uniformly among all the out-links of the page and it follows that link. When they have changed their position, we can repeat this process and if this process converges in the end, we will use the fraction of random surfers that are on a page as an index of the importance of that page.

2.1 Matrix properties

Property 1. *The determinant of a matrix is equal to the determinant of the transpose of the same matrix:*
$\det A = \det A^T$

Proof. We know that we can compute the determinant of A as

$$\det A = \sum_i$$

When we try to use this definition in order to compute the determinant of the transpose of A we get:

$\det A^T = \sum$ e exchange the indexes
e get:
i i

$$\sum_j$$

that is equal to the determinant of A.

□

Property 2. *A square matrix A and its transpose A^T have the same eigenvalues. Proof.* The characteristic polynomial is the quickest way to find all the eigenvalues of a matrix: $\det(A - \lambda I) = 0$

This polynomial will have the number of roots (if there is any) equal to the dimension of A, and each root will identify an eigenvalue of A. Exploiting the results of Property 1, it should also be that

$\det(A - \lambda I)^T = 0$

When we distribute the transposition operator we obtain

$\det(A - \lambda I)^T = \det(A^T - \lambda I) = 0$

Therefore, the roots of the characteristic polynomial that allows us to find the eigenvalues of A are equal to the ones that allows us to find the eigenvalues of A^T.

□

Property 3. *If a matrix A is row-wise stochastic, that is* $\forall i : \sum_j a_{ij} = 1$*, then* 1 *is an eigenvalue of A.*

Proof. Consider a generic element $(A\underline{1})_i$ of the product $A\underline{1}$, where a_{ij} is the generic element of A and 1 is the generic element of $\underline{1}$. It will be

$$(A\underline{1})_i = \sum_j$$

because the elements of a row of A sum up to 1. Thus, independently of i, the i-th component of $A\underline{1}$ is 1 and therefore

$A\underline{1} = \underline{1}$

If we multiply the right-hand side by 1 we get

$A\underline{1} = 1 \cdot \underline{1}$

that is in the same form of $Av = \lambda v$, so $\lambda = 1$.

□

Property 4. *If a matrix is column-wise stochastic, then* 1 *is an eigenvalue of the matrix.*

Proof. The transpose of a row-wise stochastic matrix is a column-wise stochastic matrix. Thus, if we take a row-wise stochastic matrix A, we have that $A^T$is column-wise stochastic. Property 3 tells us that 1 will be an eigenvalue of A and, by exploiting Property 2, we say that A and A^T will have the
same eigenvalues. Therefore, 1 will also be an eigenvalue of A^T.

□

Property 5. *If a matrix A is row-wise stochastic, then A^k will also be row-wise stochastic for any integer k.*

Proof. We will prove this via induction. Start from $k = 1$: $A^1$is A itself, so it still is row-wise stochastic. Let's now say that $A^k$is row-wise stochastic and let's prove that A^{k+1} is again row-wise stochastic. The generic entry of A^{k+1} will be a^{k+1}_{ij}. Be aware that while A^{k+1} means A to the power
of $k + 1$, the superscript applied to a is just to indicate that a^{k+1}_{ij} is the ij-th element of the matrix
A^{k+1}. Knowing that $A^{k+1} = A^k \cdot A$, we can write it as

$$a^{k+1}_{ij} = \sum_j \quad \sum_j$$

By hypothesis of our induction step, we know that A^k is row-wise stochastic, so $\sum_s a^k_{is} = 1$ and therefore

$$a^{k+1}_{ij} = \sum_j$$

Moreover, we also know that A is row-wise stochastic, so $\sum_j a_{sj} = 1$ and therefore

$$a^{k+1}_{ij} = \sum_j \quad i$$

□

Property 6. *The highest eigenvalue of a column-wise stochastic matrix A is 1.*

Proof. We already know that 1 is always an eigenvalue of a column-wise stochastic matrix. We need to know that there are no other eigenvalues of the same matrix which are higher than 1. So let us suppose that, conversely, such eigenvalue exists:

$\exists \lambda > 1$ is an eigenval. of A

As A and A^T have the same eigenvalues we can also say that λ is an eigenvalue of $B = A^T$. Note that as A is column-wise stochastic, B is row-wise stochastic and, according to Property 5, we have that $\forall k : B^k$is row-wise stochastic. Now from the fact that B has λ as eigenvalue we obtain that

$Bv = \lambda v$

Similarly, we can write
$B^2v = B \cdot (Bv) = \lambda Bv = \lambda^2 v$

Generalizing, we obtain
$\forall k : B^k v = \lambda^k v$ (2.1)

We can rewrite eq. 2.1 by focusing on the components of the vectors. The generic i-th component of the quantity on the left is the i-th component of the result of a product between matrix and vector:

$$X\,j \qquad v_i$$

On the one hand, as by our hypothesis we know that $\lambda > 1$, when we raise such a number to a generic positive power it becomes greater and greater. As we have not made any assumption on k, we can think that k is big enough in order to have

$\lambda^k v_i > G$ arbitrarily high

Let us rewrite the left part of the equality using the maximal component of the vector v. As the maximal component is always greater or equal than the remaining component, we can write

$$X\,j \qquad \langle\, j$$

Note that v_{max} does not depend on the summation index, so we can bring it outside the sum and on the other part of the inequality: $X\,j$ $_{ax}$

Here we are computing the sum of the element in the i-th row of a row-wise stochastic matrix, and this sum is equal to 1. Moreover, since G was arbitrarily high, also $\underline{G}$
$v_{max}$is arbitrarily high. We found
out that a constant is greater or equal than an arbitrarily high value, which is not possbile. Thus, we reject the hypotesis that there exists at least an eigenvalue of A which is strictly greater than 1. If eigenvalues of A exist, their value need to be lower or equal to 1. Finally, as we already know that such a matrix that is column-wise stochastic, it has an eigenvalue exactly equal to 1, and we can conclude
and say that actually the highest eigenvalue of a column-wise stochastic matrix A is equal to 1.

2.1.1 Power method

The *power method* allows us to numerically find the eigenvector corresponding to the highest eigenvalue in a given matrix.

Consider a $n \times n$ matrix A whose elements will be a_{ij}. It will have $\lambda_1, \lambda 2, ..., \lambda_n$ eigenvalues and they are sorted so that $\lambda_1 \geq \lambda 2 \geq ... \geq \lambda_n$. Let us associate each eigenvalue to the corresponding eigenvector $v_1 \geq v_2 \geq ... \geq v_n$. If we consider jointly the n eigenvectors, they form a orthonormal basis. Being these values a basis, if we take any vector x_0 we can always express it as a weighted sum of the eigenvectors:

$x_0 = \alpha_1 v_1 + \alpha_2 v_2 + ... + \alpha_n v_n$

We actually don't know the values of each α, but that is not important at the moment. Let us define a vector x_1 as

$x_1 = Ax_0$
$= A(\alpha_1 v_1 + \alpha_2 v_2 + ... + \alpha_n v_n)$ $v_1 + \alpha_2 \lambda_2 v_2 + ... + \alpha_n \lambda_n v_n)$
$= \alpha_1 A v_1 + \alpha_2 A v_2 + ... + \alpha_n A v_n$ 3)
$= \alpha_1 \lambda_1 v_1 + \alpha_2 \lambda_2 v_2 + ... + \alpha_n \lambda_n v_n$

recalling that v_1 is the eigenvector of A corresponding to λ_1 and that $Av_1 = \lambda_1 v_1$. Similarly to what we did for x_1, we can define a new vector x_2 such that

$x_2 = Ax_1$
$= A \cdot Ax_0$
$= A^2 x_0$

where the last equation we used eq. 2.2. Through distribution of this product and recalling that v_i is the eigenvector of A associated to λ_i, eq. 2.3 becomes equal to

$\alpha_1 \lambda_1 A v_1 + \alpha_2 \lambda_2 A v_2 + ... + \alpha_n \lambda_n A v_n = \alpha_1 \lambda^2{}_1 v_1 + \alpha_2 \lambda^2{}_2 v_2 + ... + \alpha_n \lambda^2{}_n v_n$

since $Av_i = \lambda_i v_i$.

If we continued for x_3, x_4..., at a generic step t of this iteration we would obtain $x_t = \alpha_1 \lambda^t{}_1 v_1 + \alpha_2 \lambda^t{}_2 v_2 +$ $... + \alpha_n \lambda^t{}_n v_n$

$= \lambda^t 1$

α_n

Since the eigenvalues are sorted, we have that $\Delta 2$

This means that Δn oreover, raised to t, it becomes even smaller.

λ_1 is approaching zero. As we continue to create elements in the sequence $x_1, x_2, ...$, the contribution of α_2 mes negligible.

$+ \alpha_n \Delta\Delta \lambda_1$

Therefore, as t increases eq. 2.4 is well approximated by

$x_t \approx \alpha_1 \lambda^t_1 v_1$

Recall that an eigenvector is not unique; there are infinite eigenvectors all with the same direction. So, if v_1 is an eigenvector, also $\alpha_1 \lambda^t_1 v_1$ is an eigenvector and it will be equivalent to 1. Thus, if we want to find the eigenvector corresponding to the biggest eigenvalue, we can use the power method, which consists in repeatedly multiplying the matrix by any vector; we take the result and we multiply it by the matrix and we repeat. In practical terms, we can stop once we obtain that one of our vectors in the sequence $x_1, x_2, ...$ is sufficiently close to the previous one.

17

2.2 PageRank

In order to formalize the intuition behind PageRank, we have to rewrite a graph in a analytical form. We previously said that the random surfer positioned in one page will have to choose uniformly at random among the out-links. Referring to Figure 2.1, if the surfer is positioned in *A*, it can go with probability $\frac{1}{3}$to *B*, *C* or *D*. Thus, we rewrite this graph in terms of transition probability matrix, that is a square matrix whose dimension is equal to the number of pages we are considering ($n = 4$ in this case), and each entry of this matrix m_{es} is the probability of moving from the start node *s* to the end node *e*:

$$M = \begin{matrix} & & \frac{1}{2} & C\frac{1}{3}0 & 0 & \frac{1}{2} & D \\ & & & & s & & \end{matrix}]_{n \times n}$$

A $0\frac{1}{2}1$ 0 □□□

The columns represent the starting nodes while the rows are the end nodes.

We are interested in the probability that surfers are in one of our web pages. Let us introduce a vector *v* that depends on time:

$v(t) = [v_e(t)]_n$

Each element of this vector needs to be related to one of the web pages, so $v_e(t)$ is the probability of being in node *e* at time *t*.

