Applied Big data

Michel de Rougemont

September 13, 2021

Contents

1 Introduction **7**
1.1 Mégadonnées . . . 8

2 Preliminaries **11**
2.1 Algorithms . . . 11
2.2 Basic probabilities . . . 12
2.3 Probabilistic algorithms . . . 12
2.3.1 Error amplification . . . 15
2.4 Examples of probabilistic algorithms . . . 16
2.4.1 Arithmetic corrector . . . 16
2.4.2 Trusting a flip . . . 17
2.4.3 ** Random walk in an undirected graph . . . 17
2.5 **Important inequalities . . . 19
2.5.1 Markov . . . 19
2.5.2 Chebyschev . . . 19
2.5.3 Chernoff-Hoeffding . . . 19

3 Hadoop: Distributed File System **21**
3.1 HDFS . . . 21
3.2 Map-Reduce . . . 22
3.3 A hard example: the edit distance . . . 22

4 Property Testing **25**
4.1 Is a function linear? BLR Linearity Test . . . 26
4.1.1 Monotonicity Test * . . . 27
4.2 Testing words . . . 28
4.2.1 Testing Membership for the Edit distance with moves . . . 28
4.2.2 Testing Membership for the Edit distance . . . 31
4.3 Testing Graphs . . . 34
4.4 Testing vs. Learning . . . 35
4.4.1 Learning a linear Classifier . . . 36
4.4.2 Learning a Community in a graph . . . 36
4.5 Exercices . . . 37

5 Streaming **39**
5.1 Moments in a stream of values . . . 39
5.1.1 Reservoir Sampling . . . 39
5.1.2 Morris Algorithm for estimating F_1 . . . 40
5.1.3 Estimating F_0 . . . 40
5.1.4 Basic estimator for F_2 . . . 41
5.2 Graph properties from a stream of edges . . . 42

5.2.1 Graph properties by sampling [**?**] . . . 42
5.2.2 Graph properties in a stream . . . 42

6 Social Networks **45**
6.1 Pagerank . . . 46
6.2 Clusters of a given graph . . . 46
6.2.1 Basics of linear algebra . . . 47
6.2.2 Spectral methods . . . 48
6.2.3 Modules via the modularity matrix . . . 49
6.3 Clusters in a stream of edges . . . 50
6.4 Random graphs . . . 50
6.4.1 Random graphs with a power law degree distribution and a cluster . . . 50
6.5 Dynamic Random graphs . . . 51
6.5.1 Uniform Dynamics . . . 51
6.5.2 Concentrated Dynamics . . . 51
6.5.3 General Dynamics . . . 51
6.5.4 Stream of edges . . . 51
6.6 Deciding properties . . . 51
6.6.1 Deciding a dynamic property: $\Diamond P$ . . . 52
6.6.2 Correlation between two streams . . . 53
6.7 Twitter streams . . . 53
6.8 Search by correlation . . . 55

7 Dimension reduction **57**
7.1 The fundamental result . . . 57
7.1.1 Random projections . . . 58
7.1.2 Gaussian distributions . . . 58
7.2 PCA: Principal Components Analysis . . . 59
7.2.1 Covariance . . . 60
7.2.2 Gram representation . . . 60
7.2.3 Principal components . . . 61
7.3 Python code . . . 61
7.3.1 Covariance, Eigenvectors . . . 61
7.3.2 Gram's decomposition . . . 62
7.3.3 PCA: reconstruction . . . 62
7.3.4 PCA: reconstruction via the Gram matrix . . . 64
7.4 Recommendation Systems . . . 65
7.4.1 Python's code: Recommendation Systems . . . 66
7.5 Applications . . . 69
7.6 SVD decompositions . . . 69

8 Learning **71**
8.1 Neural Networks . . . 71
8.1.1 Basic neuron . . . 71
8.1.2 MLP: Multilayers percepton . . . 72
8.1.3 Back Propagation . . . 72
8.2 Reinforcement Learning . . . 74
8.2.1 Markov Decision Processes and Probabilistic Automata . . . 74
8.2.2 Existence of strategies and Equivalence . . . 76

9 Python **77**
9.1 Random projections 78
9.2 Json to Dictionary 79
9.2.1 Json API 79

10 R **81**

Chapter 1

Introduction

The analysis of big data uses specific techniques which we present in these notes. It opens new perspectives as we can measure parameters associated with physical phenomena, have a better understanding and ask new questions. All scientific areas are concerned and here are some examples.

In astronomy, telescopes around the earth generate many streams of images: how do we efficiently process these streams? If we look for new interesting planets, where do we point these telescopes? Social networks provide many streams of data: how do we efficiently analyze them to predict an election, a trend, new markets and in general some parameters which are difficult to precisely define? For a given brand, can we find potential new markets or better know our clients? Netflix and Amazon have millions of clients and thousands of products. How do they predict a potential product we may be interested in?

The class is oriented towards Economy students so that we will insist on social networks and provide tools to analyze Twitter data at a large scale. We insist on the main concepts and give concrete examples.

In order to characterize the area of big data, one often uses the 4 Vs:

- Volume: generally speaking, if n is the main parameter for the size of the data, $n > O(10^9)$,
- Velocity: the data are not static but change in time,
- Variety: different sources provide data in various formats: numerical values, trees, relations,....
- Veracity: we should be aware that the data are not perfect, i.e. may be incorrect because of errors which are either random or malicious. A classical example is the stream of tweets on a given subject. There are malicious tweets, either created by bots or by individuals who pursue opposite goals.

In classical data analysis, one may look at temporal series such as the price of commodities on some market. If we measure 100 commodities every minute, we set a stream for 100 times $60.24 = 1440$ values per day or 4320000 values per month, i.e. $4,3\ 10^6$ values. Financial values are assumed correct, preformatted, and represent a moderate volume: they do not fit the *big data model*, but the classical data model.

In Business Intelligence, we start with an Information System and an OLAP schema to define a class of OLAP or Analysis queries. The Information System can be large with several data sources. We do not question the veracity of the data and stay in the realm of classical computations.

Consider two streams of data: the first one is a temporal serie which provides the value of the $\#bitcoin$ every minute, and the second one is the stream of Twitter tweets which contain the tag $\#bitcoin$. The volume of the Twitter stream is of the order of 20.10^3 tweets per hour, but varies in time. After a week of observations, we enter the realm of big data. Volume, Velocity with large sudden variations, Variety as the

two streams are structurally different, Veracity as a large proportion of the tweets on the *#bitcoin* or other cryptocurrencies, are generated by bots. We have all the ingredients of the big data.

Most of the techniques we use are probabilistic. We mostly sample the bigdata, either as a datawarehouse or as a stream. There is an underlying probabilistic space Ω and most of our decisions are taken with high probability. This is often written as :

$$Prob[\text{Condition holds}] > 0.9$$

which states that the probability that the Condition holds is larger than 0.9. We also write $1 - \delta$ instead of 0.9, as the notations are often equivalent.

The main techniques presented are:

- Hadoop/Map-Reduce: we store large data with a robust distributed file system, and use multiple processors to speed the analysis of the data.
- Sample the data, according to some distribution, as the uniform distribution. In general there is a distance between objects, and we can only guarantee that the data satisfy a property or are ε-far from the property.
- Streaming data: this is the core of the class. Streaming data are ubiquitous and need to be analyzed without being stored.
- Dimension reduction: if we have a table with many columns, could we reduce the number of columns (the dimension) ? This is the classical PCA (Principal Component Analysis), which we revisit. The typical application is a Recommendation system: a matrix A is such that $A(i, j) = 1$ if client i *likes* product j, and 0 otherwise. We know only 1% of the matrix A, and yet Amazon and Netflix predict which product you might be interested in.
- Learning: we can use millions on data to learn various concepts. In particular specific images of cats or dogs using neural networks, a specific model. Facebook analyzes 10^9 images each day and can write a text description of the image. In another example, we can learn strategies in games such as GO, by playing millions of games and improve a given strategy.

1.1 Mégadonnées

L'analyse des grandes données utilise des techniques spécifiques que nous présentons dans ces notes. Elle ouvre de nouvelles perspectives car elle permet de mesurer les paramétres associés aux phénoménes physiques, de mieux comprendre ces paramétres et de poser de nouvelles questions. Tous les domaines scientifiques sont concernés comme le montrent les exemples suivants.

En astronomie, les téléscopes autour de la Terre générent de nombreux flux d'images : comment traiter efficacement ces flux? Si nous cherchons de nouvelles planétes intéressantes, oé pointerons-nous ces télescopes? Les réseaux sociaux fournissent de nombreux flux de données : comment les analyser efficacement pour prévoir une élection, une tendance, de nouveaux marchés et en général des paramétres difficiles é mesurer précisément? Pour une marque donnée, pouvons-nous trouver de nouveaux marchés potentiels ou mieux connaétre nos clients? Netflix et Amazon ont des millions de clients et des milliers de produits. Comment prédisent-ils un produit potentiel qui pourrait nous intéresser?

Le cours est destiné aux étudiants en économie de sorte que nous insisterons sur les réseaux sociaux et fournirons des outils pour analyser les données de Twitter à grande échelle. Nous insistons sur les concepts principaux et donnons des exemples concrets.

Afin de caractériser la zone de grandes données, on utilise souvent les 4 Vs :

- Volume: en général, si n est le paramétre principal pour la taille des données, $n > O(10^9)$,
- Vitesse : les données ne sont pas statiques mais changent avec le temps,
- Variété d'articles : différentes sources fournissent des données sous différents formats : valeurs numériques, arbres, relations,.....
- Véracité : nous devons étre conscients que les données ne sont pas parfaites, c'est-é-dire qu'elles peuvent étre incorrectes en raison d'erreurs aléatoires ou malveillantes. Un exemple classique est le flux de tweets sur un sujet donné. Il y a des tweets malveillants, créés soit par des robots, soit par des individus qui poursuivent des objectifs opposés.

Dans l'analyse classique des données, on peut s'intéresser é des séries temporelles telles que le prix des matiéres premiéres sur certains marchés. Si nous mesurons 100 marchandises chaque minute, nous établissons un flux de 100 multiplié par $60 * 24 = 1440$ par jour ou par 4320000 par mois, soit des valeurs de $4,3.10^6$. Les valeurs financiéres sont supposées correctes, préformatées et représentent un volume modéré : elles ne correspondent pas au modèle de données le plus important, mais au modèle de données classiques.

En Business Intelligence, nous commenéons par un système d'information et un schéma OLAP pour définir une classe de requétes OLAP ou d'analyse. Le système d'information peut étre volumineux avec plusieurs sources de données. Nous ne remettons pas en question la véracité des données et restons dans le domaine des calculs classiques.

Considérons deux flux de données : le premier est une série temporelle qui fournit la valeur du $\#bitcoin$ chaque minute, et le second est le flux de tweets Twitter qui contient le tag $\#bitcoin$. Le volume du flux Twitter est de l'ordre de 20.10^3 tweets par heure, mais varie dans le temps. Aprés une semaine d'observations, nous entrons dans le domaine des grandes données. Volume, Vitesse avec de grandes variations soudaines, Variété car les deux flux sont structurellement différents, Véracité comme une grande proportion des tweets sur les $\#bitcoin$ ou autres cryptocurrencies, sont générés par des robots. Nous avons tous les ingrédients des grandes données.

La plupart des techniques que nous utilisons sont probabilistes. Nous échantillonnons la plupart du temps les bigdata, soit sous forme d'entrepôt de données, soit sous forme de flux. Il y a un espace probabiliste sous-jacent Ω et la plupart de nos décisions sont prises avec une probabilité élevée. C'est souvent écrit comme:

$$Prob[\text{Condition vraie}] > 0.9$$

qui indique que la probabilité que la condition soit vraie est supérieure é $0,9$. Nous écrivons aussi $1 - \delta$ au lieu de 0.9 et les notations sont équivalentes.

Les principales techniques présentées sont :

- Hadoop/Map-Reduce : nous stockons des données volumineuses avec un système de fichiers distribués qui duplique les fichiers et utilisons plusieurs processeurs pour accélérer l'analyse des données.
- échantillonner les données, selon une certaine distribution, comme la distribution uniforme. En général, il y a une distance entre les objets, et nous pouvons seulement garantir que les données satisfont une propriété ou sont ε-loin de la propriété.
- Streaming data : c'est le coeur du cours. Les données en continue sont omniprésentes et doivent étre analysées sans étre stockées.
- Dimension reduction : si nous avons un tableau avec plusieurs colonnes, pourrions-nous réduire le nombre de colonnes (la dimension) ? C'est l'ACP classique (Analyse en Composantes Principales), que nous revisitons. L'application typique est un système de recommandation : une matrice A est telle que $A(i,j) = 1$ si le client i *apprécie* le produit j, et 0 sinon. Nous ne connaissons que 1% de la matrice A, et pourtant Amazon et Netflix prédisent le produit qui pourrait vous intéresser.

- Learning : nous pouvons utiliser des millions de données pour apprendre divers concepts. En particulier des images spécifiques de chats ou de chiens utilisant des réseaux neuronaux, un modéle spécifique. Facebook analyse environ 10^9 images chaque jour et peut écrire une description textuelle de l'image. Dans un autre exemple, nous pouvons apprendre des stratégies dans des jeux comme GO, en jouant des millions de parties et en améliorant une stratégie donnée.

Traduit avec www.DeepL.com/Translator

Chapter 2

Preliminaries

2.1 Algorithms

Consider a binary word $x \in \{0,1\}^n$ of length n which may encode an integer. For example, $x = 1001$ encodes the integer 9. An algorithm A takes x as input, follows a sequence of elementary operations to produce an output $y \in \{0,1\}^m$: we write $A(x) = y$. Assume as elementary operations, the addition multiplication, division, test and loops, as they exist in all programming languages. Consider an arbitrary total function f such that:

$$f(x) = y$$

We say that the algorithm A implements f if $\forall x, y \;\; f(x) = y \leftrightarrow A(x) = y$.

Consider the following functions:

- $f_1(x) = 1$ if x is a prime number and 0 otherwise
- $f_2(x) = y$ if y is the smallest prime number greater than 1 in the prime decomposition of $x \neq 1$. Assume $f_2(1) = 1$.

For $x = 1001$, notice that $f_1(x) = 0$, as 9 is not a prime number. Notice also that $f_2(x) = 11$, the binary representation of the integer 3 which is the smallest prime number which divides x.

When the output y is 0 or 1, we consider f as a *Decision problem*, as we can interpret 0 as reject (or NO) and 1 as accept (or YES). The function f_1 is simply the Decision problem: *is x a prime number*? One often defines the set of inputs x such that $f(x) = 1$ as $L_f = \{x : \;\; f(x) = 1\}$ also called the language of f.

When the output y has arbitrary values, we consider f as a *Search problem*. The function f_2 is a search problem: *what is the smallest prime factor of x*?

In general x and y are finite structures, words, trees, graphs or hypergraphs. On an input x of size n, we may tolerate approximate answers in the following sense. For a search problem where y is an integer, $A(x)$ may be close to the correct answer y if:

$$y.(1-\varepsilon) \leq A(x) \leq y.(1+\varepsilon)$$

The parameter ε is the approximate error, say $\varepsilon = 10\%$ for example. If the correct answer is $y = 800$, a good approximate answer is in the interval $[720, 880]$, within $\varepsilon.y$ of the correct answer.