Let us describe what happens when all random surfers perform a transition. That is, we know $v(t)$ and we want to compute $v(t+ 1)$. Consider generic element $v_e(t+ 1)$: it will be the probability of being in node *e* at time $t + 1$. The only possibility for a surfer to be in *e* at time $t + 1$ is if at previous time *t* the surfer was on some node and it transitioned from that node to *e*. Therefore, we can exploit the total probability theorem in order to write this probability as a sum of the products of the probability of being at node *k* at time *t* times the probability of moving from node *k* to *e*, where *k* ranges in all possible nodes:

$$\sum_{k=1}^{n}$$

We recognize that $\sum_{k=1}^{n} m_{ek} v_k$ is the product between the matrix *M* and the vector *v* at time *t*. Thus, we can write the vector *v* at time $t + 1$ as

$v(t + 1) = M \cdot v(t)$

2.2.1 PageRank convergence

It is possible to show that the numerical method at the base of the computation of PageRank should converge (in principle). Denote by $M = [m_{es}]_{n \times n}$ the transition matrix of the portion of web we are studying, where m_{es} is the probability of switching from the node s the node e: $m_{es} = P(s \rightarrow e)$. We know that the computation of PageRank is a process where M is repeatedly multiplied by a vector (not any vector) and, in order to apply the total probability theorem, we need that vector to describe our probability distribution so that their entries should be non-negative and the sum over all the entries should be one.
Let's start for instance with a vector r_0 that satisfies these requirement:

$$r_0 = \begin{matrix} 1 \\ n \end{matrix}$$

What happens if we describe another vector r_1 as $r_1 = Mr_0$? For the total probability theorem it will describe again our probability distribution over all the nodes of the portion of web we are considering.

This holds for all the elements of the possibly infinite sequence of vectors whose generic element would be the vector
$r_{t+1} = Mr_t$

Probably, this process will converge to some value, but we will come on later about that; let's see first why it should converge. The matrix M has been defined as a column-wise stochastic matrix. From Property 6 we know that its highest eigenvalue λ will be 1. Moreover, we also know that repeatedly iterating this kind of product, regardless of the initial chosen vector, brings this sequence to converge to a special vector r which is the eigenvector, which in turn corresponds to the highest eigenvalue:

$r_t \rightarrow r$ eigenvect. for $\lambda = 1$

If now write the definition of eigenvector of a matrix M for $\lambda = 1$ we get:

$Mr = \lambda r \rightarrow Mr = r$

We are sure that if the matrix M is column-wise stochastic, this sequence will converge to r and this vector will have some interesting spectral properties, but up to now the interesting thing to us is that it is supposed to converge. Moreover, if we have chosen r_0 as the description of a probability over the n nodes of the graph we are studying, we're also sure that each element in this sequence will be itself a probability distribution, and so it will be also r. Finally, we know that the process at the basis of PageRank should converge to something which is a probability distribution over a discrete set of n elements.

2.3 Web structure

The Web is often pictured out saying that it has the form of a bow tie because it highlights 3 different principal components of the web. The central one is called *strongly connected*

component. Taken any 2 pages belonging to this component, it's always possible to reach any of the two starting from the remaining one since there is a path between them.
The remaining two components are respectively called an *in* and *out* component of the bowtie. From the *in* component we can reach the *out* component either by going through the strongly connected component (in this case we cannot reverse the arrows, so we can have a path that is: *in* → *scc* → *out* but not the reverse) or we can bypass is using *tubes*. Through *tendrils* we can go out from the *in* component or reach the *out* component without passing through the strongly connected component. Note that you don't have the possibility of using a tube in order to start from the out component and get to the in component because that would make the three components collapse. So, if such a tube would exist, then we would have only one big, strongly corrected component with possible tendrils. Finally, we do have some isolated components: starting from any of those we cannot either reach the in, out or strongly connected component.
This particular shape of the structure of the web means that we have some *dead ends*: they are paths starting from a page that end up to a page with no more out links. This is considered to be bad because it impacts on how the PageRank computation converges. Moreover, there is also the possibility that in any of the components that we are studying we find some *page cycles*, better known as *spider traps*; these sub-structure of the web can impact negatively on how to PageRank computation behaves, too.

2.3.1 Dead ends

A *dead end* is a web page that has no out links. In Figure 2.2 you can see that we have three nodes, each referring to a web page. From the left node, we can reach the central one viceversa; from the central one we can reach either the left one or the right one, which happens to be a dead end. Let's see why a structure like this is not convenient when we want to compute the PageRank. Let's uniformly assign several surfers (the black points) to these three nodes and try to repeat the iterative process at

the basis of the computation of PageRank.
At each iteration we move the surfers according to the transition matrix. This means that, at $t = 0$, all the surfers at the left node will move to the central node (since the left node has only one link); half of the surfers of the central node will go to the right node and the other half will go to the dead end (since the central node has two outlinks, so $\frac{1}{2}$of the surfers will go to a node and $\frac{1}{2}$to the other node); the surfers on the dead end will disappear. Iterating we see that the total number of surfers decreases and all surfers will eventually disappear.
Therefore, running the PageRank computation process in a graph that has at least one dead end will cause the vector that contains a probability distribution over a discrete set of nodes to become a null vector.

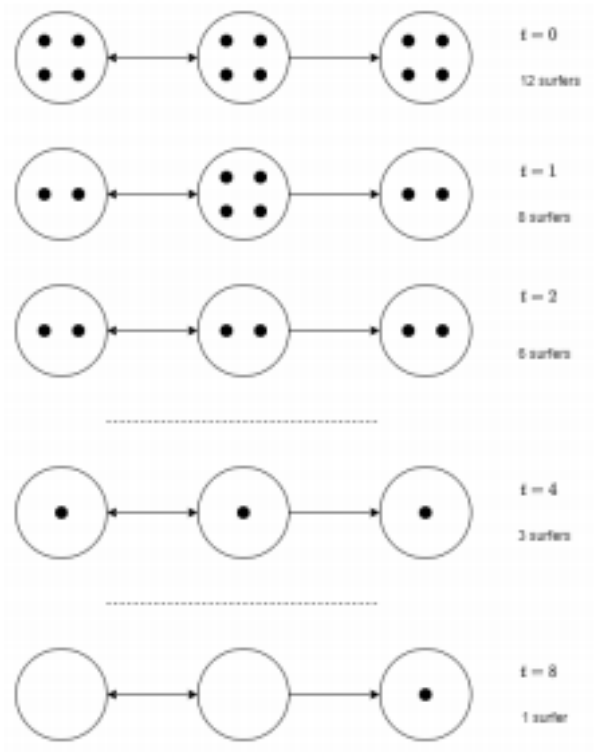


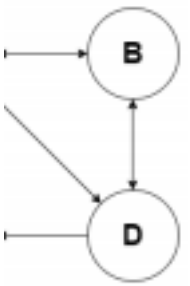
Figure 2.2: Surfers and dead ends

Let us consider a slightly modified version of the graph we've seen for PageRank in Figure 2.1, that is shown in Figure 2.3. When we compute the PageRank of such a graph we obtain a null vector, which is useless to us. How can we deal with this problem? The first thing that we can do is to simply cut out all the nodes that contain a dead ends. Note that cutting out these nodes does not guarantee that the resulting graph is free from dead ends. Indeed, if we remove the node *E* the resulting graph will still have a dead end, that would be the node *C*. We could remove *C* too and end up with a graph with nodes *A*, *B* and *D* only. Computing PageRank of this reduced version of the graph would lead us into assigning a PageRank value of $\frac{2}{9}$to node *A*,$\frac{4}{9}$to node *B* and $\frac{3}{9}$to node *D*.

However, we can still get back to the original graph and compute a pseudo PageRank value also for the original nodes. The idea is that we can propagate the PageRank values which we have already computed: each node is going to distribute evenly its own PageRank value through its out links. Considering back *C* and its nodes in the graph, as in Figure 2.1, it will receive part of the PageRank values from the nodes pointing to it. This means that *A*, that has three outlinks, will give $\frac{1}{3}$of $\frac{2}{9}$to *C*. Similarly, *D* will give $\frac{1}{2}$of $\frac{3}{9}$to *C*. In the end, *C* will have a PageRank value of $\frac{13}{54}$.

If we also include again node *E*, *C* will distribute evenly its PageRank value to it. Since *C* has only one out link, *E* will have a PageRank value of $\frac{13}{}$

distribution over the nodes. 2.3.2 Spider traps ever, this will not be anymore a probability



.3: Directed graph with dead end

A *spider trap* is a subset of the node in a graph where we have a cycle that does not contain any out link to nodes outside of it. At the PageRank computation process, all the surfers will be contained in spider trap, that are the nodes in the loop. While the PageRank for the remaining nodes will be zero, we do not obtain a degenerate vector since it still describes a probability distribution, but only on the subset of nodes that belong to the spider trap.

2.3.3 Teleportation

There is a variant of the process which we have seen that computes the PageRank value in a graph that can deal with the problems of spider traps and dead ends. We usually start from a vector

$$r_0 = \begin{matrix} 1 \\ n \end{matrix}$$

and then we describe a sequence of vectors described by

$r_{t+1} = Mr_t$

If we fix a real value $\beta \in [0, 1]$, we can modify this updated equation for PageRank such that the update becomes
$r_{t+1} = \beta M r_t + (1 - \beta) r_0$

For instance think that $\beta = 0.5$: we can give to it the interpretation that the probability that a random surfer, at each iteration, decides to uniformly select at random one of the out links and follows it (so continuing with the standard process at the basis of page rank) or, with probability $1 - \beta$, he could decide to magically teleport to any node of the network. So, if $\beta = 0.5$, we are saying that at each iteration a surfer flips a coin and, if it gives head, it selects at random one of out links and it follows it; otherwise it selects at random one page of the web and it teleports to that page.

Indeed, this variant is called *teleport method* or *taxation*. Why taxation? It comes from the idea that $1 - \beta$ can be considered as a tax that a random surfer should pay each time to a standard update process.

It's rather easy to see that this teleport variant can deal both with dead ends and spider traps. Let's suppose that we do have a dead end. All the random surfers that enter the dead end and that select the first part of the update process (i.e.: $\beta M r_t$) will disappear. However, a fraction of $1-\beta$ surfers will be redistributed evenly on the overall graph. Indeed, they could be redistributed in one of the node that is contained by our dead end, but if the number of nodes in the graph is high that would be extremely unlikely. Instead, if we do have a spider trap, we will be dealing with the same situation: again, a fraction of $1 - \beta$ surfers will be redistributed evenly on the overall graph and with high probability they will be teleported outside the spider trap.

Note that this variant can be also used in order to provide the so-called *topic-sensitive* variant of PageRank, which tries to tailor its computation process to specific class of users. From a query "Jaguar", a user who has been searching cars will likely get as first results the car brand, while a user who has been searching mostly animals will likely get the jaguar animal among the first results. However, it would be unfeasible from a computational point of view to perform different PageRank computations for each user, since the number of users is really high.

A possible solution to this problem would be classifying our users in several categories and form a PageRank factor for each category that will prioritize pages of a specific category. In order to drive the update process towards the pages of a specific category we will not use r_0 in the second part of the update process (i.e.: $(1 - \beta) r_0$), but we will use another vector that will only describe a uniform probability on all the pages of the web that speak of a given topic. So, if for instance we do have a set S that contains the indices of pages that talk about movies, then we could replace r_0 with

$r_{S} = \frac{1}{|S|} I_S$

where each element of I_S is 1 if $i \in S$, otherwise 0.

2.3.4 Spam farm

It is actually possible to cheat PageRank. That is, if we have access at to a sufficient number of web pages, we can structure them in a way that a specific page will have its PageRank value raised. Let's divide the overall n pages in three categories: inaccessible pages, accessible pages and controlled pages. The first one is, trivially, the set of all the pages which are inaccessible to the spammer. On the other hand, the so-called controlled pages are pages that can be shaped as the spammer wants since he/she has complete access to the content of those pages. We have a target page t and other m support pages; each support page has an out link to t and t has an out link to each of support page (thus, t will have m out links). The set of accessible pages contains

pages that are not fully controllable by the spammer, but he/she is able to put links from these pages (i.e.: comments in forums or blogs) pointing to our target page *t*. In this way, our target page *t* will have its own PageRank value raised. Let's see it with an example.
Let us call *y* the PageRank of *t* and *x* the PageRank of the accessible pages sent to *t*. The PageRank of any support page is rather easy to compute because the only in-link is the link coming from *t*, any support page will only receive a PageRank value from *t*, that is, *t* will distribute evenly its PageRank to the *m* support pages. Thus, if no taxation would occur, each support page would receive a total of $\frac{y}{m}$ PageRank quantity. But, as taxation occurs, we know that each page will receive only $\beta\frac{y}{m}$. For the same reason, each page in the web will also receive a part of PageRank because of teleportation. Therefore, the final PageRank of each support page will be

$$\text{PageRank of support page} = \beta \frac{y}{m} + (1-\beta)\frac{1}{n}$$

22
The PageRank of *t*, that we called *y*, can be computed by summing all the PageRank coming from outside the spam farm, that is *x*, as

$$y = x + m\beta \left| \frac{y}{m} + (1-\beta)\frac{1}{n} \right.$$

where *x* comes from accessible pages, $m\beta(\beta\frac{y}{m} + (1-\beta)\frac{1}{n})$ comes from support pages, and $(1-\beta)\frac{1}{n}$ comes from teleportation. Be aware that we are embedding into *x* the fact that not all the PageRank coming from the external is actually sent, but only a fraction; this is not important since we wanted to highlight is the dependency on *y* on the quantity of PageRank that comes from the outside.
Let's make some simplifications. First of all, let's not consider the part coming from teleportation: by doing this we will underestimate the PageRank of *t*, so keep in mind that its true value would be higher. Let us simplify the product coming from the support pages, and rewrite *y* as

$$y = x + \beta^2 \frac{y}{m} + \beta(1-\beta)\frac{m}{n}$$
$$(1-\beta^2)y = x + \beta(1-\beta)\frac{m}{n}$$
$$y = \frac{x}{1+\beta} + \frac{\beta}{1-\beta^2}\cdot\frac{m}{n}$$

We clearly see that the right part of the sum will increase the PageRank value of *y* as *m* increases. This means that, the more support pages we can bring to our spam farm, the more we can spam the PageRank value of our target page *t*.

2.3.5 TrustRank

The taxation variant of the process for computing PageRank can be adopted in order to avoid the problems caused by spam farms. The idea is the same at the basis of the specialized topic-sensitive PageRank, and in particular, we compute a specialized PageRank more devoted to trusted pages, giving us a result that can be compared to ordinary PageRank in order to detect pages that could be part of spam farm. TrustRank is acutally PageRank computed on a set of pages that we are sure are not part of spam farms, like institutional pages, academic websites, etc.
How do we compare these two different PageRank values? In order to do that, we introduce the

spam mass. It is computed as $\frac{r - t}{t}$

where r is the PageRank value and t is the TrustRank value. It is positive if $r > t$ and viceversa negative if $r < t$. The idea is that if these values are close to 0, or smaller than 0, then we will trust the node since the TrustRank is greater than the PageRank. However, we still need establish how much the spam mass should be close to zero in order to trust a page. This typically depends on risk you we willing to take on the specific application field.

2.4 HITS

There are several other procedures that captures importance of web pages and, in general, of objects that are organized into relations that we can describe using a graph. For instance, one of such procedure is called *Hyperlink Induced Topic Search* (HITS). This method makes an assumption: it partitions all the web pages, and in general the objects we are considering, into two big classes: hubs and authorities. An hub is a collector of pointers towards pages that speak of specific topic, and such pages are called authorities. Think, for instance, at the web information system of your University, you will probably find that there are pages that are devoted to speaking of a specific subject, for instance, dealing with international exchange programs; those pages are what we would call authority. These are pages that

act as pointers that contain several links to interesting resources. Instead, there are pages that contain all the links to the courses of a given degree: we would call such pages hubs. The HITS approach tries to model separately the degree of “hubbiness” and “authoritativeness” of each single page. In order to do so, HITS has to model the relation and it does so with a connection matrix. This connection matrix is slightly different from the one used in PageRank because it’s a boolean matrix that only encodes the existence or non-existence of links between pages. Let us call this matrix L and it will be

$L = [l_{ij}]_{n \times n}$

where n is the number of web pages of the portion of web we’re considering, or in general, the number of objects that we are studying within a relation. Moreover l_{ij} will either be equal to 1 if the node i contains a direct link to the node j; otherwise, zero.

Define two vectors that encode how each node of the graph is a good hub or a good authority: $h = [h_i]_n$, h_i = hubbiness of i-th node

$a = [a_i]_n$, a_i = authoritativeness of i-th node

In order to compute these two vectors, we will show directly the equivalent of the iterating algorithm that converges to acceptable values for h and a. We have an initialization step, step 0, where we provide initial value for the vectors (they are not requested to be probability distribution, they could be unit factors). Then, we have two separate updates steps, one for h and another one for a. The rationale behind this update is rather simple: we say that a node is a good hub if it points to good authorities; on the other hand, a node is a good authority if it’s pointed by good hubs. It’s a kind of indirect recursion that reminds us the infinite recursion for the informal definition of PageRank. The update rule for h is the following:

$h = \lambda L a$

where λ is a normalization factor. Similarly, we define the update for a as

$a = \mu L^T h$

where μ is again a normalization factor. Now we simply iterate and repeat these updates until we meet some suitable stopping criterion that could be, for instance, when two successive values of both h and a are close enough.

Let us see why this kind of update actually gives us that informal definition of hubbiness and au thoritativeness. Concerning h: after the update, the generic entry h_i would be

$$h_{i = \lambda} \mathrm{X}\, j$$

Recall that l_{ij} is either 0 or 1, so the sum would be the sum of the a_j for which l_{ij} is not 0. Thus, eq. 2.5 is equal to the sum of the authoritativeness of the successors of the node i. Thus, if the i-th node is a node which is directly connected to other nodes that have a high value of authoritativeness, that would be considered a good hub.
Similar reasoning can be applied to the update of a:

$$a_{i = \mu} \mathrm{X}\, j$$

This means that in eq. 2.6 we are summing the hubbiness of the predecessors. Just one note about those two constants λ and μ. If we didn't put them, each time we perform the update the values of the components will tend to increase, thus risking in an overflow of the result. Therefore, by computing the most suitable values for λ and μ so that a and h are normalized at each iteration, we would avoid this problem.

Chapter 3

Regression

3.1 Linear Regression

We speak of regression in general when we want to predict some quantity which depends on other quantities and ranges within a continuous interval. The simplest way to do regression is to suppose that the quantities at play are related linearly; this brings us to the field of linear regression. First of all, let us formalize how we describe the data that we are going to deal with regression. We typically have a vector of quantities associated to a (scalar) target label; they form a pair like which we will call $(x^{(1)}, y^{(1)})$. We have N of such pairs and they form the *training set*: $\{(x^{(1)}, y^{(1)}), ..., (x^{(n)}, y^{(n)})\}$. However, we will use x_i to refer to the i-th component of the vector x, and $x^{(j)}$ to refer to the j-th vector of observation. The linear regression tries to predict the label by computing a linear function of the components in a vector. So, we say that once we observe a vector x, we compute the inner product between a weight vector and our vector: $w \cdot x$. And to be as general as possible, let us add

here fixed term and get our result, so that the product becomes:

$\hat{y} = w \cdot x + w_0$ (3.1)

$\hat{y}$ is our prediction. Of course linearity here appears because we can expand the product as: $\hat{y} = w \cdot x + w_0$

$$= \sum_{i=1}^{d}$$

Starting from our training set, how do we find a vector w and a fixed term w_0 that will give us good predictions for our labels? We have an observation x that can be expressed as

$x = (x_1, x_2, ..., x_d)^T$

because it is composed of real numbers. Here we place a transposition as well because we usually deal with column vectors. Let us say that actually we will always add a further component so that our vector becomes

$x = (1, x_1, x_2, ..., x_d)^T$

thus having $d + 1$ components. What happens when we compute an inner product of this augmented x and the corresponding weight vector that contains d elements? If we write our weight vector as

$w = (w_0, w_1, ..., w_d)^T$

then as a result of the product we get w_0 and the original vector. This means that whatever we can do in terms of a inner prodcut plus a fixed term (affine transformation), can also be done to a purely

linear transformation. So if we derive some interesting mathematical properties only dealing with this inner product, we can transfer them into the more general affine case. And if we do have some method that only works for purely linear transformation, we can simply adapt it on the more general case of affine transformation explicitly augmenting our data, adding further component somewhere and fixing it to one.