Another very useful approximation is the following:

$$Prob_{\Omega}[y.(1-\varepsilon) \leq A(x) \leq y.(1+\varepsilon)] > 0.9$$

How do interpret the probability? Imagine that the algorithm uses a special instruction *Flip a coin*, several times. It indirectly defines a probabilistic space Ω used in this definition. We need a quick recall on the basic concepts of probabilities.

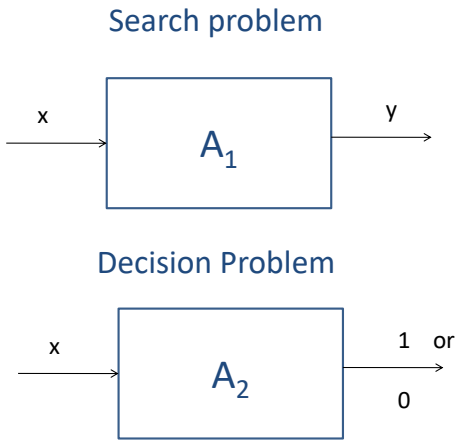


Figure 2.1: Decision and search problems

2.2 Basic probabilities

For a fair coin, we have the same chance to hit Head (H) or Tail (T). We say that the chance or probability of each event is 1/2. Assume we repeat two Flips, as in Figure 2.2 (a). We can represent this process as a tree with all 4 possibilities: HH, HT, TH, TT on each leaf of the tree. What is the chance to obtain HH? The Flips are independent, hence the probabilities multiply: it is $1/2 * 1/2 = 1/4$.

In Figure 2.2 (b), we have a biased coin the probability to have Head is (p=2/3), hence the probability to have Tail is (1/3).

Assume Algorithm A_1 computes a Search problem: the value obtained is on each leaf of the tree. Algorithm A_2 computes a Decision problem and its value is also on each leaf.

How do evaluate the probability of an event? Consider each leaf and evaluate if the event is true on the leaf. Take the sum of the probabilities of the leaves where the event is true.

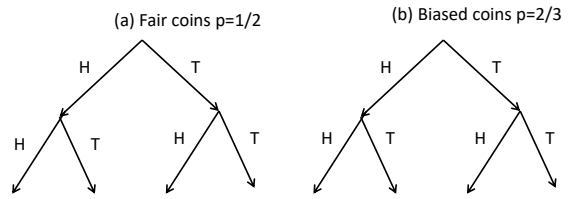
For the Algorithm A_1, what is the probability of the event: $[y.(1-\varepsilon) \leq A_1(x) \leq y.(1+\varepsilon)]$? For $\varepsilon = 10\%$, the event is true on the first 3 leaves and false on the $4th$ leaf. For the case (a), the probability is then $1/4 + 1/4 + 1/4 = 3/4$. For the case (b), the probability is $4/9 + 2/9 + 2/9 = 8/9$.

The probabilistic space Ω is defined as the set of leaves with their probabilities.

Suppose the algorithm Flips 30 coins: how many leaves are concerned? The number of leaves will be $2^{30} = (2^{10})^3 = 1024^3 \simeq 10^9$, a billion leaves. It is a typical situation for a probabilistic algorithm.

2.3 Probabilistic algorithms

A classical view is to only consider algorithms *without error*. We now consider probabilistic algorithms for Decision problems such that: if $x \in L$, the probability to accept will be close to 1 and if $x \notin L$, the probability



Probabilities	1/4	1/4	1/4	1/4	4/9	2/9	2/9	1/9
$A_1(x)$	810	790	785	13	810	790	785	13
$A_2(x)$	1	1	1	0	1	1	1	0

Figure 2.2: Two Flips with fair coins (a) and biased coins (b)

to reject will be close to 1.

A procedure without error can be implemented on a sequential or parallel machine model, made of hardware components which are not completely reliable. The software environment (Unix, for example) is also error prone, as every computer scientist will attest. Let us suppose a rate of error of 10^{-8} for a given machine. An imperfect procedure with a rate of error of 10^{-10} will not be distinguishable from a perfect procedure, because the error probability due to the algorithm is negligible in comparison with the error probability due to the machine. In practice, a probabilistic algorithm can be extremely useful.
A probabilistic algorithm is a constructive procedure which uses a new instruction: *random choice*. One can flip a coin or randomly choose a value among k equiprobable distinct values. In this chapter, a random choice is the selection of the values 0 or 1 with probability $\frac{1}{2}$. We can associate a non-deterministic machine with a probabilistic algorithm: just consider the random choice as a non-deterministic instruction. We define the notion of probabilistic acceptance of a language, through the use of the computation tree of specific non-deterministic Turing machines.

Definition 1 *A* **probabilistic program** *is a non-deterministic process whose computation tree is a complete binary tree, i.e. all the branches have the same length. At each leaf a deterministic process computes a value.*

For simplicity, suppose the value is 0 or 1. A *probabilistic execution* starts with the initial input x and follows at each step a possible transition of the program. At each non-deterministic node, the machine makes a *random choice* between two possible transitions with a uniform distribution. The computation tree is a complete binary tree of depth t where all the paths have equal probability. The probabilistic space is

$$\Omega_t = \{(\rho, \frac{1}{2^t}) : \rho \in \{0,1\}^t\}$$

The notion of acceptance is defined by comparing the number of accepting paths, i.e. paths with a value 1, $\text{acc}_M(x)$ and rejecting paths $\text{rej}_M(x)$ i.e. paths with a value 0. In an equivalent way, we talk of acceptance probability as the quotient of the number of accepting paths by the total number of paths:

$$\Pr_\Omega[M \text{ accepts } x] = \frac{\text{acc}_M(x)}{2^t}$$

$$\Pr_{\Omega}[M \text{ rejects } x] = \frac{\text{rej}_M x)}{2^t}$$

A **run** or **experiment** is a path in the computation tree leading to an accepting, or rejecting state.

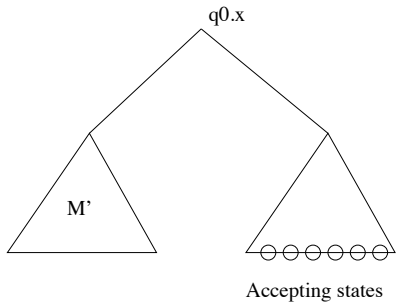


Figure 2.3: The computation tree of M'' which accepts with probability $> \frac{1}{2}$ iff $x \in \mathcal{L}$.

A *Monte-Carlo* algorithm accepts with the following condition:

Definition 2 *A language L is* Monte-Carlo-*decided by a program M if there exists a constant $\epsilon > 0$ such that:*

- *if $x \in \mathcal{L}$, then* $\Pr_\Omega[M \text{ accepts } x] \geq \frac{1}{2} + \epsilon$,
- *if $x \notin \mathcal{L}$, then* $\Pr_\Omega[M \text{ accepts } x] \leq \frac{1}{2} - \epsilon$.

Notice that ϵ must be constant, independent of the input x. In some cases we may accept or reject with probability 1, and the definition is asymmetric. Another class is a *Las Vegas* algorithm

Definition 3 *A language L is* Las-Vegas-*decided by a program M if it satisfies:*

- *if $x \in \mathcal{L}$, then* $\Pr_\Omega[M \text{ accepts } x] = 1$,
- *if $x \notin \mathcal{L}$, then* $\Pr_\Omega[M \text{ accepts } x] = 0$.

There is no error but there is no bound on the time to reach a decision. One may in certain cases bound the expected time to reach a decision.

Example 1 *Consider the computation tree below for $m = 16$ and $\epsilon = 0.09$. The number of accepting leaves (9) is not greater than $16/2 + 16 * 0,09 = 9,44$ and the input is not accepted for this $\epsilon = 0.09$. It is accepted for $\epsilon = 0.06$.*

A fundamental property of the class of *Monte-Carlo* algorithms is that the error probability of $\frac{1}{2} - \epsilon$ can be made arbitrarily small between 0 and $\frac{1}{2}$, in particular very close to 0. This property is called *error amplification* and is used repeatedly in all probabilistic situations.

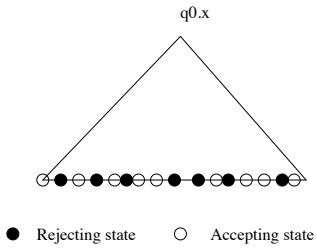


Figure 2.4: A probabilistic computation tree with 16 final states, 9 accepting and 7 rejecting.

2.3.1 Error amplification

Let us repeat k independent experiments and let us accept x if one of the experiments is accepting. In this case, the error probability is the probability of obtaining k rejecting answer, i.e. less than $p^k < (\frac{1}{2})^k$.

Let us show that if a problem is in the class BPP, one can also reduce the error probability to $(\frac{1}{2})^k$, i.e. to an exponentially small amount, after $O(k)$ experiments. Let us iterate $2m+1$ times the BPP algorithm and let us accept according to a majority test, i.e. if the number of accepting experiments is greater than m. Let p be the error probability $\frac{1}{2} - \epsilon$ and $q = 1 - p$. The error probability of this new procedure is the probability of not accepting when we should have, i.e. the probability μ of obtaining no more than m accepting answers. We have $\binom{2m+1}{i}$ possibilities of obtaining i accepting answers among the $2m+1$ cases.

$$\mu = \sum_{i=0}^{m} \binom{2m+1}{i} \cdot p^{2m+1-i} \cdot q^i = p \cdot p^m \cdot q^m \sum_{i=0}^{m} \binom{2m+1}{i} \cdot p^{m-i} \cdot q^{i-m}$$

$$\leq p \cdot p^m \cdot q^m \sum_{i=0}^{m} \binom{2m+1}{i} = p \cdot p^m \cdot q^m \cdot 2^{2m}$$

μ is indeed the sum on $i = 0, ..., m$ of the probability of obtaining i positive answers, and so $2m+1-i$ negative answers. By bounding p by 1, one obtains

$$\mu \leq p^m \cdot q^m \cdot 2^{2m} = (4pq)^m$$

If $m = c \cdot k$, then $\mu \leq [(4pq)^c]^k$. But $4pq = 4 \cdot (\frac{1}{2} - \epsilon) \cdot (\frac{1}{2} + \epsilon) = 1 - 4\epsilon^2 < 1$; there exists a constant c such that $(4 \cdot p \cdot q)^c < \frac{1}{2}$ and $\mu \leq (\frac{1}{2})^k$.

The key of the argument rests on the facts that $\mu \leq (4pq)^m$ and that $4pq = 1 - 4 \cdot \epsilon^2 < 1 - \delta$.

In the case of a PP algorithm, one cannot conclude that $4pq < 1 - \delta$: the probability of error can be $p = \frac{1}{2} - (\frac{1}{2})^n$ for a computation tree of depth n. One obtains then: $4pq = 4 \cdot (\frac{1}{2} - (\frac{1}{2})^n) \cdot (\frac{1}{2} + (\frac{1}{2})^n) = 1 - (\frac{1}{2})^{2n-2}$. In order to bound the error probability by $(\frac{1}{2})^k$, we need to find c such that $(1 - (\frac{1}{2})^{2n-2})^c < \frac{1}{2}$ and obtain $c > 2^{2n-3}$. We need to repeat $2m + 1 = 2c \cdot k + 1 > 2k \cdot 2^{2n-3} + 1$, i.e. $\Omega(2^{2n})$, an exponential number of experiments.

2.4 Examples of probabilistic algorithms

We give three examples of probabilistic algorithms. In each case, the probabilistic algorithm has an advantage compared with deterministic algorithms, usually for time or space:

- Arithmetic corrector
- Trusting a flip
- a random walk for the problem UGAP.

2.4.1 Arithmetic corrector

Assume some arithmetic circuits (addition, multiplication, division) which make errors. For two random $x, y \in_r [0, ...N]$, the *division* circuit computes $div(x, y) = (q, r)$ where q is the quotient and r is the rest. Assume (q, r) are correct with probability $1 - p$ and incorrect with probability p. For example $p = 0, 3$. Let us consider the quotient.

Could we reduce the error, to something very small such as 10^{-9}? Yes we can, with the simple probabilistic algorithm, often called the *Majority decision*. The problem is defined as:

Input: two integers x et y in the interval $[0, ...N]$.
Ouput: the quotient q in the division of x by y.

The algorithm A which solves the problem takes an additional parameter m which depends on the final error rate such as 10^{-9}:

$A(x, y, m)$**:**
Repeat $2m + 1$ times:
Generate r_i random in $[1, ..., N]$,
Compute $x_i = x.r_i$, $y_i = y.r_i$,
Let q_i be the quotient of $div(x_i, y_i)$
If $\{q_1, ...q_{2m+1}\}$ has an answer q' which appears at least $m+1$ times, called the majority answer, output q'.

Let us show that q' is correct with probability 10^{-9} if we choose some small value for m. The probability that we don't have a majority answer is the probability to have at least $m + 1$ or $m + 2$ or.... $2m + 1$ wrong answers. Let p be the error probability, for example $p = 0, 3$ and $q = 1 - p = 0.7$. The probability μ of not having a majority answer is:

$$\mu = \sum_{i=0}^{m} C_{2m+1}^{i} p^{2m+1-i}.q^{i} = p.p^{m}.q^{m} \sum_{i=0}^{m} p^{m-i}.q^{i-m} C_{2m+1}^{i} \leq p.p^{m}.q^{m}.2^{2m}$$

μ is the sum over all possible combinations of i among $2m + 1$ to have i correct answers and $2m + 1 - i$ incorrect answers. We bound p by 1 and obtain:

$$\mu \leq p^{m}.q^{m}.2^{2m} = (4pq)^{m}$$

Let $p = 1/3$, $q = 2/3$ and $4.p.q = 8/9$. In order to have $(4pq)^{m} \leq 10^{-9}$, choose $m \geq \frac{9.log10}{log9/8} \geq 180$.

This argument is valid aslong as $p \geq 1/2 - \epsilon$.

2.4.2 Trusting a flip

Consider two distant persons *Alice* and *Bob*. They want to flip a coin, but do not trust each other. How should they proceed? For example, Alice lives in Paris, Bob moved to New-York as they are getting divorced. They share a car and agree to decide the car's ownership by flipping a coin.

A Hashing function h takes an arbitrary input x and $h(x)$ is a binary word of a fixed length which satisfies two conditions:

- If $x' = x + \Delta(x)$, then $\mathsf{dist}(x, x')$ is large,
- Given y it is hard to find some x such that $h(x) = y$

$Sha256$ is a classical hash function where y is of length 256. Consider the following algorithm:

$A()$**:** Alice Hashes her choice (with the function Sha):
x=Sha('I choose head'),
Alice sends x to Bob,
Bob flips a coin and announces the result to Alice.

If Bob declares 'Head', Alice sends the message y='I choose head' and concludes that she won. Bob verifies that x=Sha(y) and is convinced that Alice had choosen 'head' before Bob flipped his coin. If Bob declares 'tail', Alice agrees that she lost and tells Bob that he wins. Both trust the mechanism. In this example, the Hashing function is necessary to build a trust in a random choice.

2.4.3 ** Random walk in an undirected graph

Let $G_n = (D_n, E)$ be an undirected graph with e edges. UGAP, Undirected Graph Accessibility Problem, is a decision problem defined as:

A. Input: An undirected graph and two nodes s and t.

Output: 1 if there is a path between s and t , 0 otherwise.