3.2 Non-linear mapping

Despite its natural simplicity, linear regression is always an option to be considered because sometimes simplicity pays out or maybe because we are actually dealing with data that are well modeled using a linear transformation. We can also model non-linear relations through linear regression thanks to data augmentation. Think that we have a rather simple vector

$x = (x_1, x_2)$

in this bi-dimensional space (i.e.: two components). The idea is that of considering a mapping between the original space where our data lives an another space with more dimensions. Let's define the mapping $\varphi : R^2 \rightarrow R^4$ so that

$\varphi(x) = (x^2_1, x_1x_2, x_2x_1, x^2_2)^T$

is a vector that has as many components as the ways that we have to compute the product between the two of the original components. Let's consider another mapping φ^0: $R^2 \rightarrow R^3$such that $\varphi^0(x) = (x^2_1, \sqrt{2}x_1x_2, x^2_2)^T$

Why this mapping is wiser than the previous one? When it comes to computing linear regression in the image of φ, using those two transformation is exactly the same thing. But why would we have to compute a linear transformation in the image of the space? Because a line, or even better, a hyperplane in the image of this transformation maps to a much more complex, and for sure non linear transformation, in the original space of our data. In that case, you see that we will have some polynomial involved, so likely that the hyperplane will map on a polynomial curve on R^2 and this allows us to model some non-linear relation between our values in x. An interesting result comes out when we have two vectors $a, b \in R^2$ and we want to compute the inner product $\varphi^0(a) \cdot \varphi^0(b)$. It becomes:

$$\varphi^0(a) \cdot \varphi^0(b) =$$
□ □

$$= (a_1b_1)^2 + 2(a_1b_1)(a_2b_2) + (a_2b_2)^2$$
$$= (a_1b_1 + a_2b_2)^2$$
$$= (a \cdot b)^2$$
$$= \varphi(a) \cdot \varphi(b)$$

This means that if we wisely choose the mappings that we introduce in order to leverage linear regres sion into a form of non-linear regression, we could be lucky that we don't actually have to compute the mappings because if our algorithms only rely on computing inner products in the image space, we can do that by simply computing the inner product in the original space and then using the result as argument to our particular expression.

3.3 Standard equations

Given a training set $\{(x^{(1)}, y^{(1)}), ..., (x^{(n)}, y^{(n)})\}$ and our prediction $\hat{y}^{(j)} = w \cdot x^{(j)}$, a way that tells us if this specific choice of w is good or bad is to define the *loss function*:

$$\ell(w) = \sum_{j=1}^{n} \quad)^2 = \sum_{j=1}^{n} \quad \cdot y^{(j)})^2$$

Being a sum of non-negative elements, the loss will always be greater or equal than zero and as much all our predictions are close to the corresponding target, as lower (i.e.: close to 0) will be the square of this difference. Ideally, if all the prediction are precise, there's no error and the loss is equal to zero. Most of the time we will not simply randomly pick values for our vector w and check if they are good or bad. We could simply find the argument that minimizes the loss function and so

choose the best w such that:

$$w^* = \text{argmin}_w \cdot y^{(j)})^2$$
$$= \text{argmin}_w \cdot yk^2$$
$$= \text{argmin}_w$$

X is a $n \times d$ matrix which is obtained by stacking all the observations $x^{(1)}, x^{(2)}, ..., x^{(n)}$; that is, each row of X is one of the vectors. Of course, the vectors will be transposed because we are dealing with
column vectors, like

$$X = \quad d$$

When we compute $X \cdot w$ we obtain a vector of n elements where its first component is actually the inner product $w \cdot x^{(j)}$. Thus, the result of $X \cdot w$ will be a vector that has, as first component, the prediction of the first element, and the same holds for the remaining components; therefore, we can write $X \cdot w = \hat{y}$. Now, if we define y as $y = [y^{(j)}]_n$, we can see that once we compute the differ ence $X \cdot w - y$ we obtain the quantities in each component that are squared in the computation of the loss function. And finally, when we compute the squared norm of that difference, we simply ob tain the sum of its squared components. Therefore, we have proved that $(w \cdot x^{(j)} - y^{(j)})^2 = kX \cdot w - yk^2$.

Notice that $\text{argmin}_w kX \cdot w - yk^2$is a convex problem that we can solve by computing the first derivative of the loss with respect to w:

$$\frac{\partial`}{\partial w} = 2X^T(Xw - y) \quad (3.3)$$

and setting it equal to zero:

$$2X^T(Xw - y) = 0$$
$$X^T X \cdot w - X^T y = 0$$
$$X^T X \cdot w = X^T y$$
$$w = (X^T X)^{-1} X^T y$$

X is a $d \times n$ matrix, thus $(X^T)^{-1}$ will be a $n \times d$ matrix; y is a vector of n components. So, we have that w will be a vector of dimension d.

3.4 Avoid overfitting

Anytime we consider an induction process we should be aware of the so called *overfitting*

phenomenon, that is that we end up with a model that has learned not actually the relation among the data used on the induction process itself, but the actual association of any datapoint to the corresponding target. It's somehow similar to saying that we have to output a specific label when we see a specific value. In the end, we will give very precise answers for the N objects in the training set, but as we drift even a little from any of these objects, our prediction will be inaccurate. It could be shown that in the specific case of linear regression, if we consider the vector w^* that is found by our algorithm, it will tend to grow in case of overfitting. We could deal with this problem by trying to perform the optimization process of our loss function not by simply being driven by the need of finding the minimal value of that function but by trying to keep the components of our vector small and we can do that through a fairly simple modification of the minimization problem stated in eq. (3.2). We can simply add a real parameter $\lambda > 0$ in the objective function that tells us if it's more important to find the minimum of this quantity or if it's more important to keep the component of the resulting vector small (i.e.: close to zero):

$$w = \text{argmin}\, w \qquad yk^2 + \lambda kwk^2 \quad (3.5)$$

It's easy to see that the square norm amounts to the sum of the square components of w. So, in order to prevent overfitting, there is some evidence that we have to keep the components of the vector w close to zero. Of course, if we try to keep kwk^2 small, as this quantity is the sum of the square components, the sum will be small if all the components will tend to be small. If λ is very high, it means that a vector w that maybe is good to minimize $kX \cdot w - yk^2$ will be penalized. This happens because the square norm will be more or less high because it would get multiplied by λ, which happens to be very big. So, the overall value for our extended objective function will be high and that particular w will be avoided. On the other hand, if λ is close to zero and we find a vector w whose components are high, the impact of all of those component will be softened by the fact that we are multiplying them by a very small vector, so that they will become small.

For now, let's say that we have chosen our specific vector for λ. Let's see what happens when we want to compute that *argmin*. As we have already stated, we are dealing with a convex problem and thus we can simply compute the gradient and modify it. Let's compute the partial derivative with respect to w:

$$\frac{\partial}{\partial w} = 2X^T(Xw - y) + 2\lambda w \quad (3.6)$$

and then set it equal to zero:

$$2X^T(Xw - y) + 2\lambda w = 0$$
$$X^T Xw - X^T y + \lambda w = 0$$

By exploiting $\lambda w = \lambda I w$, we can rewrite the last equation above as

$$X^T Xw - X^T y + \lambda w = 0$$
$$(X^T X + \lambda I)w = X^T y$$
$$w = (X^T X + \lambda I)^{-1} X^T y$$

Thus, the same procedure that we can use in order to find the optimal vector for standard linear regression can be used, not as is but via simple modification, in order to avoid overfitting.

3.5 Linear regression complexity

Let's see the space and time complexity of the algorithm that computes the best vector w^* for linear regression (note that we are not dealing with the variant that avoids overfitting). The algorithm that performs induction is simply based on the computation of the inverse of a matrix, which is a $d \times d$ matrix, multiplied by a transposed matrix, which is a $d \times n$ matrix, multiplied by a vector of dimension n. This is the equality stated in eq. (3.4).

Let us focus on the time complexity. First of all, we have to deal with a matrix inversion, and let's say that it is $O(d^3)$. Then, we have the product $X^T X$, that requires $O(nd^2)$. We have other computations, like the transpose, but they are dominated. Therefore, the final time complexity is

$$O(d^3 + nd^2)$$

Let's focus on space complexity. We have to store $X^T X$ and then its inverse; in both cases we are speaking about $d \times d$ matrices and thus it is $O(d^2)$ of matrix storage. We also have to store X^T, that is $O(nd)$ since it is a $n \times d$ matrix. Again, the storage of the y vector is dominated. Therefore, the overall space complexity amounts to

$$O(d^2 + nd)$$

What happens when the sizes of our data grows? The size can grow in different ways because here we have to deal with two key quantities: the number n of objects and the dimension d of each object. Of course we will not mention the case when both this quantity are small because we can simply execute the algorithm that describes this operation using standard computing facilities and using standard programming languages. So, we will deal with the situation that arises when either of these two values becomes big. Thus, we have three possibilities:

1. Big d, small n.

2. Big n, small d.

3. Both d and n big.

We will avoid speaking about the first situation because this is a problem well suited for another topic which is called dimensionality reduction; that is, we have not that much data, but each data item is described by a huge vector. So, we want to perform a sort of a compression scheme that takes those vectors an reduces the number of dimensions in them. Thus let us consider the case big n, small d. The first term, both in the time and the space complexity, depends only on d, so that it will not be much of a concern to us because the matrix inversion and its storage, that required $O(d^3)$ and $O(d^2)$ respectively, can be done in main memory if d is small. Therefore, the real problems arise with the computation and storage of X^T. If the matrix has a huge number of rows, the solution to this problem is using distributed storage and computation. This applies also to the case when we have both d and n big.

3.6 Outer products

If we want to distribute both storage and computation when we have to compute the product between two matrices, we can reformulate the overall problem in the form of matrix multiplication. Let us say that we have two matrices

$A = [a_{ik}]_{n \times c}$
$B = [b_{kj}]_{c \times m}$

and let their product be P

$P := A \cdot B =$

Let us introduce the *outer product* operation and, for each k, denote as P_k the result of this operation $P_k = A_k \oplus B^k$

where A_k mean the k-th column of matrix A and B^k means the k-th row of matrix B and the operation $\oplus$ is the outer product. For example, the outer product of $\binom{a}{b}$ and $\binom{c}{d}$ is:

$a\ b \qquad\qquad c\ bd$

Note that we can, in principle, computer outer products using, as operands, two vectors with different dimensions. In our case, $P_k = A_k \oplus B^k$ is interesting because P can be expressed as a sum of the
various outer products P_k:

$^{c}\ k=1$

So, for example, the following

2 3

1 2

Thus, in order to solve the computation and storage problem when n is big, we can rewrite the product
$X^T X$ as 3.8)

$k=1$

The k-th column of X^T is the k-th row of X. A row of X is one of our observation and as the number of component of each observation is small, we can deal with the computation of one addend in this sum in a standard computer. Indeed, the idea is that of distributing the computation of the product by sending the observations to different nodes and letting each node deal with one single

observation at a time. It could be that each node just deals with just one observation and computes these outer product or it could be that it deals with several observation, the ones that are stored there.

At the local level, concerning space complexity, each node receives one observation (so it will have to deal with something like $O(d)$ floats), but it will also have to store the result of these outer product which is a $d \times d$. So concerning this second form of storage, it will need to deal with

$O(d^2)$

complexity for storage. Regarding time complexity: we will have to compute each entry of this square matrix. Therefore, we will have to multiply the number of entries of this matrix (that is d^2) by the time complexity of the computation of each entry, that is constant. Finally, time complexity is

$O(d^2)$

Again, the critical quantity here is not d but n, and it appears only in the sum of eq. (3.8). How do we deal with summing the results that have been computed each at local level? We could for instance, set up a map reduce job where the single node operates on maps that compute these single outer products, and the reduce step is devoted to summing the results. Afterwards, we also need to invert the matrix that results from these sums, but again that inversion operation does not pose problems on the complexity side because it's the inversion of a $d \times d$ matrix where d is small; thus, we are able to do that in a single node.