It is also known as GAP, Graph Accessibility Problem, in the case when the graph is symmetric. It is computable in polynomial time because there are P algorithms to compute a path between two nodes s and t. It is also in the class NL because UGAP is a restriction of GAP, an NL-complete problem. Is this problem in the class Ł? The answer to this question is not presently known. Let us show, however, that there exists a probabilistic algorithm in space $O(\log n)$, i.e. that UGAP is in the class RL. We will analyze a *random walk* from s which will eventually terminate in t on a positive instance.

Probabilistic algorithm for UGAP.

Iterate k times the procedure.

Let $u := s, i := 1$.
While $i < 2.n^3$
{ Consider the edges whose origin is u.
Select a random edge (u, u') with a uniform distribution[1].
$u := u'$, $i := i + 1$. If $u = t$, then UGAP $= 1$.}
UGAP $= 0$.

[1]If there are m edges, each one is selected with the probability $1/m$.

The space requirement of this algorithm consists of two registers, one to code u an arbitrary node in a graph with n nodes and the other for the integer i whose value is less than $2n^3$. Both use $O(\log\ n)$ bits. Let us show now that a random walk of length $O(n^3)$ has a probability greater than $\frac{1}{2}$ to find t if there exists a path between s and t.

The proof uses some classical results on Markov chains [10] to estimate the average time $T(i)$ necessary to visit all the nodes from a given node i. This average time $T(i)$ will be bounded by $O(n^3)$. We will conclude that a random walk of length greater than $T(i)$ will have a probability greater than $\frac{1}{2}$ to reach t if there exists a path from s to t.

The previous algorithm defines a Markov process with a transition matrix A, such that $a_{i,j}$ is the probability to reach the node i from the node j. We can then compute A , A^2,...A^k. The matrix A^k gives the probability to reach a node i from a node j after k steps. The *stationary* probabilities if they exist, are defined as the limit probabilities on each of the nodes and are represented by the vector π such that:

$$A.\pi = \pi$$

where $\sum_{i=1}^{n} \pi(i) = 1$. If $d(i)$ is the degree of the node i, then

$$\pi(i) = \frac{d(i)}{2e}$$

where $2e$ is the number of edges $(i,j) \in E$. The previous equation admits a unique solution, and this last expression is a solution. A classical result on Markov chains is: if the chain is irreducible (the graph G is connected), finite and aperiodic, then π is unique.

Definition 4 *Let $t(i,j)$ the expected number of transitions to reach j from i and $T(i)$ the expected number of transitions necessary for a random path to reach all the nodes from i.*

$T(i)$ is the inverse of the stationary probability $\pi(i)$ and $T(i) = \frac{2e}{d(i)}$. For an edge a, let $f(a)$ the number of times a random walk crosses the edge a and $I\!E(f(a))$ the expectation of this random variable, i.e. the average number of crossing of the edge a. We also use the fact that:

$$I\!E(f(a)) = \frac{1}{2e}$$

This expectation is also called *the stationary frequency of an edge* and is independent of the edge considered.

Lemma 1 *If i and j are two adjacent nodes of G , then $t(i,j) + t(j,i) \leq 2e$.*

Proof : If i and j are adjacent nodes, $t(i,j) + t(j,i)$, the average number of transitions for a random path from i towards j and back is $2e$ times E, the expected number of occurrences of an edge a. This expectation is independent of the edge a from the previous remark. So $t(i,j) + t(j,i) = 2e.E$. Consider the edge $a = (i,j)$. The expectation of the number of appearances of this edge, written E_a, is less than 1, because numerous paths from i to j and back do not follow this edge[2]. Therefore $t(i,j) + t(j,i) \leq 2e$.

From this lemma, we can deduce that if $d(i,j)$ is the distance between i and j i.e. the number of edges of the shortest path between i and j, then

$$t(i,j) + t(j,i) \leq 2e.d(i,j)$$

Lemma 2 $T(i) \leq 2e.(n-1)$.

[2]If we need to take this edge a, it is called an isthmus, and $E_a = 1$.

Proof : Let H be a spanning tree for the graph G. To explore all the nodes of G from a node s, it is necessary to cross all edges e of H in both directions. Therefore,

$$T(i) \leq \sum_{(j,j') \in H} (t(i_j, i_{j'}) + t(i_{j'}, i_j))$$

From the previous lemma, we conclude:

$$T(i) \leq 2e.(n-1)$$

Notice that $e < n^2$ and so $T(i) < 2.n^3$.

Theorem 1 *The problem* UGAP *is in the class* RL.

Proof : Generate a random path from s of length $2.n^3 + 1$. The probability to find t, if there exists a path between s and t is greater than $\frac{1}{2} + \epsilon$.

If there is no path between s and t, this procedure does not make any error. If there is a path, we repeat this procedure k times, the rate of error is less than $1/2^k$, and we obtain an RL algorithm.

2.5 **Important inequalities

An essential tool for probabilistic algorithms is to bound the possible error they could make. In general, we want to bound the probability that we deviate from a correct unknown value.

2.5.1 Markov

$$Prob[X \geq a] \leq I\!E[X]/a$$
$$Prob[X \geq a.I\!E[X]] \leq 1/a$$

2.5.2 Chebyschev

$$Prob[X - I\!E[X] \geq a] \leq Var(X)/a^2$$

$$Var(X) = I\!E[(X - I\!E[X])^2]$$

2.5.3 Chernoff-Hoeffding

Hoeffding is the case of bounded variables X_i. We select n independent variables $X_1, X_2, ...X_n$. Let

$$Y = \frac{\sum_{i=1...n} X_i}{n}$$

$$Prob[Y - I\!E[Y] \geq t] \leq e^{-2tn^2}$$

Chapter 3

Hadoop: Distributed File System

A first approach to Bigdata is to imagine more processors in order to speed the computations.. Given N processors, the goal is to process the data N times faster. It is only possible for very limited problems.

Another requirement is to improve the reliability. We distribute the data on different disks and different sites, to assure a better reliability. If 10% of the sites are down, the whole system still works and is only 10% slower.

We first describe the file HDFS file system, then the MapReduce approach and finally consider a hard example for MapReduce, the classical edit distance on strings.

3.1 HDFS

Each file is decomposed in blocks and each block is duplicated on several nodes representing some independent disk storage as in Figure 3.1.

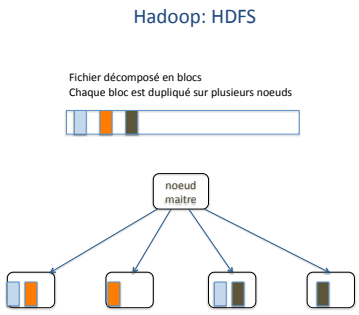


Figure 3.1: Nodes storing blocks of a file

When a node fails, the master node can still recover the file. In our example, we duplicate each block twice. If 1 node fails, we recover all the files. If we duplicate each block d times, we tolerate $d-1$ failures.

3.2 Map-Reduce

Suppose a large piece of data x can be decomposed into a partition $x_1, x_2, ...x_k$ where each x_i is located on a specific processor. We could apply the same function *map* to all the x_i, then exchange some information between the processors. Then we apply the same function *reduce* on all the processors, and finally assemble the result, as in the Figure 3.2. The information is represented as pairs $< k, v >$ for keys, values.

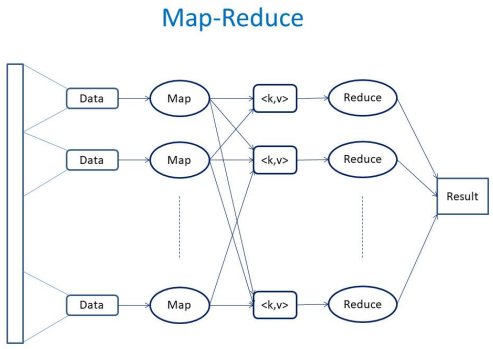


Figure 3.2: MapReduce implementations

Typical examples are:

- Hamming distance between two strings. Given two large strings x, x' of size n, the Hamming distance is the number D of positions i such that $x_i \neq y_i$. The relative Hamming distance d is D/n and is best understood as a percentage such as 2% or 10%.

  If we partition each string x into contiguous segments of length n/k, i.e. $x = x_1.x_2, ...x_k$, we can assign each x_i to each processor i. The map function computes the local Hamming distance. The Reduce function makes the sum and normalizes the value.

- Words counting. A given text can be partitioned as in the previous example. The map function computes a set of (key, value) where the key is a word and the value is the number of occurrences of the word in each part of the text. We reduce by taking the sum of the values for a given word. The final result makes a similar operation.

3.3 A hard example: the edit distance

The classical *edit distance* on words is a standard measure between two words w and w'. An *edit* operation is a *deletion*, an *insertion* or a *modification of a letter*. The *absolute edit distance* is the minimum number of edit operations to transform w into w' and the *relative edit distance*, $\mathsf{dist}(w, w')$, is the absolute edit distance divided, by $Max(|w|, |w'|)$. We mainly use the relative distance, a value between 0 and 2.

The *edit* operations are:

- Deletion of a has cost 1,
- Insertion of a, has cost 1,
- Modification of a into b has cost 1,

An additional operation is a *move* : it selects a factor (subword), suppress it and insert it at some other location, at the cost of 1, often called cut/pace. The *edit distance with moves*, $\mathsf{dist}_m(w, w')$, is the generalization when this extra operator can be used.

Two words w, w' are ε-close if $\mathsf{dist}(w, w') \leq \varepsilon$. The distance between a word w and a language L of timed words is defined as $\mathsf{dist}(w, L) = Min_{w' \in L} \mathsf{dist}(w, w')$.

Examples: let $w_1 = aaabbbba$ and $w_2 = bbbbaaaa$. Then $\mathsf{dist}(w, w') = 6/8 = 3/4$ whereas $\mathsf{dist}_m(w, w') = 1/8$.

The edit distance between two words w_1, w_2 is computable in polynomial time. Let $A(i, j)$ be the array where w_1 appears on the top row ($i = 1$) starting with the empty character ε, w_2 appears on the first column starting with the empty character ε as in Figure 3.3, where $w_1 = aba$ and $w_2 = aab$. The value $A(i, j)$ for $i, j > 1$ is the edit distance between the prefix of w_1 of length $j - 2$ and the prefix of w_2 of length $i - 2$. Let $\Delta(i, j) = 0$ if the letter symbols are identical, 1 otherwise. It is the edit distance between two letters.

For $i, j > 1$, there is a simple recurrence relation between $A(i, j), A(i-1, j), A(i, j-1)$ and $A(i-1, j-1)$, which reflects the 3 possible transformations: deletion of $w_1(i-2)$, deletion of $w_2(j-2)$ or edition of the last letters. Hence:

$$A(i, j) = Min\{A(i, j-1) + w_1(i-2), A(i-1, j) + w_2(j-2), A(i-1, j-1) + \Delta(i, j)\}$$

In the example of Figure 3.3, the timed edit distance is 2, and we can trace the correct transformations by tracing the Minimums for each $A(i, j)$:

		w_1			
	ε	**a**	**b**	**a**
ε	0	1	2	3
a	1	0	1	2
w_2 a	2	1	1	1
b	3	2	1	2

Figure 3.3: Classical array $A(i, j)$ for the edit distance between $w_1 = aba$ and $w_2 = aab$.

The edit distance is can only be solved in time $O(n^2)$ and Hadoop will not help. There are very few problems where Hadoop can provide a significant gain.

Chapter 4

Property Testing

In the case of a decision problem, i.e. the output is 0 or 1, the probabilistic approximation provides a first approach. We want a weaker approximation, in order to consider algorithms which just sample the data and provide useful approximations. The ideal situation is to fix some parameter ε and on an input of size n admit a number of samples which only depends on ε but not on n and a time complexity which only depends on ε. If it is not possible, we may have a number of samples $O(\sqrt{n})$, which is sublinear. In both cases, we do not need to read the entire input.

Given a distance between inputs, an ε-tester for a property P accepts all inputs which satisfy the property and rejects with high probability all inputs which are ε-far from inputs that satisfy the property. Inputs which are ε-close to the property determine a gray area where no guarantees exists. These restrictions allow for sublinear algorithms and even $O(1)$ time algorithms, whose complexity only depends on ε.

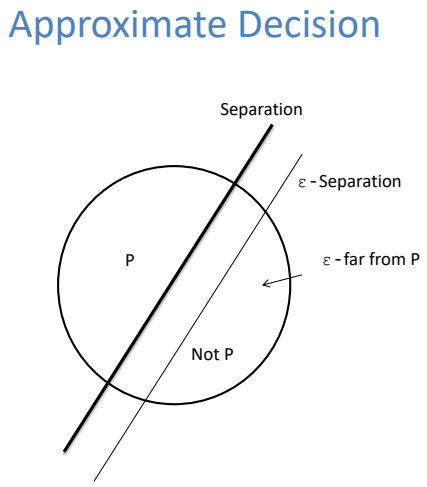


Figure 4.1: The Property Testing schema.

Let **K** be a class of finite structures with a normalized distance dist between structures, i.e. dist lies in $[0, 1]$. The structure **K** can be word, a tree, a graph or a more general structure.

For any $\varepsilon > 0$, we say that $U, U' \in \mathbf{K}$ are *ε-close* if their distance is at most ε. They are *ε-far* if they are not ε-close. In the classical setting, satisfiability is the decision problem whether $U \models P$ for a structure $U \in \mathbf{K}$ and a property $P \subset \mathbf{K}$. A structure $U \in \mathbf{K}$ *ε-satisfies* P, or U is ε-close to $\mathbf{K}$ or $U \models_\varepsilon P$ for short, if U is ε-close to some $U' \in \mathbf{K}$ such that $U' \models P$. We say that U is ε-far from $\mathbf{K}$ or $U \not\models_\varepsilon P$ for short, if U is not ε-close to $\mathbf{K}$.

Definition 5 (Property Tester [7]) *Let $\varepsilon > 0$. An* ε-tester *for a property $P \subseteq \mathbf{K}$ is a randomized algorithm A such that, for any structure $U \in \mathbf{K}$ as input:*
(1) If $U \models P$, then A accepts;
(2) If $U \not\models_\varepsilon P$, then A rejects with probability at least $2/3$.[1]

A *query* to an input structure U depends on the model for accessing the structure. For a word w, a query asks for the value of $w[i]$, for some i. For a tree T, a query asks for the value of the label of a node i, and potentially for the label of its parent and its j-th successor, for some j. For a graph a query asks if there exists an edge between nodes i and j. We also assume that the algorithm may query the input size. The *query complexity* is the number of queries made to the structure. The *time complexity* is the usual definition, where we assume that the following operations are performed in constant time: arithmetic operations, a uniform random choice of an integer from any finite range not larger than the input size, and a query to the input.

Definition 6 *A property $P \subseteq \mathbf{K}$ is* testable*, if there exists a randomized algorithm A such that, for every real $\varepsilon > 0$ as input, $A(\varepsilon)$ is an ε-tester of P whose query and time complexities depend only on ε (and not on the input size).*

Property testing was introduced in [7], building on earlier notions of Program testing [4]. The main definition is inspired by the IP and PCP complexity classes, as an attempt to have a probabilistic generalization of the class NP. It is a statistics based approximation technique to decide if either an input satisfies a given property, or is far from any input satisfying the property, using only few samples of the input and a specific distance between inputs. The idea of moving the approximation to the input was implicit in *Program Checking* [3, 13], in *Probabilistically Checkable Proofs* (PCP) [2], and explicitly studied for graph properties under the context of Property Testing [7].