3.7 Gradient descent

When the dimension of observation is small, we can distribute the computation of all the involved quantities using *map-reduce*-like operation and basing typically on outer products. But what happens when we are in the worst case scenario where they all involved key quantities become unmanageable (locally, that is both the number of observation and the dimension of each observation becomes big)? In this case, we can resort to *distributed optimization*. Since both space and time complexity become unmanageable from a single computer, we need to keep the complexity both for computation and space linear in d and n and possibly exploited some sparsity condition. The easiest thing to do is to exploit some local optimization procedure that is up to be parallelized. Note that here we are dealing with a convex problem, so we don't actually have to worry about the locality of our optimization algorithm. Thus, even a simple gradient rule can do the trick. Think about a simple case where we want to opti mize a function of on a single argument: the idea is that the derivative of the function in any point, always points towards the direction of maximization. Indeed, if we change the sign of the derivative, it will point to the direction to which our function decreases. We can set up an iterative process that con tinuously computes the derivative and subtracts to the current point and updates the current point as the result of this subtraction. It will stop in correspondence of the local minimum, where the derivative is zero. We can choose, typically randomly, the first value for our iteration an continue these updates until we obtain some form of convergence. We know that when it converges, we will be on a local mini mum and in our case this is not a problem because our problem is convex so we just have one minimum.

Let's express this problem formally:

Algorithm 1: Gradient descent (scalar)
choose $x_0 \in \mathbb{R}$
$i = 0$
while *!stop* do
$x_{i+1} = x_i - f'(x_i)$;
$i++$;
Output: x_i

We do have to make some changes to this first simple procedure. First of all, we are not dealing with function that only has a single argument. The function that we want to minimize is our loss function that depends on w: thus, instead of choosing a real value, we will have to choose a real valued vector $w \in \mathbb{R}^d$. The update becomes vectorial and it will return a vector. An important edit is to define how big is the step of the update we are going to take, and we do that through the learning rate $\xi_i > 0$. Notice the subscript: we want to be able to choose a different learning rate at each update. A good strategy is that of starting with big step sizes and reduce the learning rate once we are getting closer to the minimum: thus we can define the learning rate as:

$$\xi_i = \frac{\xi_0}{n^{\frac{1}{i}}} \quad (3.9)$$

Finally, we can rewrite the gradient descent procedure with the updates we have made:

Algorithm 2: Gradient descent (multi-variable)
choose $w \in \mathbb{R}^d$
$i = 0$
while *!stop* do
$w_{i+1} = w_i - \xi_i \nabla \grave{}(w_i)$;
$i++$;
Output: w_i

What happens when we want to apply the gradient descent method to the optimization of the quadratic loss function on the basis of linear regression? In that case the problem we are focusing on is

$$w^* = \text{argmin}_w \cdot y^{(j)})^2$$

where $\sum_{j=1}^{n}(w \cdot x^{(j)} - y^{(j)})^2 = \grave{}(w)$. In order to apply the gradient descent we need to compute all the partial derivatives of this function:

$$\frac{\partial}{\partial w_i} \grave{} = \frac{\partial}{\partial w_i} \cdot y^{(j)})$$

$$= 2\sum_{j=1}^{n} \cdot y^{(j)})(x^{(j)})_i$$

because the component w_i in the addends will be different from the w_i from which we are computing the derivative in all cases but one; that is, the derivative will be zero in all components but one. We need to transform this derivative into vectorial because we want to compute the gradient:

$$\frac{\partial}{\partial w} = 2\sum_{j=1}^{n} \cdot y^{(j)})x^{(j)} \quad (3.11)$$

The update of the gradient descent will be:

$$w_{i+1} = w_i - \xi_i \sum_{j=1}^{n} y^{(j)})x^{(j)} \quad (3.12)$$

Note that in the actual implementations we do not always compute this update because we have some problems with the size of n. To solve this problem, we could focus on a single element of the sum that appears in the update and compute $(w_i \cdot x^{(j)} - y^{(j)})x^{(j)}$ in a node of a network. Note that in order to compute this quantity a node would have to be fed with both the previous value of w_i, one observation $x^{(j)}$ and the corresponding target $y^{(j)}$.

Which is the space and time complexity needed in order to compute $(w_i \cdot x^{(j)} - y^{(j)})x^{(j)}$? We have to store locally the terms in parenthesis, and they take d elements, so that we have linearity. Similarly, computation only needs requires us to compute d products, so we have linearity too. Note also that as d is big, we can further distribute the computation of these inner products using a map-reduce based algorithm because once we have computed separately all the elements, we just need to compute a sum and that can be done through standard reduce operations.

3.8 Logistic Regression

If we consider our data set, which is again expressed as a sequence of pairs $\{(x^{(1)}, y^{(1)}), ..., (x^{(n)}, y^{(n)})\}$ where the first component is vector, but now the second component is a scalar that no more ranges in a continuous set; even further, it can just be obtained by choosing between two different possible values −1 and +1, that is $y^{(j)} \in \{-1, +1\}$. We are actually moving to a binary classification problem because the fact that y is either −1 or +1 can be used in order to denote that the corresponding observation x belongs or not to a specified class. Think that the vector x will encode a linearized picture and that the two classes are related to the presence or absence in that picture of something that interests us: pictures that contain a car will be labeled +1 otherwise −1. Is there something similar to regression that allows to find a vector w that, properly mixed in a linear way with our observation, gives a good

approximation of the labels? The fact that we have used +1 and −1 in order to denote the membership to classes can be exploited in building a suitable loss function because we will use those two values also for predictions; then, it is rather easy to find out if a given prediction is correct or not. Here we are no more in the case of regression, so our prediction could be close or far away from the corresponding labels. This happens because either the prediction is equal to the

class or not; so, there is a lot of space in order to be wrong; indeed, you can weight differently predicting as positive labels that were negative from the opposite situation, and viceversa.
So, once we have found a vector w, we could for instance compute the inner product between it and an observation:

$\hat{y} = \text{sign}(w \cdot x)$ (3.13)

This is a real number and we can consider the sign of that number as our prediction. When we compute this product, we need to obtain +1 if our prediction is correct. When we have any of the two possible errors, either we have true label −1 and prediction +1 or viceversa, the product is −1. Roughly speaking, this amounts to considering a hyperplane in the space of our observations and to consider the two out-spaces that are found considering the two portions of the space that are separated by this hyperplane; we are basically saying that everything that falls on the top of that hyperplane will be classified as positive (+1) and everything that falls on the bottom will be classified as negative (−1). We might have to decide what happens on the point that actually lie on that hyperplane: we could decide to associate them either to one class or the other. As we know, we can see if a prediction is correct by multiplying it with the corresponding label, so we could use a loss that typically depends on the product between the label and the corresponding prediction:

$\grave{}(y, \hat{y}) = \grave{}_{0/1}(y\hat{y})$ (3.14)

Let us introduce the so called *zero-one loss*: it is one if we are making prediction mistakes, zero otherwise:

$\grave{}_{0/1}(z) =$ ( ) 15)

We can see that this catches the intuition behind a loss function because we want the loss function value to be high when we are not behaving well, and we want it to be low when we are behaving well (that is when we are making correct predictions). To actually write the global loss, we would have to sum all this single losses for all the observations we have. So, ideally, we would like to minimize over w the sum on all observations of the loss having as argument the product between the j-th label and
j-th prediction:

$$\min \qquad \cdot x^{(j)}) \quad (3.16)$$

The overall product in parenthesis will be only tested against its sign. So, if the prediction is negative, also $w \cdot x^{(j)}$is negative.
We said that ideally we want to minimize this function, but the problem here is that the zero-one loss is not convex. If we try to graph it, the function value for negative arguments is 1, otherwise it is 0. We want to find something that suitably approximates this loss but that does not lack the convexity requirement. We can do that by considering the so called *log loss* which is defined by:

$\grave{}_{\log}(z) = \log(1 + e^{-z})$ (3.17)

This is an “usable” approximation for the zero-one loss because it is kept small (that is, close to 0) for positive values of its argument, and it becomes big for negative values of the arguments. So, we can simply switch the loss in the minimization problem:

$\cdot x^{(j)}$) (3.18)

$w^* = \text{argmin}\ w$

Of course we will obtain something which is somehow approximated, but now we can mathematically deal much more easily with this new form of our optimization problem.

In order to apply the gradient descent algorithm to the variant of our loss function based on the log loss, we have to compute the derivatives. The loss function $\grave{}(w)$ has the form

$\cdot x^{(j)}$) (3.19)

$\grave{}(w) = \sum_{j=1}^{n}$

where we could have summed another term $\lambda \|w\|^2$ in order to avoid overfitting; however, we will not consider it when computing derivatives. Let's compute the first derivative of the log loss with respect to its scalar argument z:

$$\frac{d\grave{}_{\log}(z)}{dz} = \frac{d}{dz}\log(a + e^{-z}) = 1$$

$$1 + e^{-z}(-e^{-z}) = -1 + e^{-z} - 1 \quad (3.20)$$

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

Then we compute the derivative of the loss with respect to w: $\frac{\partial}{\partial w} = \sum_{j=1}^{n}$

$$\frac{\partial \grave{}}{} \quad {}^{(j)} \quad w \cdot x^{(j)}$$

$$= \sum_{j=1}^{n} w \cdot x^{(j)}(-y^{(j)}x^{(j)})$$

Now, using gradient descent, we are able to find the best value for w that minimizes the loss.