The class of sublinear algorithms has similar goals: given a massive input, a sublinear algorithm can approximately decide a property by sampling a tiny fraction of the input. The design of sublinear algorithms is motivated by the recent considerable growth of the size of the data that algorithms are called upon to process in everyday real-time applications, for example in bioinformatics for genome decoding or in Web databases for the search of documents. Linear-time, even polynomial-time, algorithms were considered to be efficient for a long time, but this is no longer the case, as inputs are vastly too large to be read in their entirety.

4.1 Is a function linear? BLR Linearity Test

Let $f : F_2^n \to F_2$ be a boolean function. Such a function is linear if either property is satisfied:

- $\forall x, y \in F_2^n \quad f(x) + f(y) = f(x + y)$
- $\exists a \in F_2^n \quad f(x) = a.x$, i.e. $f(x) = \sum_{i \in S} x_i$ for $S \subseteq \{1, 2, ...n\}$

[1] The constant $2/3$ can be replaced by any other constant $0 < \gamma < 1$ by iterating $O(\log(1/\gamma))$ the ε-tester and accepting iff all the executions accept

Notice that these 2 conditions are equivalent. Clearly the second condition implies the first. To show that the first implies the second, use an induction on n. For $n = 1$, find a by setting $x = 1$. If $f(1) = 1$ then $a = 1$ otherwise $a = 0$. Assume the property true for $n - 1$. Find a_n by checking $f(1, 0, ...0)$. Any x can be written as $(1, 0, ...0) + (0, x_2, ...x_n)$ or as $(0, 0, ...0) + (0, x_2, ...x_n)$. We then apply the induction hypothesis and prove the result.

Consider the approximate version of the linearity conditions:

- for most $x, y \in F_2^n$ $f(x) + f(y) = f(x + y)$
- $\exists a \in F_2^n$ such that for most x $f(x) = a.x$, i.e. $f(x) = \sum_{i \in S} x_i$ for $S \subseteq \{1, 2, ...n\}$

It is interesting to note that the second condition implies the first and it is not obvious to prove the opposite, i.e. the first condition implies the second. It is another motivation to introduce the BLR test.

Let K be the class of boolean functions $f : F_2^n \to F_2$ and let $\mathcal{L}$ the class of linear functions. The distance between two functions f, g is the number of inputs on which they disagree, i.e. $\frac{\#x:f(x)\neq g(x)}{2^n} = Prob[f(x) \neq g(x)]$. Given a function f and a value i, it is natural to ask for $f(i)$ and this is a *query*. Can we efficiently test if $f \in \mathcal{L}$ or if f is ε-far from $\mathcal{L}$?

The answer is "yes", with the following test introduced by Blum, Luby and Rubinfeld in [4]:

BLR Test(f)
Input: $f : F_2^n \to F_2$

Repeat 3 times:
1. Generate uniformly and independently $x, y \in F_2^n$,
2. Reject if $f(x) + f(y) \neq f(x + y)$,

Accept.

If $f \in \mathcal{L}$, clearly f passes the Test with probability 1. We have to prove that if f is ε-far to a linear function then $Prob[f$ passes the test$] \leq c$. Equivalently, by contraposition, If $Prob[f$ passes the test$] > c$, then f is ε-close to a linear function. We will show in theorem **??** that we can take $c = 1 - \varepsilon$. It is a hard part of the result, which we admit. The proof requires the use of Fourier Analysis.

4.1.1 Monotonicity Test *

Let $f : F_2^n \to \{0, 1, ...r\}$ be a discrete function. Such a function is monotone if:

$$f(0, x_{-i}) \leq f(1, x_{-i})$$

Monotonicity Test(f, t)
Repeat t times
1. Generate uniformly $i \in_r \{1, 2, ...n\}$ and $x_{-i} \in F_2^{n-1}$,
2. Reject if $f(0, x_{-i}) > f(1, x_{-i})$

Accept.

Clearly, if f is monotone, it satisfies the Monotonicity Test with probability 1. The main question is to bound t to insure that a function f which is ε-far from monotone is rejected with high probability.

Theorem 2 *For $t = \Omega(n/\varepsilon)$, the Monotonicity Test rejects every function f which is ε-far from monotone.*

Proof : Equivalently, let us show that the probability that the test rejects in one trial is at least ε/n when f is ε-far. The number of possible edges of the hypercube is $n.2^{n-1}$. The Tester selects an i, and an hyperplane A_i separating the nodes with $x_i = 0$ from the nodes $x_i = 1$. Let α_i be the number of edges which cut the hyperplanes with $f(0, x_{-i}) > f(1, x_{-i})$. Then the Tester rejects with probability:

$$\frac{\sum_i \alpha_i}{n.2^{n-1}}$$

Consider the following partial Corrector: for each edge e of the hypercube which cuts the hyperplane, if $f(0, x_{-i}) > f(1, x_{-i})$, then switch the values of f. A case by case analysis shows that the new function f_i has no more errors than f on each hyperplane A_j, i.e. for $j = 1, ...n$, $\alpha_j \geq \alpha'_j$. For every $A_1, ...A_n$ we make two modifications for every edge with an error: hence f can be made monotone after modifying at most $2.\sum_i \alpha_i$ values. If f is ε-far from monotone, then $2.\sum_i \alpha_i \geq \varepsilon.2^n$.

The probability to reject is $\frac{\sum_i \alpha_i}{n.2^{n-1}} \geq \frac{\varepsilon.2^{n-1}}{n.2^{n-1}} = \frac{\varepsilon}{n}$.

4.2 Testing words

Consider the following operations on strings:

1. Modification of a letter
2. Insertion of a letter
3. Deletion of a letter
4. Move of a factor (substring), i.e. Cut/Paste

Given two strings w, w', the absolute distance is the minimum number of operations (among some of the 4 possibilities we select) necessary to transform w into w'. The distance $\mathsf{dist}(w, w')$ is the relative distance divided by the maximum length of w, w'. Notice that dist is symetric ans satisfies the triangular inequality.

If we only take the Modification, we get the Hamming distance dist_H. If we take Modification, Insertion, Deletion we get the Edit distance dist_E and if take all 4 operations we get the Edit distance with Moves dist_M. Notice that:

$$\mathsf{dist}_M(w, w') \leq \mathsf{dist}_E(w, w') \leq \mathsf{dist}_H(w, w')$$

4.2.1 Testing Membership for the Edit distance with moves

Let $\mathsf{ustat}_k(w)$ be the *uniform statistics of order* k, i.e. the k-gram of the word w. For a word w of length n on an alphabet Σ, the vector $\mathsf{ustat}_k(w)$ of dimension $|\Sigma|^k$ gives the density of the subwords u_i of length k. Let $\#u$ be the number occurrences of a subword u and $u_1, u_2, ...u_{|\Sigma|^k}$ a lexicographic enumeration of the u_i's.

$$\mathsf{ustat}_k(w) = \frac{1}{n-k+1}. \begin{pmatrix} \#u_1 \\ \#u_2 \\ ... \\ \#u_{|\Sigma|^k} \end{pmatrix}$$

We can also interpret $\mathsf{ustat}_k(w)$ as the distribution over u_i's observed on a random position $1 \leq i \leq n-k+1$ if $u = w[i].w[i+1]...w[i+k-1]$.

As an example, for binary words, and $k = 2$, there are 4 possible subwords of length 2, which we take in lexicographic order. Let $\Sigma = \{a, b\}$ be the alphabet. For the word $w = aaabbb$, $\mathsf{ustat}_2(w) = \frac{1}{5}(2, 1, 0, 2)$, i..e. $\#aa = 2$, and the first component is $\frac{2}{5}$. If we extrapolate to a word

$$w = aaaaaaaaaabbbbbbbbbb$$

then:

$$\mathsf{ustat}_k(w) = \frac{1}{19}.\begin{pmatrix} 9 \\ 1 \\ 0 \\ 9 \end{pmatrix} \simeq \begin{pmatrix} 0.5 \\ 0 \\ 0 \\ 0.5 \end{pmatrix}$$

The distance between two vectors u and v is the L_1 distance, $\sum_i |u(i) - v(i)|$. In the previous example the distance is $\frac{1}{2.19} + \frac{1}{19} + \frac{1}{2.19} = \frac{2}{19} \simeq 0.1$. If we extrapolate to a word of length n, then the error would be $\frac{2}{n-1}$. But $\lim_{n\to\infty} \frac{2}{n-1} = 0$.

For a regular expression, what is $\{\mathsf{ustat}_k(w) : w \in L(A)\}$? The answer is surprisingly simple: consider the regular expression a^* what is the limit of $\lim_{n\to\infty} \mathsf{ustat}_k(w)$ when $w \in a^*$? In this case it is simple as:

$$\mathsf{ustat}_k(w) = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = s_{a^*}$$

for all k. The limit is identical. Similarly for $b*$:

$$\mathsf{ustat}_k(w) = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} = s_{b^*}$$

For the regular expression $(ab)^*$, if $w \in (ab)^*$ then:

$$\mathsf{ustat}_2(w) = \begin{pmatrix} 0 \\ 0.5 + \varepsilon \\ 0.5 - \varepsilon \\ 0 \end{pmatrix}$$

as there is 1 more occurrence of ab, compared to ba, so $\varepsilon \simeq 1/n$ if n is the length of w.

$$\lim_{n\to\infty} \mathsf{ustat}_2(w) = \begin{pmatrix} 0 \\ 0.5 \\ 0.5 \\ 0 \end{pmatrix} = s_{(ab)^*}$$

Notice that for $k = 3$:

$$\lim_{n\to\infty} \mathsf{ustat}_3(w) = \begin{pmatrix} 0 \\ 0 \\ 2/3 \\ 0 \\ 0 \\ 1/3 \\ 0 \\ 0 \end{pmatrix}$$

Just imagine *abababababab*.... The only subwords of length 3 are *aba* and *bab*. In this case *aba* is twice more frequent than *bab*.

For any word u, we can similarly compute $\lim_{n\to\infty} \mathsf{ustat}_k(w) : w \in u^*$. For the regular expression $r_1 : a^*.b^*$, we take the following approach: any valid word w has a λ_1 proportion in a^* and a λ_2 proportion of b^*, such that $\lambda_1 + \lambda_2 = 1$. Hence the $\mathsf{ustat}_k(w)$ vector car be written as:

$$\lambda_1.s_{a^*} + \lambda_2.s_{b^*}$$

as represented by the line joining the two points s_{a^*} and s_{a^*} in Figure 4.2.

This a general strategy: for the regular expression $b^*.(ab)^*.(abb)^*$, we first compute each limit for a given k. We need to compute the vector associated to $(abb)^*$, as we already computed the previous vectors.

$$\lim_{n\to\infty} \mathsf{ustat}_2(w) = \begin{pmatrix} 0 \\ 1/3 \\ 1/3 \\ 1/3 \end{pmatrix} = s_{(abb)^*}$$

Any valid word w for $r_2 : b^*.(ab)^*.(abb)^*$ will have a statistical vector of the form:

$$\lambda_1.s_{b^*} + \lambda_2.s_{(ab)^*} + \lambda_3.s_{(abb)^*}$$

where $\sum_i \lambda_i = 1$, as indicated in Figure 4.2.

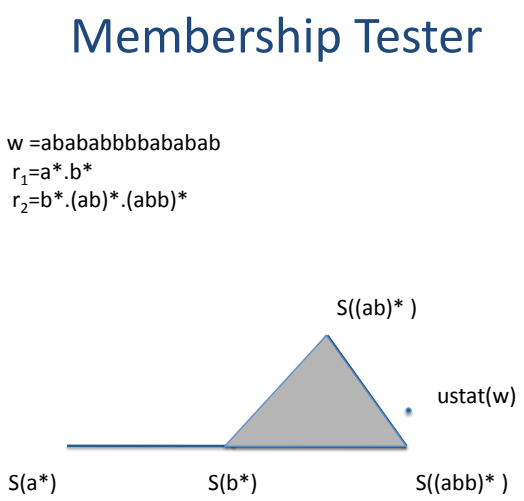


Figure 4.2: Polytopes for the regular expressions r_1 and r_2

Given a regular expression r, we first construct the union of polytopes $\bigcup_i P_i$ associated with the regular expression, as explained on these two examples. Assume we can compute the Euclidian distance *dist* of a

point x to a polytope in the statistical space. It generalizes the basic calculus for the distance *dist* between a point and an hyperplane, and can be formalized as a linear program. We can then apply the following tester:

Tester $T(w, r, \varepsilon)$

1. Sample $1/\varepsilon^2$ factors u of length $k = 1/\varepsilon$ of w_n,
2. Construct $\widehat{\mathsf{ustat}_k(w)}$, i.e. the ustat_k vectors on the samples,
3. Accept if there is a polytope P_i such that $dist(\widehat{\mathsf{ustat}_k(w)}) \leq \varepsilon$

In the example of Figure 4.2, the distance between $w = abababbbbabababb$ and $r_2 : b^*.(ab)^*.(abb)^*$ is 2/15. First move the last *ababab* in front, and remove the last b. If $\varepsilon \simeq 1/8$, the tester takes 64 samples of length 8: it would only make sense if w is very large, i.e. of length 10^6 for example. Based on only 64 samples, we validate w or not.

It is simple to see that if w is valid for r_2, the Tester accepts with probability 1. It is slightly more difficult to see that if w is ε-far of r_2, the Tester will reject with high probability. The argument uses the following:

- For each i, $\mathbb{E}[\widehat{\mathsf{ustat}_k(w)}(i)] = \mathsf{ustat}_k(w)(i)$. Hence we can use a Chernoff's bound to have a small error probability for the difference between $\widehat{\mathsf{ustat}_k(w)}(i) - \mathsf{ustat}_k(w)(i) > t$. With a union bound we conclude that $\widehat{\mathsf{ustat}_k(w)}(i)$ and $\mathsf{ustat}_k(w)(i)$ are close.
- The distance to a polytope is approximately the edit distance with moves.
- Hence if w is far from r, it will be far from any polytope P_i and we will reject with high probability, given only $\widehat{\mathsf{ustat}_k(w)}$.

4.2.2 Testing Membership for the Edit distance

Given an automaton $\mathcal{A}$, let $L(\mathcal{A})$ be the language accepted. A Tester for the Edit distance has the following structure. We separate the strongly connected components $C_1....C_p$ and the possible paths $\Pi = C_1....C_l$ from C_1 which contains the initial state to C_l which contains a final state, such that C_{i+1} is reachable from C_i. We first construct a Tester T_1 for one component C, then a Tester for $\Pi = C_1....C_l$ and finally a tester for $L(\mathcal{A})$.

Tester $T_1(w, C)$

1. Sample a factor u of length $k = 2.(m + 1)/\varepsilon$ of w_n,
2. If u is C-compatible Accept else Reject.

Tester for a Component C

We say that w is C-compatible if there is run for w in C, i.e. there are two states q, q' such that we reach q' from q reading w. We want to show that if w is ε-far from C, there are many incompatible factors u of some length. We conceive the following Corrector for C.

Start w in some state q which maximizes the length of a run. At this point we have a *cut*: we remove the letter, and start again from another state q_1 which maximizes another run. We then introduce a *link*, a small word of length less than m to connect q' with q_1. We made at most $m+1$ Edit operations. Let h the number of cuts in w. We write

$$w = w_1 \ |_1 \ w_2 \ |_2 \ w_3 \ _3| \ ... \ |_h \ w_{h+1}$$

for a word with h cuts.