We don't want to give a definitive answer to the question "is this observation belonging to that class?". This means that we don't want our answer to be +1 or −1 but we want to somehow being able to graduate between belongingness and exclusion to that class. We could, for instance, resort to a sort of probabilistic output and using as prediction the probability that the observation belongs to that class, given that the random vector of our observation is the vector we have actually observed. As output of this probability, we could rely on the inner product $w \cdot x$ and use this as

argument to a function σ whose value might be interpreted as probability values:

$P(Y = 1|X = x) = \sigma(w \cdot x)$

A suitable choice for σ is the *sigmoid* function:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \quad (3.22)$$

Thus, we will proceed in the following way:

1. Logistic regression $\rightarrow w$

2. $P(Y = 1|X = x) = \sigma(w \cdot x)$

3. Fix a threshold $0 \leq \tau \leq 1$. If $P(Y = 1|X = x) > \tau \rightarrow +1$; otherwise -1.

Figure 3.1: Sigmoid function

One of the ways to choose τ is to use the ROC curve. We evaluate $P(Y = 1|X = x) > \tau$ on the test data in order to get the true positives, true negatives, false positives and false negatives so that we end up with a *confusion matrix* :
T P F P
F N T N

We can then compute some indices and get the ROC curve[1]:

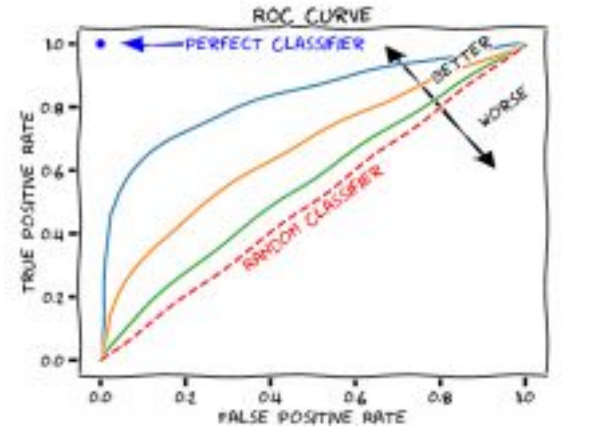


Figure 3.2: ROC curve

On the y-axis we have $\frac{TP}{\#\ positives}$ while on the x-axis we have $1 - \frac{TN}{\#\ negatives}$. We do have several special cases. For instance, the first one happens when we find out to be in the upper right corner of the square. In that case we are using a sort of dumb rule which continuously gives the same answer (i.e.: yes, +1) regardless of the vector that it considers. At the opposite corner, the situation is almost analogous but the classifier always outputs −1. If we are on the top left corner, the classifier makes zero mistakes while on the bottom right corner the classifier misclassifies every observation. If we are on the diagonal line, we have a classifier that acts like a coin flip. Thus, we want to be above (i.e.: on the left side) of the diagonal. Depending on the choice of τ we will end up in a different place in that square. So, we can check up different possible values for that threshold and choose the one that is nearest to this ideal case. We can switch from a qualitative analysis of this curve to a quantitative one by measuring the area between the ROC curve and the *x*-axis, which is called AUC (Area Under the ROC Curve) and the more it is close to 1, the closer we are to the best possible situation.

[1]Source: https://en.wikipedia.org/wiki/Receiver operating characteristic

Chapter 4

Clustering

The term clustering refers to a specific machine learning subfield which is known as unsupervised learning. We speak of learning because we're trying to figure out something just starting from data, and we say that is it is unsupervised because we don't have some kind of labelled data. In that case we just have data and we want to figure out if they organize themselves in a interesting fashion. For instance, if you look at Figure 4.1[1], you see that we have points which we have colored differently. But even if we could not have colored those points differently, just by looking at

the picture, anyone would see that those points are grouped in tree *clusters* because we can easily see that we do have some points that tend to be close one another.

Figure 4.1: Example of clusters

The mathematical concept that allows us to easily define if things are close or far is that of distance. There are several ways that allow us to define a function which is a distance within a space. For now, let's keep it simple and just think about the length of the segment that joins two points. Informally speaking, will say that two points are close if their distance tends to be low and they are far if their distance tends to be high. So, if we take any two points here in the upper part, we see that the segment that joins them tends to be small, that is, it has a small length. Here is intended as such because if we compare its length with the length on any other segment joining points of different color, the second

[1]Source: http://scikit-learn.sourceforge.net/0.6/auto examples/cluster/plot mean shift.html 36

segment would have a far bigger length. In general, we will say that having fixed a distance measure on a space and a set of points in the space, the clustering process has the goal of finding out two or more groupings of these points and we will call each of these grouping a "cluster" where a cluster is somehow defined by the fact that all points belonging to it tend to be close. Here we have to make an important difference between the concept of Euclidean and non-euclidean spaces. The reference space is the classical space that somehow refers to R^2, which is the prototypical euclidean space, where in general an Euclidean space has a very nice property that can be summarized as follows: if we consider any two points in that space, then the segment that joins those two points is itself composed by an infinity of points and each such point belongs to the space. Of course, the density property of the real number allows us to say that $R^2$is an Euclidean space, and for instance if we consider those two points, the middle point will be a point in R^2. This property obviously extends to the very center of a set of points. So that, if you're working in a Euclidean space and we are specifically dealing with clustering, we can select any cluster by computing its center of gravity (barycenter) and that will be an object of our space. Thus, we could summarize the property of belonging to that cluster by virtually collapsing all those points on their

barycenter. Those specific barycenters are called *centroids*.

Note that we can have data that live in a non-Euclidean space. Think about strings of a given length: considering two strings of fixed length and figure out their midpoint may not lead to something with an actual meaning. We are no more able to compute the barycenter of the data and thus we have to find a new entity that becomes a sort of object representing the whole cluster, and that has to be necessarily one of the observations: in this case, we speak about *centroids*.

One of the biggest problems is the fact that in Figure 4.1 it's really easy to figure out how many clusters there are and which point belongs to which cluster because we can easily visualize the points. This is not possible when we have points that live in a space of higher dimension; in that case we have to resort to some automatic procedure and the fact that the dimensionality of points raises up can pose interesting problems. We also have to point out that when we deal with a massive amount of data, we cannot think of loading all the data and main memory and perform some automatic procedures. So, we will have either to resort to some distributed storage and computation technique, or figure out some clever way to store data in main memory. For instance, instead of loading all points we could think about compressed representation of the points within a cluster.

4.1 Curse of dimensionality

The clustering process is heavily based on pre-fixed distance factor and when we deal with objects that live in a space with several dimensions, it could turn out that distances we are used to employ might become sort of meaningless. This problem is known as the *curse of dimensionality*. Consider, for instance, the classical Euclidean distance for space with n dimensions:

$$d(x, y) = \qquad (4.1)$$

If we choose uniformly at random in $[0, 1]^n$ two points x and y when n is big, it will be very likely that $|x_j - y_j| \approx 1$ for at least one dimension. But if this absolute value is approximately equal to one, also its squared value is close to one so that when we compute the distance between x and y it will likely be
$d(x, y) \geq 1$. On the other hand, we know that $|x_i - y_i| \leq 1$ always, for any n, and thus $d(x, y) \leq \sqrt{n}$. Now, let's choose h points (no more two), compute the pairwise distances and perform some simple descriptive statistical analysis (for instance, computing an histogram) and check up what happens at the shape of that histogram. When we continuously repeat this experiment changing the value of n, we will see that the histogram becomes more peaky, which means smaller variance as n increases. This means that when we deal with points of big dimensions, the Euclidean distance tends to assume always

the same value and this is actually a "curse" because if all points tend to have the same distance, we will not be able to use that distance in order to perform clustering; we would end up with just one cluster, which is completely uninformative. This is not actually a flaw in the Euclidean distance because the curse of dimensionality tends to be independent from the chosen distance. Take for

example the cosine distance:

$$\arccos \frac{x \cdot y}{\|x\| \cdot \|y\|} = \arccos \frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{\sum_{i=1}^{n} x_i^2} \sqrt{\sum_{i=1}^{n} y_i^2}}$$

it is affected by the curse of dimensionality too. Think that you have a reference space and think to just throw three points *A*, *B*, *C* at random uniformly in your space, and we see that those points allows to form 2 vectors. When we try to compute the cosine distance when *n* is big, it turns out that the numerator is a product between components that are sometimes positive and sometime negative. Since we are throwing points uniformly at random, we will have approximately the same amount of positive and negative components and thus, the numerator is going to become the sum of uniformly drawn numbers in an interval that is centered around zero. When you sum up several uniform values you obtain something which is normally distributed (in general when you sum up anything that is strongly independent from the same distribution you end up, by the central limit theorem, with something which is approximately distributed as a Gaussian); the point here is that it is centered around zero. So, as *n* becomes big, the numerator becomes something that is more and more similar to the specification of a random variable which is normal and centered around zero. The fact that the random variable is centered around zero tells us that the numerator will not be that close to zero when *n* becomes high. Regarding the denominator, when *n* increases, both sums need to increase (since they are sum of non negative terms) and the whole denominator will grow indefinitely. Finally, as *n* increases, regardless of the points that we are considering, the distance will be always approximately zero, so again we have the same distance regardless of the points we are considering. This curse of dimensionality is something that tends to happen when you deal with a high number of dimensions, regardless of the distance that you tend to use.

4.2 Hierarchical clustering

We speak of *hierarchical clustering* when we build up clusters by subsequently merging clusters ob tained via an initial assignment of points to singleton clusters. Consider, for instance, a case where we have nine points on the bidimensional space and they've been labeled. There is a sort of initialization phase where we build one cluster for each point, so we will have as many clusters as points that we are considering; each of these clusters will contain exactly one point. Then, we have a two-step iterative part: it consists in selecting two of the available clusters and merge them together until a fixed stopping criterion (there are several ones which from which we can choose). Selecting clusters means select the two clusters that are more apt to merge together in order to former acceptable cluster. This means, for example, to choose clusters basing on their distance. If we are within our euclidean space, we know that each cluster is described by a centroid and the centroid is itself a point of the space we are dealing with. So, we can use the same distance between points also as a distance between clusters; precisely, the distance between two clusters will be the distance between their centroids. When we merge two clusters, we can then compute the centroid of the new resulting cluster. The representation of the merging steps is a tree that we call *dendrogram*: it tells us how the single elements have been merged together up to joining all the points in a single cluster if we read it from bottom to the top; or we could reverse the direction

and reading from the top to the bottom in order to say how the global set of points starts to differentiates in different group up to becoming nine (in our case) different specialized objects. We could repeat this process until we have gathered all points in the same cluster, but that will not be informative.