Lemma 3 *Let C have at most m states. If w has h cuts, then it is $h.(m+1)$ close to C.*

Proof : We just correct w with the sequence of modifications along the cuts, defined by the previous Corrector.

By contraposition, if w is ε-far from C, we expect many cuts. Observe that if a sample u contains 2 cuts, it is necessarly incompatible, i.e. a witness that w is not accepted by C, for any initial and final state. We need to bound the size of a sample so it occurs with a constant probability.

Let $\alpha_i = |\{w_j : \ 2^{i-1} \leq |w_j| < 2^i\}|$ where $|w_j|$ is the length of w_j. By definition $h = \sum_i \alpha_i$ is the number of cuts. We need to find a bound i_l such that:

$$\sum_{0 \leq i \leq i_l} \alpha_i \geq \varepsilon.n$$

It will guarantee that samples of length $k = 2^{i_l}$ will contain many incompatible factors.

Lemma 4 *If w is ε-far from C, then the probability to find an incompatible factor u of length $2.k = 8.(m+1)/\varepsilon$ is greater than $c = 3.\varepsilon/8.(m+1)$.*

Proof :

We need to estimate $\sum_{0 \leq i \leq i_l} \alpha_i$ when we choose $k = 2^{i_l}$. First let us estimate $\beta = \sum_{i \geq i_l} \alpha_i$, i.e. for large i. There are at most $n/2^{i_l}$ feasible w_j of length larger than 2^{i_l}, i.e. $\beta \leq n/2^{i_l}$. Hence

$$\sum_i \alpha_i = \sum_{0 \leq i \leq i_l} \alpha_i + \beta \geq \varepsilon.n/(m+1)$$

$$\sum_{0 \leq i \leq i_l} \alpha_i \geq \varepsilon.n/(m+1) - \beta \geq \varepsilon.n/(m+1) - n/2^{i_l}$$

Let $k = 4.(m+1)/\varepsilon = 2^{i_l}$, or $i_l = \log(4.(m+1)/\varepsilon)$. Then

$$\sum_{0 \leq i \leq i_l} \alpha_i \geq 3.\varepsilon.n/4.(m+1)$$

Hence:

$$\beta = \sum_{i \geq i_l} \alpha_i \leq n/2^{i_l} \leq \varepsilon.n/4.(m+1)$$

Let us estimate the probability to have two consecutive small w_j, w_{j+1} in a sample u which starts in w_j. If this probability is large, it will guarantee that a sample of length $2.k$ starting in w_j is incompatible. We bound the probability that we hit a small w_j such that the following w_{j+1} is large.

$$Prob[|w_j| \leq k] \geq 3.\varepsilon/4.(m+1)$$

It is the weight of the small blocks: in the worst case they are of length 1 and their weight is $3.\varepsilon/4.(m+1)$. Consider now, the probability that the random u starts on a small w_j followed by a large w_{j+1}:

$$Prob[|w_{j+1}| > k \ \ | \ \ |w_j| \leq k]$$

For this event, a large block of size at least $4.(m+1)/\varepsilon$ follows a small block: there are only $\varepsilon.n/4.(m+1)$ small blocks which could be used. There remains $\varepsilon.n/2.(m+1)$ small blocks. Let us call A the set of all positions determined

by the small blocks followed by the large blocks. The remaining $\varepsilon.n/2.(m+1)$ small blocks occupy at least $n-\mid A \mid$ positions. Hence:

$$Prob[|w_{j+1}| > k \mid |w_j| \leq k] \leq Prob[w_j \in A]/2 \leq (1-\varepsilon/2.(m+1))/2 = 1/2 - \varepsilon/4.(m+1) \leq 1/2$$

By contraposition:

$$Prob[|w_{j+1}| \leq k \mid |w_j| \leq k] \geq 1/2$$

We can then bound the probability that a sample of weight $2k$ is incompatible:

$$Prob[\text{a sample } u \text{ of weight } 2k \text{ is incompatible }] \geq Prob[|w_j| \leq k].Prob[|w_{j+1}| \leq k \mid |w_j| \leq k]$$

$$\geq (3.\varepsilon/4(m+1)).1/2 \geq 3.\varepsilon/8.(m+1)$$

Hence we take $c = 3.\varepsilon/8.(m+1)$.

We can conclude:

Theorem 3 *The Tester T_1 accepts if $w \in L(\mathcal{A})$ and rejects with probability $3\varepsilon/4(m+1)$ if w is ε-far from $L(\mathcal{A})$*

Tester for a sequence Π of compatible connected components

Consider the following decomposition for $\Pi = C_1, C_2$, as in Figure 4.3, which we can generalize for an arbitrary Π: start in all possible states of C_1 wich are accessible from the initial state and take the longest compatible prefix w_1. It determines a cut of weight c. We continue in a similar way until we reach cuts of total weight at least $\varepsilon.n/2$ for C_1. We then switch to cuts for C_2. The position of the last cut for C_1 determines a border between C_1 and C_2, set by the intervals I_1 and I_2. If there are cuts for C_1 of weight less than $\varepsilon.n/2$, or cuts for C_1 of weight $\varepsilon.n/2$ and cuts for C_2 of weight less than $\varepsilon.n/2$, then the word w_n is ε-close to the regular expression associated with $\Pi = C_1, C_2$.

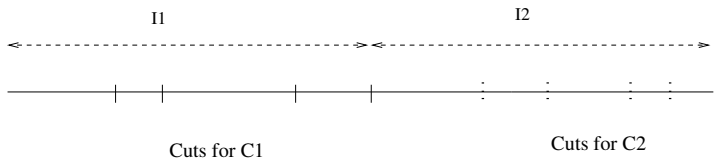


Figure 4.3: A possible decomposition of w_n into feasible components for $\Pi = C_1, C_2$.

We say that two independent samples $u_1 < u_2$ are $\Pi = C_1.C_2$ compatible if u_1 is compatible for C_1 or C_2 or $C_1.C_2$ and u_2 is compatible accordingly. We generalize to $\Pi = C_1....C_l$.

Tester $T_2(w, \Pi)$
$\Pi = C_1....C_l$

1. Sample l independent factors $u_1 < u_2 < ... < u_l$ of length $k = 2.(m+1)/\varepsilon$ of w_n,
2. If $u_1 < u_2 < ... < u_l$ is Π-compatible Accept else Reject.

If w_n is ε-far for Π, there exists such a decomposition, given as (I_1, I_2) where there are heavy cuts, i.e. of weight greater than $\varepsilon.n/2$, for C_1 and C_2. We then show that there are many samples u of a some finite weight which contain a w_i for the C_1 decomposition and a w_j for the C_2 decomposition. This decomposition generalizes to $\Pi = C_1....C_l$ by taking cuts for C_{i_1} of weight $\varepsilon.n/l$, cuts for C_{i_2} of weight $\varepsilon.n/l$ until possible

cuts for C_l of weight $\varepsilon.n/l$.

If w_n is ε-far from $\Pi = C_1, C_2$, we know that there are many samples u's of weight k incompatible for C_1 in the interval I_1 and many samples u's of weight k incompatible for C_2 in the interval I_2. The density of the incompatible samples is greater than $3.\varepsilon.n/4.(m+1)$. We can then conclude that the Tester will reject with constant probability.

Lemma 5 *If w_n is ε-far from $\Pi = C_1, C_2$ then the Word Tester rejects with constant probability.*

Proof : Assume w_n is ε-far from $\Pi = C_1, C_2$. Consider two samples $u_1 < u_2$ taken independently. If u_1 is incompatible for C_1 and u_2 is incompatible for C_2, then the Tester rejects. Hence:

$$Prob[\text{Tester rejects}] \geq Prob[u_1 \text{ incompatible for } C_1 \wedge u_2 \text{ incompatible for } C_2] \geq$$

$$Prob[(u_1 \in I_1 \wedge u_1 \text{ incompatible for } C_1) \wedge (u_2 \in I_2 \wedge u_2 \text{ incompatible for } C_2] \geq$$

These two events are independent, hence we can rewrite the expression as:

$$Prob[(u_1 \in I_1 \wedge u_1 \text{ incompatible for } C_1)].Prob[(u_2 \in I_2 \wedge u_2 \text{ incompatible for } C_2]$$

But $Prob[(u_1 \in I_1 \wedge u_1 \text{ incompatible for } C_1)] \geq (\varepsilon/2).(3.\varepsilon/4.(m+1))$ and similarly for u_2. Hence:

$$Prob[\text{Tester rejects}] \geq (3.\varepsilon^2/8(m+1))^2$$

This decomposition generalizes to $\Pi = C_1....C_l$ by taking cuts for C_{i_1} of weight $\varepsilon.n/l$, cuts for C_{i_2} of weight $\varepsilon.n/l$ until possible cuts for C_l of weight $\varepsilon.n/l$.

Notice that another Tester $T_2'(w, \Pi)$ could be used: given $u_1 < u_2 < ... < u_l$, use $T_1(u_i, C_i)$, i.e. if each u_i is compatible for C_i for $i = 1, ...l$. It would accept of one of test T_1 accepts and reject otherwise.

Tester for a $L(\mathcal{A})$

We can now use the previous Tester T_2 and have a Tester T_3 for $L(\mathcal{A})$. A sequence $\Pi = C_1....C_l$ is essential if the initial state is in C_1 and some initial state is in C_l.

Tester $T_3(w, L(\mathcal{A}))$

For all essential $\Pi = C_1....C_l$ of the automaton:

1. If $T_2(w, \Pi)$ Rejects, Reject,

Accept.

4.3 Testing Graphs

In the context of undirected graphs [7], the distance is the (normalized) Edit Distance on edges: the distance between two graphs on n nodes is the minimal number of edge-insertions and edge-deletions needed to modify one graph into the other one. Let us consider the adjacency matrix model. Therefore, a graph $G = (V, E)$ is said to be ε-close to another graph G', if G is at distance at most εn^2 from G', that is if G differs from G' in at most εn^2 edges.

In several cases, the proof of testability of a graph property on the initial graph is based on a reduction to a graph property on constant size but random subgraphs. This was generalized for every testable graph properties by [8]. The notion of ε-reducibility highlights this idea. For every graph $G = (V, E)$ and integer $k \geq 1$, let Π denote the set of all subsets $\pi \subseteq V$ of size k. Denote by G_π the vertex-induced subgraph of G on π.

Definition 7 *Let $\varepsilon > 0$ be a real, $k \geq 1$ an integer, and ϕ, ψ two graph properties. Then ϕ is* (ε, k)-reducible to ψ *if and only if for every graph G,*

$$\begin{aligned} G \models \phi \quad &\Longrightarrow \quad \forall \pi \in \Pi,\ G_\pi \models \psi, \\ G \not\models_\varepsilon \phi \quad &\Longrightarrow \quad \Pr_{\pi \in \Pi}[G_\pi \not\models \psi] \geq 2/3. \end{aligned}$$

Note that the second implication means that if G is ε-far to all graphs satisfying the property ϕ, then with probability at least $2/3$ a random subgraph on k vertices does not satisfy ψ.

Therefore, in order to distinguish between a graph satisfying ϕ to another one that is far from all graphs satisfying ϕ, we only have to estimate the probability $\Pr_{\pi \in \Pi}[G_\pi \models \psi]$. In the first case, the probability is 1, and in the second it is at most $1/3$. This proves that the following generic test is an ε-tester:

Generic Test(ψ, ε, k)

1. Input: A graph $G = (V, E)$
2. Generate uniformly a random subset $\pi \subseteq V$ of size k
3. Accept if $G_\pi \models \psi$ and reject otherwise

Proposition 1 *If for every $\varepsilon > 0$, there exists k_ε such that ϕ is $(\varepsilon, k_\varepsilon)$-reducible to ψ, then the property ϕ is testable. Moreover, for every $\varepsilon > 0$,* ***Generic Test***$(\psi, \varepsilon, k_\varepsilon)$ *is an ε-tester for ϕ whose query and time complexities are in $(k_\varepsilon)^2$.*

In fact, there is a converse of that result, and for instance we can recast the testability of c-colorability [7, 1] in terms of ε-reducibility. Note that this result is quite surprising since c-colorability is an NP-complete problem for $c \geq 3$.

Theorem 4 ([1]) *For all $c \geq 2$, $\varepsilon > 0$, c-colorability is $(\varepsilon, O((c \ln c)/\varepsilon^2))$-reducible to c-colorability.*

If for every $\varepsilon > 0$, there exists k_ε such that ϕ is $(\varepsilon, k_\varepsilon)$-reducible to ψ, then the property ϕ is testable. Moreover, for every $\varepsilon > 0$, **Generic Test**$(\psi, \varepsilon, k_\varepsilon)$ is an ε-tester for ϕ whose query and time complexities are in $(k_\varepsilon)^2$.

In fact, there is a converse of that result, and for instance we can recast the testability of c-colorability [7, 1] in terms of ε-reducibility. Note that this result is quite surprising since c-colorability is an NP-complete problem for $c \geq 3$.

Theorem 5 ([1]) *For all $c \geq 2$, $\varepsilon > 0$, c-colorability is $(\varepsilon, O((c \ln c)/\varepsilon^2))$-reducible to c-colorability.*

4.4 Testing vs. Learning

There are two related concepts:

- Testing, i.e. approximately deciding a property: in this case, we must decide if an input x satisfies the property or is $\varepsilon-$far from the property,
- Approximating the property: if f is the boolean function associated with the property, finding another boolean function g which is $\varepsilon-$close to f.

Clearly, if we approximate the property, we can also test but the converse is not necessarly true.

4.4.1 Learning a linear Classifier

A Classifier may take data labelled with the name of a class and outputs linear functions which best separate the classes. In the case of two classes, and two dimensional data, we have labelled data $(x_i, y_i, 0)$ if 0 is the label of the first class and $(x_i, y_i, 1)$ if 1 is the label of the second class.

A linear regression finds the linear function $a.x + b.y + c = 0$ which minimizes the global error, defined as the sum of the distances from the points to the line, the Ordinary least squares.

PAC-learning is the generalization to words, graphs and general data.

4.4.2 Learning a Community in a graph

In this case, we don't have labelled data as input. We want to find the dense subgraphs or clusters. It is often called unsupervised learning. The problem is particularly important on streams of edges, when we can not store the entire graph. We will introduce some simple solutions for this problem.

4.5 Exercices

1. A function $f : F_2^n \to F_2$ is a *Dictator function* if $f = x_i$ for some $i \in \{1, 2, ...n\}$, i.e. a special linear function. Construct a Tester for this property.

2. Recall that a function f is k-linear if there is S such that $|S| = k$ and $f(x) = \sum_{i \in S} x_i = \bigoplus_{i \in S} x_i$. A function f is a k-junta if there is S such that $|S| = k$ and $f(x) = f(y)$ whenever $x_i = y_i$ for $i \in S$. Construct a tester for this class of functions and show a lower bound $\Omega(k)$ lower bound for the number of queries.

Chapter 5

Streaming

Streaming algorithms have become very important, as data from many fields can be accessed as a stream: astronomy, medicine, social networks,... We are interested in algorithms which read data step by step and maintain a small memory, if possible constant or poly(log) in the size of the stream.

We first consider a stream s of numerical values $x_i \in \{1, 2, ...n\}$, then a stream of edges $e = (v_i, v_j)$ in a graph $G = (V, E)$ and finally balanced words.

5.1 Moments in a stream of values

Let $U = \{1, 2, ...n\}$ be a set of elements and a data stream $s_m = x_1, ...x_m$ of elements of U. Consider the frequency $f_i \in \{0, 1, 2, ...m\}$ the number of occurrences of the element i in the stream.

The k-th frequency moment $F_k = \sum_{j \in U} f_j^k$. Notice that F_0 is the number of distinct elements in the stream, $F_1 = m$, and as k increases we give more weight to the most frequent element. It is then natural to define $F_\infty = \mathrm{M}ax_j f_j$.

5.1.1 Reservoir Sampling

A classical technique, introduced in [14] is to sample each new element x_n of a stream s with some probability p and to keep it in a set S called the *reservoir* which holds k tuples. It also applies to values with a weight. Let $s = x_1, x_2, ...x_n$ be the stream and let $\widehat{S_n}$ be the reservoir at stage n. We write $\widehat{S}$ to denote that S is a random variable.

k-Reservoir sampling: A(s)

- Initialize $S_k = \{x_1, x_2, ...x_k\}$,
- For $j = k + 1, ...n$, select x_j with probability k/j. If it is selected, replace a random element of the reservoir (with probability $1/k$) by x_j.

Lemma 6 *Let $\widehat{S_n}$ be the reservoir at stage n. Then for all $n > k$ and $1 \leq i \leq n$:*

$$Prob[x_i \in S_n] = k/n]$$

Proof : Let us prove by induction on n. The probability at stage $n+1$ that x_i is in the reservoir $Prob[x_i \in \widehat{S_{n+1}}]$ is composed of two events: either the tuple x_{n+1} does not enter the reservoir, with probability $(1 - k/n + 1)$ or the tuple x_{n+1} enters the reservoir with probability $k/(n + 1)$ and the tuple x_i is maintained with probability $(k - 1)/k$. Hence:

$$Prob[x_i \in \widehat{S_{n+1}}] = k/n((1 - k/(n + 1) + k/(n + 1)\ .(k - 1)/k)$$

$$Prob[x_i \in \widehat{S_{n+1}}] = k/n((n+1-k)/(n+1) + (k-1)/(n+1))$$

$$Prob[x_i \in \widehat{S_{n+1}}] = k/(n+1)$$

Interestingly, imagine we are interested in the most recent data. At every step some values are outdated and leave the reservoir and new values appear. How do we generalize the reservoir into a *Window Reservoir*? Some other techniques based on hashing are possible, but the strict generalization of the classical reservoir is an interesting open problem.

5.1.2 Morris Algorithm for estimating F_1

A counter needs $O(\log n)$ space. Could we use less space? Morris answered this question in 1977, with a surprising technique.

Assume $n = 2^i$, and let us store i. As n increases we increase i with probability $1/2^i$.

Given i (which requires $O(\log \log n)$ space to store), we estimate n as $Z = 2^i$. If $n = 2^j$, then $E[Z] = n$.

5.1.3 Estimating F_0

Let h be a random function $h : \{1, 2, ...M\} \to \{1, 2, ...M\}$.

Algorithm $A_2(s_m)$:

- Let h be a random function.
- $Min = h(x_1)$
- At every step, reading x_i, let $Min = MIN\{Min, h(x_i))\}$

Return $X = M/Min$

Intuitively, the expected value if $n = 1$ of Min is $M/2$ hence $X = 2$. If $n = 2$, then the expected value of Min is $M/3$ hence $X = 3$. As n increases the estimator is essentially unbiased.

MinHash is a variation to estimate the Jaccard Similarity between two streams. Assume A is the domain of the first stream and B is the domain of the second stream. Let $J(A, B) = |A \cap B|/[|A \cup B|$ the Jaccard Similarity. It is the probability that $X(A) = X(B)$. It can be approximated by the rate of common minimum values. Keep the 100 smallest values of $h(x_i)$ in both streams A and B. If there are 10 values which are in common, then $J(A, B)$ is approximately 10%.

Count-MinSketch [**?**] is a generalization to approximate the different frequencies. Consider a matrix $A(i, j)$ with w columns and d rows. Each row i is associated with hash function $h_i : U = \{1, 2, ...n\} \to \{1, 2, ...d\}$. Every time we read x_k, we compute

$$h_1(x_k), h_2(x_k), ...h_w(x_k)$$

and we increase each value $A(i, h_i(x_k))$ by 1. If we want the frequency of x_k, we approximate it by the $Min_i\{A(i, h_i(x_k))\}$. If we take $w = 3/\varepsilon$ and $d = \log 1/\delta$, we approximate the frequency within an additive factor ε with probability $1 - \delta$.

5.1.4 Basic estimator for F_2

The approach is to find an estimator whose expectation is F_2 and whose variance is bounded.

- Le h: $U \to \{-1, +1\}$ be a function which assigns a random sign to a an element of U.
- $Z = 0$. For each j in the stream, $Z = Z + h(j)$.
- Return $X = Z^2$

$Z = \sum_j f_j.h(j)$.

Analysis

Lemma 7 $I\!E[X] = F_2$

Proof :

$$I\!E[X] = I\!E[Z^2]$$
$$I\!E[X] = I\!E[(\sum_j f_j.h(j))^2]$$
$$I\!E[X] = E[\sum_j f_j^2 + 2\sum_{j<l} f_j.f_l]$$
$$I\!E[X] = F_2 + \sum_{j<l} E[\sum_{j<l} f_j.f_l]$$
$$I\!E[X] = F_2$$

Let $Y = \frac{\sum_t X_t}{t}$, i.e. the average of t independent trials.

Lemma 8 $Var[Y] = \frac{Var[X]}{t}$

Proof :

$$I\!E[X] = I\!E[Z^2]$$

Lemma 9 $I\!E[Var[X]] \leq 2.F_2^2$

Proof :

$$Var[X] = I\!E[X^2] - (I\!E[X])^2$$
$$I\!E[X^2] = I\!E[(\sum_j f_j.h(j))^4]$$

Most of the terms $h(j_1).h(j_1).h(j_1).h(j_1)$ have a 0 expectation. Hence:

$$I\!E[X^2] = I\!E[(\sum_j f_j.h(j))^4] = I\!E[\sum_j f_j^4.h(j)^4 + 6\sum_{j<l} f_j^2.f_l^2.h(j))^2.h(l)^2]$$
$$I\!E[X^2] = \sum_j f_j^4 + 6\sum_{j<l} f_j^2.f_l^2$$
$$F_2^2 = \sum_j f_j^4 + 2\sum_{j<l} f_j^2.f_l^2$$

Hence $I\!E[X^2] \leq 3.F_2^2$.

We can then guarantee the quality of the approximation of Y.

Lemma 10 *If* $t = \frac{2}{\varepsilon^2.\delta}$, *then*

$$Prob[Y \in [(1-\varepsilon).F_2, (1+\varepsilon).F_2]] \geq 1 - \delta$$

Proof : Recall Chebyshev's inequality: $Prob[|Y - F_2| \geq c] \leq \frac{Var(Y)}{c^2}$.
In our case $c = \varepsilon.F_2$, and using lemma 8.3, we get:
$Prob[|Y - F_2| \geq \varepsilon.F_2] \leq \frac{Var(Y)}{(\varepsilon.F_2)^2} \leq \frac{2.F_2^2}{t.(\varepsilon.F_2)^2}$ Hence $t = \frac{2}{\varepsilon^2.\delta}$.

5.2 Graph properties from a stream of edges

Let $G(V, E)$ be a symmetric graph with n vertices and m edges. Let d_i the degree of node i.

5.2.1 Graph properties by sampling [?]

Consider a connected graph and an ergodic random walk. It has a stationary distribution π. For this distribution $Prob[x = v_i] = d_i/D$ where d_i is the degree of the node v_i. Suppose we take r nodes with the distribution π: $\{x_1, x_2, ...x_r\}$. Let $Y_{j,j'} = Y_{j,j'} = 1$ if $x_j = x_{j'}$ and 0 otherwise.

Consider the following variables which only depend on the samples. Let:

$$\psi_1 = \sum_{i=1,...r} d_i$$

$$\psi_{-1} = \sum_{i=1,...r} 1/d_i$$

$$C = \sum_{j \neq j'} Y_{j,j'}$$

The variable C measures the number of collisions. Let $R = \psi_1.\psi_{-1} - r$.

Algorithm to estimate n. Maintain R and C. Output $\widehat{n} = R/C$.

Lemma 11 *If $r \geq f(\varepsilon, \delta)$, then $Prob[\widehat{n} \in [(1-\varepsilon).n, (1+\varepsilon).n] \geq 1 - \delta$.*

Proof : Let us estimate $I\!E[C]$ and $I\!E[R]$:

$$I\!E[R] = I\!E[\psi_1.\psi_{-1} - r] = I\!E[\sum_{i=1,...r} d_i. \sum_{i=1,...r} \frac{1}{d_i} - r] = I\!E[\sum_{i \neq j} \frac{d_i}{d_j}]$$

Recall Chebyshev's inequality: $Prob[|R - I\!E[R]| \geq c] \leq \frac{Var(R)}{c^2}$.

5.2.2 Graph properties in a stream

Consider a stream of edges, defining at any time t a graph G_t. Notice that we can sample edges uniformly, with a reservoir sampling and obtain a distribution of nodes where each node has a probability to be chosen proportional to its degree.

Uniform Spanning trees

The problem of generating uniform Spanning Trees has beed studied by Aldous and Broder [**?**] and recent studies on the approximation of the effective resistance of edges [**?**] have renewed some interest in the problem. Given an edge $e \in E(G)$ the graph G/e is obtained by contracting the edge e while $G \backslash e$ is obtained by deleting the edge e.

The output of a randomized algorithm A is denoted by $A(G)$. Given a spanning tree T and a cycle C, the cycle obtained by adding the edge e to T is denoted by $C(e, T)$. The length of the cycle is denoted as $|C(e, T)|$. The number of spanning trees in G is denoted by $Z(G)$. The effective resistance for edge $e \in G$ is denoted as $R(e)$. It is well known that if T is a uniform spanning tree for G then $Pr[e \in T] = R(e)$. $Pr[e \in T] = R(e)$. The sum of effective resistances $\sum_e R(e) = n - 1$. We use this well known identity:

$$Z(G).(1 - R_e) = Z(G \backslash e)$$

Streaming Algorithm to construct a spanning tree T. Keep a spanning tree T at any time. When a new edge e appears in the stream, detect if there is a cycle C in $T + e$. If there is no cycle, add e to T, otherwise break the cycle by removing a uniform edge of C, i.e. an edge of C with probability $1/|C|$.

Let π a permutation of the edges. It is easy to see that the distribution of the spanning trees T may not be uniform for a given π. We conjecture that it is close to the uniform distribution if we choose a uniform π, i.e.

$$Prob_{\pi,r}[T] \simeq 1/Z(G)$$

when r are the random choices of the Streaming Algorithm.

Chapter 6

Social Networks

Consider the following graphs:

- The nodes are the names of the characters in the novel *Les Misérables*. Two nodes u, v are adjacent if the the names occur in the same sentence. For example, the sentence *Comment se faisait-il que l'existence de Jean Valjean eût coudoyé si longtemps celle de Cosette?* generates an edge (JeanValjean, Cosette).
- The nodes are authors of scientific publications, the edges link two authors who co-publish the same paper. For a publication with 3 authors, we generate 6 edges.
- The Facebook graph: the nodes are the users and an edge represents the friendship symmetric relation.
- The Twitter user graph: the nodes are the users and an oriented edge (u, v) exists if v follows u.
- The Twitter content graph: the nodes are the tags ($@x$ or $\#y$) and each tweet sent from $@x$ which contains $\#y, \#z, @t$ generates 3 edges: $(@x, \#y)$, $(@x, \#z)$ and $(@x, @t)$. If a tweet has 10 tags, it generates 10 edges.

These graphs have very specific structures and in particular:

- The degree distribution is a power law,
- The diameter is small (usually less than 6)
- There are dense subgraphs, called clusters or communities.
- They evolve in time

The main questions concern the information we can extract from these dynamic graphs.

- Can we quantify the importance of a node? The Pagerank answers this question.
- Which nodes are central? The closeness centrality is a possible answer.
- Which clusters appear in the graph? The Community detection algorithms approximate the clusters.
- How do these parameters evolve in time?

There are several classical techniques if the graph is entirely known. A more challenging question is to cluster the graphs when we read a stream of edges and do not store the entire graphs . Even more challenging, is to cluster the dynamic graphs defined by the most recent edges.

6.1 Pagerank

Given a graph, we assign a probability measure to each node, in such a way that a random walk of a certain length has the probability to reach the node i. The walk follows the edges of the graph with a uniform distribution. When it reaches a sink, i.e. a node without outgoing edges, it chooses a new node with a uniform distribution.

The Pagerank algorithm approximates this distribution, in a distributed way. There are many variations, depending on the length of the walk, the number of distributed walks and the way to aggregate the results. It is the technology which allowed Google to dominate the search industry in 1998.

6.2 Clusters of a given graph

There are several definitions of a cluster or community or dense subgraph. We consider a cluster of domain S as a *maximal dense subgraph* which depends on a parameter γ. Let $\gamma \leq 1$ and let $E(S)$ be the multiset of internal edges i.e. edges $e = (u, v)$ where $u, v \in S$. A γ-cluster is a maximal subset S such that $E(S) \geq \gamma.|S|^2$.

Consider the symmetric graph of the Figure 6.1 and its adjancency matrix A:

$$A = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

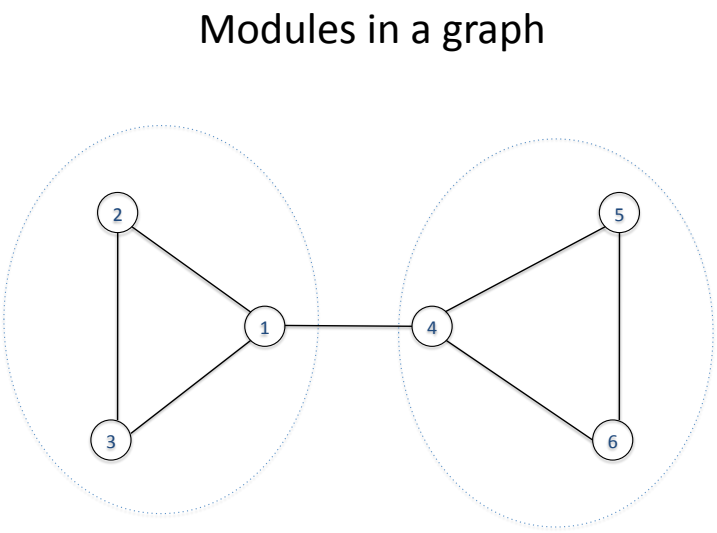


Figure 6.1: Modular decomposition of a graph

6.2.1 Basics of linear algebra

Let us recall a few important ideas:

- The rank of a matrix is the largest set of independent rows. A row is dependent if it is a linear combination of the other rows.
- Eigenvalues and eigenvectors.

Given a square matrix A, we consider a vector v and a scalar λ such that:

$$A.v = \lambda.v$$

Such a value λ is called an *eigenvalue* and the vector v is called an *eigenvector*. One method to compute the eigenvalues is to express the characteristic polynomial of A, a polynomial in λ of degree n:

$$det(A - \lambda.I) = 0$$

where I is the identity matrix. The *eigenvalues* are complex values (in C) but for a large class of matrices, called Positive Semi Definite, the values are reals and positive. Covariance matrices belong to this class.