38

Let us make some time complexity considerations. Let us say that we have m points. In order to form the singleton clusters, we will need $O(m)$ time. At the first iteration, we will need to consider the distances between all the m points: thus, we will need $O(m \cdot (m - 1))$ (but this is equivalent to $O(m^2)$). At the second iteration, we will need $O((m - 1)^2)$. The first O is for sure dominated by the subsequent ones: so, in order to know the overall time complexity, we will need to compute $\sum_{i=1}^{m} i^2$ where i is the number of iterations. This brings us to a quantity that is in the order of magnitude of $O(m^3)$. Actually, we can slightly lower this time complexity if we organize data in a smarter way. That is, if we compute (just at the beginning) all the distances between possible pairs and organize them in a priority queue which is properly managed during the overall process by pruning and inserting new elements, we can lower the time complexity to:

$O(m^2 \log m)$

There are several customization that can be applied. For instance, we could choose the cluster to be merged also using different criteria which are not necessarily based on the distance between centroids. Note, however, that if we want to compute a rule that allows us to choose clusters, the one based on the minimization of cluster centroids distance requires us to be within a Euclidean space, otherwise we would not have centroids. However, we could also rely on another kind of distance that does not require the existence of centroids at all. Think for instance that we have two distinct clusters: we could compute the pairwise distance of the points from the two clusters and then say that the distance between the two clusters is the minimum distance computed among their points. This does not require us to be in an Euclidean space. Or, we could also rely on the average distance among points in two clusters; in both cases however, we want to minimize this distance in order to merge the closest clusters. Another different criterion is to rely on the clusters that we would obtain after the merge rather than relying on the existing clusters. For instance, we could the define the radius of the result as the maximal distance between the centroid and all the points. As this relies on the concept of centroid, we need to be within an Euclidean space. Using this criterion, the idea is pretend to merge all possible clusters and choose and merge the two that satisfy our criterion. We can perform a similar reasoning using the diameter instead of the radius, where the diameter is the maximal distance among the points in the (new) cluster. In this case we don't need to have at hand a centroid, so we are not required to be in an Euclidean space. This kind of radius has been introduced because it gives us the possibility of smartly defining when to stop the iteration procedure that continuosly merges pairs of clusters. We have seen that if we iterate indefinitely that procedure, we end up eventually to have all the points in the same cluster and this will be very likely meaningless. So, how do we choose how to stop? We could stop up after, for example, k steps. So if we know how many clusters we want to end up with, we can derive the number of steps k we need to take by looking at the dendrogram. Usually, that k refers to the number of clusters we want to end up with instead of referring to the number of iterations. Another stopping criterion is that we can iterate until the radius of the newest cluster becomes higher than a specified threshold. Another similar stopping criterion is the one that considers the

concept of density instead of the radius. In analogy with physics, the density of a cluster could be thought as something that computes the ratio between the number of points and the volume (or area in the bidimensional case) of the cluster.

4.3 Clustroids

When we work inside an non-Euclidean space we don't have the possibility of running the classical clustering algorithms and obtain centroid for the found clusters. Moreover, if the algorithm needs to compute the centroid during its execution, this is no more possible just because we have no possibility of computing the very center of points within a cluster. The only possibility is to choose among the points that we know have been assigned to a cluster and are particularly representative: they are called clustroids. There are several rules that allow us to select a specific point within all the points in a

cluster in order to act as clustroid. For instance, we could consider the point that minimizes the sum of its distances with reference to the remaining points of the clusters. That is, for a cluster G we will choose the clustroid c^* that satisfies:

$c^* =$ argmin

Minimizing the sum is equivalent to minimizing the average, so we are choosing the point that has the minimal average distance with all the other points. Or we could use a very similar rule but by considering the squared distance:

$d(c, x)^2$

Or, again, we could consider a clustroid that satifies

$$c^* = \text{argmax}\ c \quad \text{max}$$

that is, we have selected the point in the cluster that maximizes its maximal distance with reference to the remaining point, which is reminiscent of the radius of the cluster itself. Once we have selected a clustroid for each cluster, we can substitute it to the centroid in order, for instance, to compute the distance between clusters. That is, the distance between clusters now will be the distance between the corresponding clustroid, which being points can be fed into the distance we have selected before executing the clustering process. Or, we could, for instance, compute the density of the cluster or the radius of the cluster and via this quantity we are able both to select the clusters to be merged in a hierarchical clustering approach and to build up a stopping criterion.

Consider this matrix that reports the *edit distances* between those strings:

	ecdab	abecb	aecdb
abcd	5	3	3
aecdb	2	2	

abecb 4

Given two strings *aecdb* and *ecdab*, the *edit distance* computes the minimal number of atomic oper ations that we have to perform in order to obtain the second string starting from the first one. In this case, the edit distance is two because from *ecdab* we need to delete *a* and then insert *a* at the beginning. We can choose the clustroid using one of the three criteria above and you can see that the choice of the clustroid changes:

Sum Max. Sum sq.
abcd 11 5 43
aecdb 7 3 17
abecb 9 4 29
ecdab 11 5 45

4.4 K-means

Besides the hierarchical approach there is another big family of clustering algorithms that is known as the family of algorithms that assign points to clusters or *point assignment algorithms*. This kind of algorithm simply determines an initial set of clusters and then consider repeatedly all the available points and assign each of these points to the clusters, possibly modifying the structure of the clusters themselves. The most known appoint assignment clustering algorithm is called *k-means* and in order to work, it needs to know the number of clusters we want to find.

Algorithm 3: k-means
Fix number of clusters $k \in \mathbb{N}$
Build clusters on k points
$\forall$ remaining points:

Assign the point to the nearest cluster

Recompute centroid

Reconsider point and reassing to clusters

Before all, we fix our natural number k, which is the number of clusters. It is typically an argument of the procedure that computes the clustering. The other argument which we need to specify is the set of points which we want to cluster. The first step is that of considering k among the points which we want to cluster and build clusters on them. We will come later on some techniques that allow us to choose suitably these k points (we can consider the first k points and we build clusters on them). This means that those points will become the only points in the cluster and also its centroid because k-means is a clustering algorithm that works within euclidean spaces. The second phase simply considers all the remaining points; thus, for each remaining point we will assign that point to the nearest cluster. Afterwards, as we have modified the composition of that cluster, we will recompute it's centroid and we do that until we have finished all remaining points

the dataset. Possibly, there is a final step that now extracts all points from clusters, keeps the clusters with their own centroid and reconsiders each point assigning it to the nearest cluster. Most of the time the original assignment of points to cluster will not be changed, but some of them could actually change.

There are several rules that allows us to choose the initial points. We could, for instance, just rely on chance and say uniformly choose *k* of the available points and start with them. That could be good, or that could be bad. Actually, we would have to select points that are not close together; so we could for instance pre-cluster our data by using a hierarchical approach and stopping, on the dendrogram, when we obtain exactly *k* clusters and so choose one point at random from each of these *k* clusters. Or we could apply another rule that selects our initial points by maximizing their minimal distance with reference to the points that we have already chosen (the first point will still be chosen at random). We can formalize this last approach:

$\min_c d(x, c)$ (4.3) nax *x*

where *c* is an already chosen point.

Another important point is how to know which is the best number of clusters, since we need to fix *k* beforehand. The typical solution is to run k-means several times for different values of *k*, each time measuring the performance of the obtained cluster by, for instance, computing the average radius of the cluster. We know that if we have less clusters than we would need, the average radius would become suddenly high. So, we can start with a high value of *k*, run k-means for that value of *k* and it will happen that if this *k* is bigger than the "true" number, we will subcluster. Once we have reached a good value for *k* and we run again k-means with a lower *k*, then if the radius will increase, that will signal that the optimal *k* was the one chosen previously. However, it may happen that even if we subcluster, we may end up with some meaningful clusters, even though they may not be in the optimal number. Most of the time we won't do that for all possible integer values of *k*, ranging from 2 up to the number of points; we could consider some logarithmic scale and when we find a good value within the logarithmic scale. We can perform some smart operation like some binary search in order to find the best value of *k* without wasting too many computation time.

4.5 BFR

How can standard clustering algorithms be adapted for a situation in which we have a massive amount of data? Of course, we cannot think to be able to run in memory algorithms such as k-means; therefore, we need to update them in order to deal with data that are distributed across a distributed file system. A simple extension of k-means is known as the *BFR* algorithm where the acronym stands for the names of the researchers who proposed it (Bradley, Fayyad and Reina). It works in a sort of analogy with k-means: it works in a Euclidean space and it needs to know before hand the number of clusters, and indeed, the first thing which is done is to provide the number of clusters.

Algorithm 4: BFR

Fix $k \in \mathbb{N}$ and compute k centroids

Read a data chunk

Process it in RAM:

Update clusters

Update mini-clusters

Update unassigned

Check if mini-clusters → cluster

Write report in disk

Reconsider point and reassing to clusters

The algorithm computes k initially empty clusters by generating their centroids. These centroids are obtained through any of the standard techniques that allow us to find k points that are likely to be far away one another. Afterwards, the algorithm processes data in chunks. That is, if data are stored in HDFS, the data file is read chunk by chunk and processed in this way, which means that we will be able to process all the data read each time in main memory. If data is not in a distributed file system, the algorithm goes through passes that at each time only consider small enough amount of data that can be processed in main memory. So, we could say that we read another chunk and we process it in main memory. In what does this process phase consists? First of all, we do have some initial choice for our clusters, and these clusters are updated according to the read data. Therefore, the first thing is to update clusters. Actually, it could be that one or more data items are not judged good enough to fit with the actually known clusters, and so such data items are not discarded, they are just considered apart for a while and in particular we will have two possible situation: a situation where we have a sort of singleton data which is far from any other data, and it's not close enough to an already discovered cluster; and another situation where we have two or more data which are close enough but there are not enough points to build a cluster. We will call this second set of data which is not sufficiently big in order to be deemed as a cluster or as a *mini cluster*. Mini-clusters will be updated too. Data which is not assigned either to clusters or to mini-clusters will be called *unassigned data*. We will then check up if mini-clusters and unassigned data can be "promoted", then we will have to write a sort of report on disk because we will have to note somewhere if our points have been assigned to clusters. Of course, we cannot think to keep in main memory the points as the various chunks have been read because after a while we will fill up the available main memory. So, the solution is that of storing in main memory only the unassigned points and exploiting some structure in clusters and mini-clusters in order to just store a small amount of information, which indeed allows us to perform all the rest of the clustering algorithm. In particular, both clusters and mini clusters are associated to two particular subsets of data which are called "discard set" and "compressed set" respectively. Analogously, the unassigned points are said to belong to the "retained set". Here retained actually means that we are storing those points with all their coordinates in main memory, discard and compressed sets are dealt

with in a different way. All points belonging to discard set or to the compressed set are not stored in main memory; we will instead keep a record of them at level of single cluster. We will store three quantities:

• *N*, which is the number of points in a (mini) cluster;

• *SUM*, which is the sum of all points in the (mini) cluster (we can sum them since each point is a vector);

• *SUMSQ*, is the sum of each vector that has been squared component-wise.