The Python code, assuming an object *la* of the " linalg" library (linear algebra) is:

```
w,v=la.eig(A)
```

The vector w gives the n eigenvalues. The matrix v gives the n eigenvectors represented as columns. For example:

$$A = \begin{pmatrix} 2 & 1 \\ 0 & 1 \end{pmatrix}$$

The characteristic polynomial is $\lambda^2 - 3.\lambda + 2 = (\lambda - 2).(\lambda - 1)$. The roots are real and are 1 and 2. To find the eigenvector associated to $\lambda = 2$, we write:

$$A = \begin{pmatrix} 2 & 1 \\ 0 & 1 \end{pmatrix} . \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 2x \\ 2y \end{pmatrix}$$

We are looking for vectors and we can always fix one of the coordinates. The equations are:

$$2x + y = 2x$$

$$y = 2y$$

The second equation implies that $y = 0$, the first equation implies x is any value, for example 1. The eigenvector is

$$v = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

Similarly the equations for $\lambda = 1$ are:

$$2x + y = x$$

$$y = y$$

The second eigenvector if we choose $y = 1$ is:

$$v' = \begin{pmatrix} -1 \\ 1 \end{pmatrix}$$

The python code to compute these eigenvectors and eigenvalues is:

```
import numpy as np
import scipy.linalg as la

A= np.array([ [2,1],[0,1]])


w,v=la.eig(A)

print("Eigenvalues of A",w)
print("Eigenvectors of A \n",v)
-----------------------------------------
Eigenvalues of A [ 2.+0.j  1.+0.j]
Eigenvectors of A
 [[ 1.         -0.70710678]
 [ 0.          0.70710678]]
```

Notice that the second eigenvector is only proportional to the vector $v' = \begin{pmatrix} -1 \\ 1 \end{pmatrix}$. The length of v' is 2 whereas the length of $\begin{pmatrix} -0.707 \\ 0.707 \end{pmatrix}$ is $\sqrt{2}$.

If the matrix is symmetric, the eigenvectors are orthogonal.

```
import numpy as np
import scipy.linalg as la

A= np.array([ [2,1],[1,1]])


w,v=la.eig(A)

print("Eigenvalues of A",w)
print("Eigenvectors of A \n",v)
-----------------------------------------
Eigenvalues of A [ 2.61803399+0.j  0.38196601+0.j]
Eigenvectors of A
 [[ 0.85065081 -0.52573111]
 [ 0.52573111  0.85065081]]
```

6.2.2 Spectral methods

The Laplacian matrix $L = D - A$ where is the diagonal matrix where $D(i,i) = degree(i)$.

$$A = \begin{pmatrix} 3 & -1 & -1 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 \\ -1 & -1 & 2 & 0 & 0 & 0 \\ -1 & 0 & 0 & 3 & -1 & -1 \\ 0 & 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & -1 & 2 \end{pmatrix}$$

The Laplacian is a Positive Semidefinite Matrix (PSD), and in particular its eigenvalues are real and positive. In this example, the eigenvalues are:

$$4.56155281e+00+0.j, -5.83070714e-16+0.j, 4.38447187e-01+0.j,$$

$$3.00000000e+00+0.j, 3.00000000e+00+0.j, 3.00000000e+00+0.j$$

Notice that the smallest eigenvalue λ_1 is 0, and the precision parameter makes it only close to 0. The second eigenvalue is $4.38447187e-01+0.j$ is important, and in particular the eigenvector. The eigenvectors are the columns of the matrix:

$$A = \begin{pmatrix} -0.6571923 & 0.40824829 & 0.26095647 & -0.57735027 & -0.30464858 & 0.1005645 \\ 0.18452409 & 0.40824829 & 0.46470513 & 0.28867513 & -0.42497055 & 0.07701434 \\ 0.18452409 & 0.40824829 & 0.46470513 & 0.28867513 & 0.72961912 & -0.17757884 \\ 0.6571923 & 0.40824829 & -0.26095647 & -0.57735027 & -0.30464858 & 0.1005645 \\ -0.18452409 & 0.40824829 & -0.46470513 & 0.28867513 & 0.31819937 & -0.7348447 \\ -0.18452409 & 0.40824829 & -0.46470513 & 0.28867513 & -0.01355079 & 0.6342802 \end{pmatrix}$$

The second eigenvector corresponding to λ_1 is the unit vector. The Laplacian is doubly stochastic, i.e. the sums of the line values or the sums of the column values is always 0. The eigenvector for λ_2 is

$$v_2 = \begin{pmatrix} 0.26095647 \\ 0.46470513 \\ 0.46470513 \\ -0.26095647 \\ -0.46470513 \\ -0.46470513 \end{pmatrix}$$

Notice that it splits the nodes into two parts according to the vector signs, finding the cut between the two clusters. It is a general phenomenon.

6.2.3 Modules via the modularity matrix

The degree vector is d where $d(i)$ is the degree of the node i.

$$d = \begin{pmatrix} 3 \\ 2 \\ 2 \\ 3 \\ 2 \\ 2 \end{pmatrix}$$

Let $B = A - d.d^t/2m$ be the modularity matrix:

$$B = \begin{pmatrix} -0.64285714 & 0.57142857 & 0.57142857 & 0.35714286 & -0.42857143 & -0.42857143 \\ 0.57142857 & -0.28571429 & 0.71428571 & -0.42857143 & -0.28571429 & -0.28571429 \\ 0.57142857 & 0.71428571 & -0.28571429 & -0.42857143 & -0.28571429 & -0.28571429 \\ 0.35714286 & -0.42857143 & -0.42857143 & -0.64285714 & 0.57142857 & 0.57142857 \\ -0.42857143 & -0.28571429 & -0.28571429 & 0.57142857 & -0.28571429 & 0.71428571 \\ -0.42857143 & -0.28571429 & -0.28571429 & 0.57142857 & 0.71428571 & -0.28571429 \end{pmatrix}$$

The eigenvalues and eigenvectors are:

$$1.73205081e+00+0.j - 1.73205081e+00+0.j 1.03359129e-16+0.j$$

$$-4.28571429e-01+0.j - 1.00000000e+00+0.j - 1.00000000e+00+0.j$$

$$\begin{pmatrix} 3.25057584e^{-01} & 6.27963030e^{-01} & -4.08248290e^{-01} & 5.77350269e^{-01} & 1.94704069e^{-17} & 1.20649538e^{-17} \\ 4.44036917e^{-01} & -2.29850422e^{-01} & -4.08248290e^{-01} & -2.88675135e^{-01} & -5.43451954e^{-01} & 3.07079803e^{-01} \\ 4.44036917e^{-01} & -2.29850422e^{-01} & -4.08248290e^{-01} & -2.88675135e^{-01} & 5.43451954e^{-01} & -3.07079803e^{-01} \\ -3.25057584e^{-01} & -6.27963030e^{-01} & -4.08248290e^{-01} & 5.77350269e^{-01} & -8.44215958e^{-17} & 7.96494760e^{-17} \\ -4.44036917e^{-01} & 2.29850422e^{-01} & -4.08248290e^{-01} & -2.88675135e^{-01} & -4.52393604e^{-01} & -6.36947403e^{-01} \\ -4.44036917e^{-01} & 2.29850422e^{-01} & -4.08248290e^{-01} & -2.88675135e^{-01} & 4.52393604e^{-01} & 6.36947403e^{-01} \end{pmatrix}$$

The first eigenvector corresponding to the only positive eigenvalues indicates the best cut.

6.3 Clusters in a stream of edges

A window can be defined by a time interval $[t, t+\lambda]$, of length λ. All the edges which appear in this interval define G_t. One may view the sequence $t = \tau, 2.\tau, 3.\tau.....$, i.e. a discrete view of G_t. We will apply this analysis to the Twitter content graph. You give some tags such as #bitcoin, #xrp, #cnn,.... and Twitter gives you the stream of tweets which contains at least one tag of your list. You convert the tweets into edges and have a stream of edges. Typically you will collect approximately 2.10^4 edges every hour, 48.10^4 every day and $30.48.10^4 = 0.15.10^8$ edges in a month. If you follow 7 streams in parallel, you collect approximately 10^8 edges, i.e. bigdata.

6.4 Random graphs

The classical Erdös-Renyi model $G(n,p)$ [6], generates random graphs with n nodes and edges are taken independently with probability p where $0 < p < 1$. The degree distribution is close to a gaussian centered on $n.p$. Most of the social graphs have a degree distribution close to a power law (such as a Zipfian distribution distribution where $Prob[d = j] = c/j^2$). The Preferential Attachment or the Configuration Model [11] provide models where the degree distribution follows such a power law. In the Configuration Model, the degree distribution can be an arbitrary distribution $\mathcal{D}$: given n nodes, we fix $d_1, d_2, ...d_{max}$ as the number of nodes of degree $1, 2, ...$ where d_{max} is the maximum degree. In order to generate a random graph with n nodes and the degree distribution $\mathcal{D}$, we enumerate each node u with d half-edges (stubs) and takes a symmetric random matching π between two stubs, for example with a uniform permutation such that $\pi(i) \neq i$ and $\pi(i) = j$ iff $\pi(j) = i$. Then all the possible graphs are obtained with a distribution close to the uniform distribution.

In the case of a Zipfian degree distribution, the number of nodes of degree j is $n.c/j^2$ which is less than 1 if $j > \sqrt{c.n}$ hence $d_{max} = O(\sqrt{n})$. We have $c.n$ nodes of degree 1, $c.n/4$ nodes of degree 2, $c.n/9$ nodes of degree 3 and so on. There are several other possible models of random graphs with a power law degree distribution, in particular that the probability for a node to be of degree j is asymptotically c/j^2.

A classical study is to find sufficient conditions so that the random graph has a *giant component*, i.e. of size $O(n)$ for a graph of size n. In the Erdös-Renyi model $G(n,p)$, it requires that $p > 1/n$, and in the Configuration Model $I\!E[\mathcal{D}^2] - 2I\!E[\mathcal{D}] > 0$ which is realized for the Zipfian distribution. There is a phase transition for both models.

6.4.1 Random graphs with a power law degree distribution and a cluster

Fix some a subset S among the nodes of high degree and consider a preferential matching for S. A stub is in S if its origin or extremity is in S. With probability 80%, match the stubs in S uniformly in S and with probability 20%, match the stubs uniformly in $V - S$. As we pick a random stub, it is in S or in $V - S$ The edges concentrate in S and create a γ-cluster.

As an example, let S be a subset of size $\sqrt{n}/2$ nodes among the nodes of degree $\sqrt{n}, \sqrt{n}-1, ...\sqrt{n}/2$. We have a γ-cluster with $\gamma.n/4$ edges.

6.5 Dynamic Random graphs

There are several possible extensions to dynamic random graphs. In our model, the Dynamics is exogenous and at any time chooses between the Uniform and the Concentrated Dynamics.

6.5.1 Uniform Dynamics

We generalize the Configuration Model in a dynamic setting. Consider the following **Uniform Dynamics**: remove $q \geq 2$ random edges, uniformly on the set of edges of G, freeing $2.q$ stubs. Generate a new uniform matching on these hubs to obtain G'. The distribution of random graphs is uniform.

6.5.2 Concentrated Dynamics

A typical graph generated by the Uniform Dynamics is not likely to have a large cluster. Consider the **S-concentrated Dynamics**, starting from an arbitrary graph G: fix some a subset S among the nodes of high degree. Remove $q \geq 2$ edges, uniformly on the set of edges of G, freeing $2.q$ stubs, as before. A stub is in S if its origin or extremity is in S. With probability 80%, match the stubs in S uniformly in S. With probability 20%, match the stubs in S uniformly in $V - S$. This dynamics will concentrate edges in S and will create a γ-cluster after a few iterations with high probability, assuming the degree distribution is a power law. The distribution of graphs with a γ-cluster is also uniform.

6.5.3 General Dynamics

a general Dynamics is a function which chooses at any given time, one of the two strategies: either a Uniform Dynamics or some S-concentrated Dynamics for a fixed S. An example is the **Step Dynamics**: apply the Uniform Dynamics first, then switch to the S-dynamics for a time period Δ, and switch back to the Uniform Dynamics. In our setting, the Dynamics depends on some external information, which we try to approximately recover. Notice that during the Uniform Dynamics phase, there are no large components and we store nothing. For the step phase, we store some components which will approximate S. More complex strategies could involve several clusters S_1 and S_2 which mayor may not intersect. One may have a Δ_1 step on S_1 and then another Δ_2 step on S_2, which may or may not overlap.

6.5.4 Stream of edges

Let $e_1, e_2, ...e_i...$ be a stream of edges where each $e_i = (u, v)$. It defines a graph $G = (V, E)$ where V is the set of nodes and $E \subseteq V^2$ is the set of edges: we allow self-loops and multi edges and assume that the graph is symmetric. In the *window model* we isolate the most recent edges at some discrete $t_1, t_2, ...$ We fix the length of the window τ, hence $t_1 = \tau$ and each $t_i = \tau + \lambda.(i-1)$ for $i > 1$ and $\lambda < \tau$ determines a window of length τ and a graph G_i defined by the edges in the window or time interval $[t_i - \tau, t_i]$. We therefore generate a sequence of graphs $G_1, G_2, ...$ at times $t_1, t_2, ...$ and write $G(t)$ for this sequence. The graphs G_{i+1} and G_i share many edges: old edges of G_i are removed and new edges are added to G_{i+1}. Social graphs have a specific structure, a specific degree distribution (power law), a small diameter and some dense clusters. The dynamic random graphs introduced in the next section satisfy these conditions.

6.6 Deciding properties

We first consider a property on static graphs and then on dynamic graphs. Let R be the Reservoir of size k after we read m edges $e_1, e_2, ...e_m$. In this simple case, we first fill the Reservoir with $e_1, e_2, ...e_k$. For $i > k$, we decide to keep e_i with probability k/i and if we keep e_i, we remove one of the edges (with probability $1/k$) to make room for e_i. Each edge e_i has then probability k/m to be in the Reservoir, i.e. uniform.

The probabilistic space Ω is determined by the choices taken at every step by the Reservoir sampling. Consider a clique S in the graph: its image in the Reservoir is the set G_S of internal edges $e = (u, v)$ in the Reservoir, where $u, v \in S$. Each edge of the clique S is selected with constant probability k/m, so we are in the case of the Erdös-Renyi model $G(n, p)$ where $n = |S|$ and $p = k/m$. We know that the phase transition occurs at $p = 1/n$, i.e. there is a giant component if $p > 1/n$ and the graph is connected if $p \geq \log n/n$.

In the case of a γ-clusters S associated with the S-concentrated Dynamics, the phase transition occurs at $p = 1/\gamma.n$. Let V_S be the set of nodes of the giant component G_S whose nodes are in S. As it is customary for approximate algorithms, we write $Prob_\Omega[\text{Condition}] \geq 1 - \delta$ to say that the Condition is true with high probability.

Lemma 12 *For m large enough, there exists $\alpha = O(\log n)$ such that if $|S| \geq m/\gamma.k$, then $Prob_\Omega[|V_S| > \alpha] \geq 1 - \delta$.*

Proof : If S is almost a clique, i.e. a γ-cluster, then the phase transition occurs at $p = 1/\gamma.|S|$. Hence if $p > 1/\gamma.|S|$, there is a giant component of size larger than a constant times $|S|$, say $|S|/2$ with high probability $1 - \delta$. As the probability of the edges is k/m, it occurs if $|S| \geq m/\gamma.k$. Hence for m large enough, there exists $\alpha = O(\log n)$ such that $Prob_\Omega[|V_S| > \alpha] \geq 1 - \delta$.