We are not keeping which point belongs to which cluster, but this is something that we write in the report after having processed a chunk. These three quantities allow us to properly assign new points to (mini) clusters and to update the centroids because:

$\frac{1}{N} \cdot SUM \rightarrow$ centroid

Moreover, we will need also to measure of the dispersion of points in a cluster around the centroid, and we can measure that as the standard deviation. The standard deviation is obtained as a square root of the variance, and we can approximate the variance as

$$\frac{1}{N} \cdot SUMSQ -$$

because $\frac{1}{N} \cdot SUM^2$ is somehow reminiscent of the expected value; $\frac{1}{N} \cdot SUMSQ$ is reminiscent of the second moment, and thus the formula (4.4) is something similar to the definition variance. Once we know the variance, we can compute the square root of (4.4) and get the standard deviation. When we process each point in a chunk we will have to decide whether to add any of each point to the existing cluster, mini-cluster or to the retained set. In order to do that, BFR poses the hypothesis by asking that all points are distributed within one cluster according to our multivariate normal distribution and it requires that there is independence among the components of that distribution (this means having counter levels for the density orthogonal to the axis of the space). Once we have our clusters, their standard deviation and their centroid, we are able to compute the density in order to check if a new point is likely to belong to one of the clusters by simply computing the probability density function for all clusters. Then, we would add the point to the cluster for which this density is the highest. However, it could be the fact that our point is actually far by from other centroids so that even the value of the probability density function is actually really low. We need a quick means that allows us to compute the value of the probability density function for each cluster. Actually, we don't need to compute the true density value because we just want to know for which cluster the density is higher. So, we can simply compute a particular distance between a cluster with centroid c and standard deviations $\sigma_1, ..., \sigma_d$ and a point p by computing

$$d(p, c) =$$

and we want this quantity to be small because in this way the probability density function value is higher. This distance is called the *Mahalanobis distance*. Therefore, we compute the Mahalanobis distance for the point with respect to each cluster, and if the minimal distance is small enough, we will assign it to that cluster; otherwise, it will go in the retained set. In case we assign it to a cluster, we need to update the three quantities. The update is rather easy:

- $N = N + 1$
- $SUM = SUM + p$

- $SUMSQ = SUMSQ + p^2$

Note that this kind of representation is clever enough to require just a few multiplications (the ones in the squaring of *p*). Sums are better than multiplications for two main reasons: the first one is that computing a sum requires less time than computing a multiplication or division; the second reason is that if we have some noise in our data, it propagates less when we perform sums or subtractions, and it tends to propagate much more when we perform more complex operations such as multiplications and divisions. Now, once we have processed all the data in a chunk, some of them will be possibly inserted into existing clusters and the other ones will be added to the retain set. Before processing the next chunk of data, all the mini-clusters and the retained set are clustered. The points in the retain set are clustered just to check if we can form some new mini clusters and we will check if this mini-clusters are close enough to already existing mini-clusters so that by merging them together we'd obtain a new cluster. Again, we will drop the points, we will update the representation of these mini-clusters and we will keep in main memory only the points that are not assigned either to new clusters that are obtained by this merging process or to mini-clusters that are kept for the next iteration of the algorithm. The remaining points will form the new retain set and they're kept in main memory. Once we finish all the data, we will need to decide what to do with the mini-clusters and the unassigned points remaining after the last iteration. Here we have two choices: we could either consider them as outliers and thus just drop them; or, we could anyway assign them to the closest clusters, even if they are not close enough with reference to the threshold that tells me if a point is close enough to a cluster in order to be added to it.

4.6 GRGPF

The GRGPF algorithm is a clustering algorithm that deals with big amount of data and removes the requirement of working within a Euclidean space. The name is after the initial of the researchers that proposed this algorithm. For each cluster, we store:

- *N*, which is the number of points
- Clustroid $p^* = \text{argmin}_p ROW\ SUM(p) = \text{argmin}_p \sum_{x \in C} d(p, x)^2$
- $ROW\ SUM(p^*)$
- *k* closest points to p^* + *ROW SUM*
- *k* farthest points from p^* + *ROW SUM*

These information are stored into a tree-like structure. We extract a sample that can be stored in main memory and then we apply the hierarchical clustering algorithm in order to obtain the full dendogram. The idea is that we keep the structure of the dendrogram up to a certain level, and that level is fixed before running the algorithm. What remains is a set of clusters and for each of these cluster we will compute the clustroid and we will place the representations of the corresponding cluster in the leaves of the tree. We will properly consider the last levels of the dendrogram in such a way that a leaf contains the representation of possibly several clusters. Non terminal nodes will contain some sampling of the clusteroid which are represented in the subtree starting from that node with pointers to the proper subtrees. Once we have built this tree, we are ready to reconsider all points. For each point *p*:

• Follow the tree: check which of the clustroids that occurs on the root is closest to *p*. That clustroid will point to the subtree that contains that clustroid closest to *p*. So we will repeat this process on the selected subtree and continue up to reaching leaf. When we get to a leaf, we will be able to find the closest cluster *p*. And then,

• Assign *p* to cluster. This will require to update the cluster representation. 44

Update representation of cluster with new point *p*:

• $N = N + 1$
• Clustroid $p^* = \text{argmin}_p ROW\ SUM(p) = \text{argmin}_p \sum_{x \in C} d(p, x)^2$
• $\forall q$ points in repr. $ROW\ SUM(q) += d(q, p)^2$

We need to check if the clustroid is still the same after having added *p* and we need to see if the new point should be retained in the representation because it either becomes the clustroid or goes into the *k* closest or farthest points. We cannot compute the *ROW SUM* of *p* because we would need to load in memory all the points in cluster *C*: thus, we need an approximation for that.
Working in a non-Euclidean space is somehow like working in a high dimensional space. In a high dimensional space we know that the cosine distance of two points taken at random is close to zero, which means that if we consider two vectors at random in the space, we see that they are approximately normal. Note that that if two vectors are normal, we can apply the equivalent of the Pythagorean Theorem so that we could say that

$$d(p, q)^2 \approx d(p, p^*)^2 + d(p^*, q)^2 \quad (4.6)$$

We can easily get from this approximation to an approximation of *ROW SUM*(*p*). In fact, when we sum both cases on *q* we get

$$\sum_q \qquad \sum_q \qquad (4.7)$$
$$\sum_q$$

where $\sum_q d(p, q)^2$ is the *ROW SUM*(*p*), $\sum_q d(p, p^*)^2$ is equal to $N \cdot d(p, p^*)^2$ because it is the distance $d(p, p^*)^2$ summed *q* times (since *q* does not appear in the sum), and finally $\sum_q d(p^*, q)^2$ is the sum of the clustroids, thus $ROW\ SUM(p^*)$. Therefore, we can rewrite (4.7) as

$$ROW\ SUM(p) \approx N \cdot d(p, p^*)^2 + ROW\ SUM(p^*) \quad (4.8)$$

Check if *p* replaces farthest or nearest points. If *p* does not replace one of these points, we will simply drop it. Otherwise, we have to store it alongside with its own *ROW SUM* and it might be the case that the clustroid is changed. It's easy for us to check this thing because we know that the clustroid minimizes *ROW SUM*, so we simply have to check if the *ROW SUM* of p^* has become lower than the *ROW SUM* of the clustroid. In that case, check if *p* replaces the clustroid. The clustroid would then become one of the nearest points and that's it because once we've done that, we're able to process all points. Note, however, that we could have incurred in two problems: the first problem happens when the clusters become too big. We will have a threshold to be computed against the radius of each cluster so that each time that we add a new point to a cluster, we will check if the radius exceeds that threshold. If this is the case, then the cluster is too big and we need to split it into 2 sub-clusters. In order to do that, we need to transport into main memory all the points of that cluster and then we can apply any clustering algorithm that runs in main memory in order to find two clusters from the previous cluster. Note that this will need an update of the tree structure that collects the centroids and if there is no more room in a leaf, then we will need to split the leaf in two. We should always take in mind that, for efficiency reasons, the tree that gathers all these structures that represented clusters is not a common tree but a special balanced tree. However, here we could incur in another problem: that tree need to be always in main memory, so if we split too much, we can exhaust it. This means that we have to reconsider the whole clustering and merge together some clusters; possibly, clusters whose representation are contained in a same leaf of sub-tree. This is done via raising of the threshold that tells us when a cluster is too big. Then, we will have to consider all pairs of clusters and try to merge them. Now, merging them is rather easy: we will simply have to check the distance between the clustroids. The problem here is what to do with the representation of the two clusters to be merged in

order to get the representation of the merged cluster. The representations are really easy to update: for instance, we know that in each representation we have stored the number of points in the two single clusters. Of course, summing those two numbers gets us the number of points in the merged cluster, but in other cases it's rather difficult to update some of the remaining parts of the structure and this is where the fact of having stored farthest point comes into play, because in order to do that we will use the farthest point and again the approximation on the computation of distances.

Deep learning and TensorFlow $\overset{\text{Logit function:}}{\overset{X}{}} w_i x_i \rightarrow y = f(\text{logit})$ (5.1)

Loss function: $w_{n)} = {}^{1}{}_{2}X_{j}(t^{(j)} - y^{(j)})^{2}$(5.2)

We have to minimize the loss, thus

we compute: ∂ logit$^{(j)}$

$\partial w_{k=}\partial$

$_{k}$when $i = k$

and

$\partial y^{(j)}$
$\partial w_{k}= f^{0}(\text{logit}^{(j)}{}_{)}\partial$

$\partial w_{k}\text{logit}^{(j)}$
$= f^{0}(\text{logit}^{(j)}) \cdot x^{(j)}$

we can combine eq. 5.3 and eq. 5.4 and derive w.r.t. w_{k}: $\partial w_{k= -}{}^{1}{}_{2}X_{j}2(t^{(j)} - y^{(j)}{}_{)}\partial$

k

$f^{0}(\text{logit}^{(j)})x^{(j)}$ k

∂E

$$f^{0}(\text{logit}^{(j)})x^{(j)}(5.6)$$

In the multivariate case:

where $(y^{(j)} - t^{(j)})$ is the error of approximating the j-th target with the j-th output. This is a “plain difference”, and it’s interesting because if the output is higher than the target, we perform the update in one direction; otherwise, we update in the opposite direction.

5.1 Back-propagation

• ADD GATE:

– Forward: $s = x + y$

– Backprop: both x and $y \leftarrow {}^{\partial f}$

$_{\partial s} \leftarrow + \leftarrow {}^{\partial f}$
∂s

• MULT. GATE:

– Forward: $p = x \cdot y$

– Backprop:

$* x \leftarrow y^{\partial f}$

$\frac{\partial p}{\partial p} \leftarrow \cdot \leftarrow \frac{\partial f}{\partial p}$

$* \; y \leftarrow x \frac{\partial f}{\partial p} \leftarrow \cdot \leftarrow \frac{\partial f}{\partial p}$

• MAX GATE:

– Forward: $m = \max\{x, y\}$

– Backprop: if $x \geq y$

$* \; x \leftarrow 1 \cdot \frac{\partial f}{\partial m} \leftarrow \max \leftarrow \frac{\partial f}{\partial p}$

$* \; y \leftarrow 0 \cdot \frac{\partial f}{\partial m} \leftarrow \max \leftarrow \frac{\partial f}{\partial m}$

Invert the results for x and y for the case $x < y$ 48