In order to decide the graph property P: *there is a large γ-cluster*, consider this simple algorithm.

Static Cluster detection Algorithm 1: let C be the largest connected component of the Reservoir R. If $|C| \geq \alpha$ then Accept, else Reject.

Theorem 6 *If $|S| \geq m/\gamma.k$ in the S-concentrated Dynamics, then $Prob_\Omega[$Algorithm 1 Accepts$] \geq 1 - \delta$ and for the Uniform Dynamics $Prob_\Omega[$Algorithm 1 Rejects$] \geq 1 - \delta$.*

Proof : If $|S| \geq m/\gamma.k$ for the concentrated Dynamics, Lemma 12 states that $|V_S| > \alpha$ with high probability, hence as $V_S \subseteq C$, the condition $|C| \geq \alpha$ is true with high probability hence $Prob_\Omega[$Algorithm 1 Accepts$] \geq 1 - \delta$. For the Uniform Dynamics ($|S| = 0$), [9] shows that the largest connected component has size $O(\log n)$. Hence $Prob_\Omega[$Algorithm 1 Rejects$] \geq 1 - \delta$.

Notice that $m = c_1.n.\log n$, as the average degree in a power law is $c_1.\log n$. If $k = \sqrt{c_1.n}.\log n$ and $|S| \geq m/\gamma.k = \sqrt{c_1.n}/\gamma$, it satisfies the condition and it can be realized with the nodes of high degree.

6.6.1 Deciding a dynamic property: $\Diamond\ P$

Let P be the previous property: is there a γ-cluster? How do we decide $\Diamond\ P$? Consider the step strategy of length $\Delta > \tau$. When we switch strategy at time t_1 there is a delay until S is a γ-cluster and symmetrically the same delay when we switch again at time $t_2 > t_1$. The probabilistic space Ω_t is now much larger.

Dynamic Cluster detection Algorithm 2: let C_i be the largest connected component of a dynamic Reservoir R_i at time t_i. If there is an i such that $|C_i| \geq \alpha$, then Accept, else Reject.

We can still distinguish between the Uniform and the S-concentrated Dynamics, if S is large enough. Let $m(t)$ be the number of edges in the window at time t. Let $G(t)$ be a graph defined by a stream of $m(t)$ edges following a power law $\mathcal{D}$.

Theorem 7 *For the step Dynamics on length Δ and $t > t_2$, if $|S| \geq m(t)/\gamma.k$ and $m(t)$ large enough, then $Prob_\Omega[A_2$ Accepts$] \geq 1 - \delta^{\Delta/\tau}$ For the Uniform Dynamics $Prob_\Omega[A_2$ Rejects$] \geq (1 - \delta)^{\Delta/\lambda}$.*

Proof : For each window, we can apply theorem 6 and there are Δ/τ independent windows. If $|S| \geq m/\gamma.k$ for the concentrated Dynamics, the error probability is smaller than the error made for Δ/τ independent windows, which is $\delta^{t/\tau}$. Hence $Prob_\Omega[$Algorithm 2 Accepts$] \geq 1 - \delta^{\delta/\lambda}$. For the Uniform Dynamics (equivalent to $|S| = 0$), the algorithm

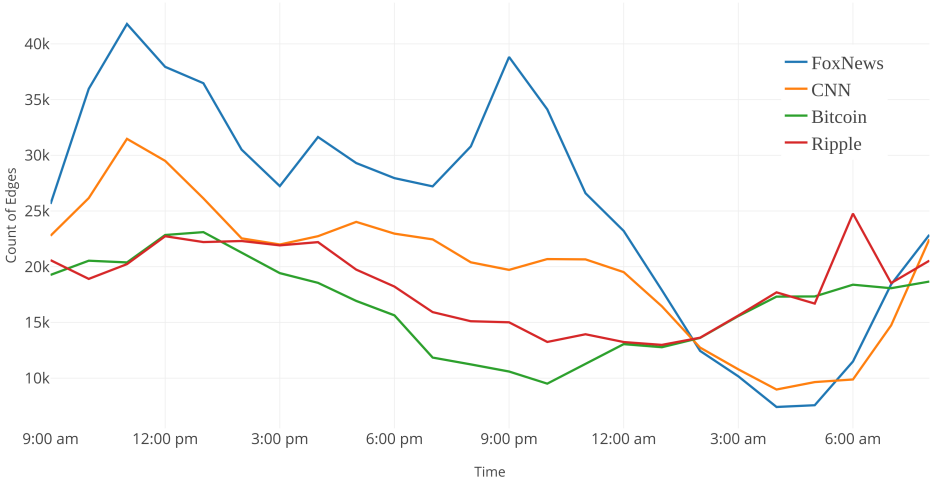


Figure 6.2: Number of edges in $1h$ windows, for 4 streams during $24h$

needs to be correct at each δ/λ step. Hence $Prob_\Omega[\text{Algorithm 2 Rejects}] \geq (1-\delta)^{\delta/\lambda}$.

The probability to accept for the S concentrated Dynamics is amplified whereas the probability to reject for the Uniform Dynamics decreases. One single error generates a global error. Clearly, we could also estimate Δ, for step strategies with similar techniques.

6.6.2 Correlation between two streams

Suppose we have two streams $G_1(t)$ and $G_2(t)$ which share the same clock. Suppose that $G_1(t)$ is a step strategy Δ_1 on a cluster S_1 and $G_2(t)$ is a step strategy Δ_2 on a cluster S_2. Let $\rho^* = J(S_1, S_2)$. How good is the estimation of their correlation? Let $C_i(t) = \bigcup_j C_{i,j}$ be the set of large clusters $C_{i,j}$ at time t_j of the graph G_i, for $i = 1$ or 2 at time t. Consider the following online algorithm to compute $\rho(t)$:

Online Algorithm 3 for $\rho(t)$. At time $t+\lambda$, compute the increase δ_i in size of $C_i(t+\lambda)$ for $i = 1, 2$ from $C_i(t)$, and δ' the increase in size of $C_1(t+\lambda) \cap C_2(t+\lambda)$. Suppose $\rho(t) = I/U$ where $I = |C_1(t) \cap C_2(t)|$ and $U = |C_1(t) \cup C_2(t)|$. Then: $\rho(t+\lambda) = \rho(t) + \frac{U.\delta' - I.(\delta_1+\delta_2)}{U.(U+\delta_1+\delta_2)}$.

A simple computation shows that $\rho(t+\lambda) = \frac{I+\delta'}{U+\delta_1+\delta_2}$, i.e. the correct definition.

Theorem 8 *Let $G_1(t)$ and $G_2(t)$ be two step strategies before time t on two clusters such that $|S_i| \geq m/\gamma.k$ for $i = 1, 2$. Then $Prob_{\Omega_t}[|\rho(t) - \rho^*| \leq \varepsilon] \geq 1 - \delta$.*

Proof : After the first observed step, for example on S_1, Lemma 12 indicates that V_{S_1} is already some approximation of S_1. After Δ_1/τ independent trials, $V_1 = \bigcup_i V_{S_{1,i}}$ will be a good approximation of S_1. Similarly for S_2 and therefore $\rho(t) = J(V_1, V_2)$ will (ε, δ) approximate ρ^*.

6.7 Twitter streams

Given a set of tags such as #CNN or #Bitcoin, Twitter provides a stream of tweets represented as Json trees whose content contains at least one of these tags. The *Twitter Graph* of the stream, is the graph $G = (V, E)$ with multiple edges E where V is the set of tags $\#x$ or $@y$ seen and for each tweet sent by $@y$ which contains

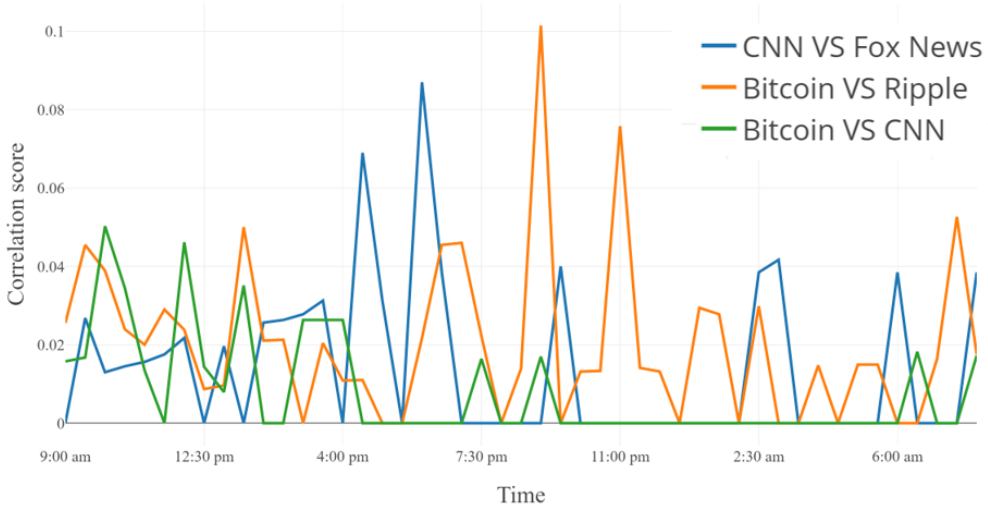


Figure 6.3: Online content correlation for 24h

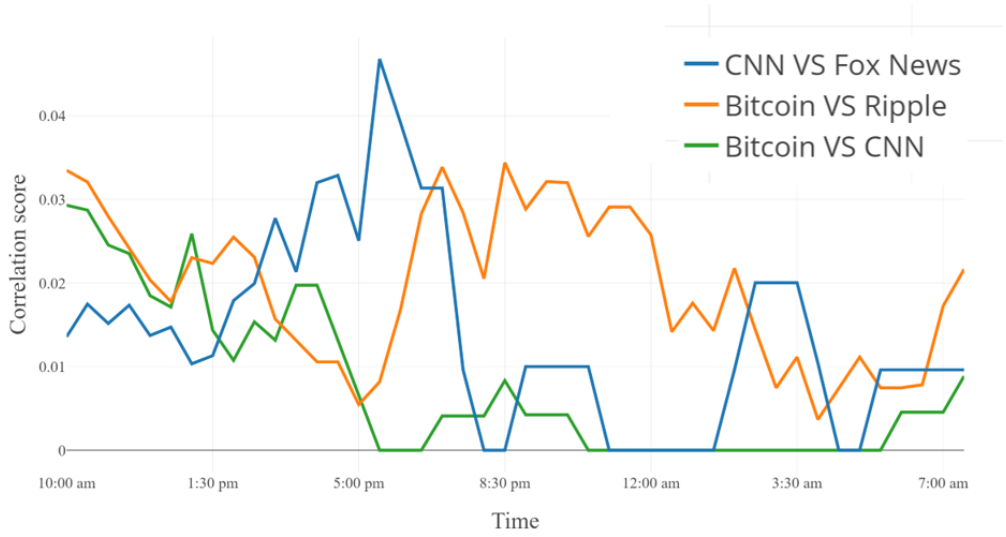


Figure 6.4: Online averaged content correlation for 24h

tags $\#x$,$@z$ we construct the edges $(@y, \#x)$ and $(@y, @z)$ in E. The URL's which appear in the tweet can also be considered as nodes but we ignore them for simplicity. A stream of tweets is then transformed into a stream of edges $e_1, ...e_m, ...$

We simultaneously captured 4 twitter streams[1] on the tags #CNN, #FoxNews, #Bitcoin, and #Xrp (Ripple) during 24 hours with a window size of $\tau = 1h$ and a time interval $\lambda = 30$mins, using a standard PC. Figure 6.2 indicates the number of edges seen in a window, approximately $m = 20.10^3$ per stream, on 48

[1]Using a program available on https://github.com/twitterUP2/stream which takes some tags, a Reservoir size k, a window size τ, a step λ and saves the large connected components of the Reservoirs R_t.

points. For 24 independent windows, we read approximately 48.10^4 edges, and globally approximately 2.10^6 edges. The Reservoirs size $k = 400$ and on the average we save 100 nodes and edges, i.e. $4.48.100 \simeq 2.10^4$ edges, i.e. a compression of 100. For $\gamma = 0.8$, the minimum size of a cluster is $m/\gamma.k \simeq 60$. Notice that k is close to $\sqrt{m}$.

Figure 6.3 gives the three mains correlations $\rho(t)$ out of the possible 6 and the averaged correlation

$$\rho'(t) = (\rho(t-1) + \rho(t) + \rho(t+1))/3$$

The correlation is highly discontinuous, as it can be expected, but the averaged version is smooth. The maximum value is 1% for the correlation and 0.5% for the averaged version. It is always small as witnessed by the correlation matrix. We experienced very small changes in the correlations and $\rho'(t)$, also computed online, witnessed it. The spectrum of the Reservoirs, i.e. the sizes of the large connected components is another interesting indicator. For the #Bitcoin stream, there is a unique very large component.

6.8 Search by correlation

We stored the large clusters for each stream, i.e. the set of nodes of the clusters. Given a search query defined by a set of tags, the answer to a query is the set of the most correlated tags, among the stored tags. From the correlation matrix of the streams, we infer a phylogeny tree. We first extend the correlation between a tag and a set of tags. We can then compute the tags with the highest correlations, as the answer to the query.

Chapter 7

Dimension reduction

Suppose we have n points in R^d, where both n and d are large. We can evaluate a property or make a prediction: can we just analyze a much smaller set of values? There are several approaches:

- Reduce the columns. Instead of $d = 10^4$ columns, maybe 200 are sufficient.
- Reduce the rows. This is sometimes called a coreset.
- Reduce both the rows and the columns.

We first study a general result, stating that we can always reduce the number of columns, without changing too much the distances between the points. We then study the practical version called *principal component analysis* or PCA.

7.1 The fundamental result

Consider d independent gaussian $N(0,1)$ variables $X_1, ...X_d$. Let $|X| = \sqrt{X_1^2 + ...X_d^2}$ be the norm of X, and let Y be $Y = \frac{(X_1,...X_d)}{|X|}$. Observe that Y is on the sphere of dimension d.

Theorem 9 *[Johnson-Lindenstrauss theorem] For any ε and any n, let*

$$k \geq 4.\log n/\varepsilon^2$$

Then for any set V of n points in R^d, there is a probabilistic map $f : R^n \to R^k$ such that for $u, v \in V$:

$$Prob[(1-\varepsilon)|u-v|^2 \leq (|f(u)-f(v)|^2 \leq (1+\varepsilon)|u-v|^2] \geq 1/n$$

and f is computable in randomized polynomial time.

Proof :

We project Y on the first k dimension, for example $d = 10^4$ and $k = 100$. Let Z be the projected point and let $L = |Z^2|$ be its length.

$$I\!E[|Z^2|] = I\!E[\sum_{i=1,...k} Y_i^2] = \sum_{i=1,...k} I\!E[Y_i^2] = \sum_{i=1,...k} I\!E[X_i^2/|X^2|] = k/d$$

Not only the expectation of L is k/d but it is concentrated around its value. It means that if $k \geq 4.\log n/\varepsilon^2$:

$$Prob[L \leq (1-\varepsilon).\mu] \leq 1/n^2$$

$$Prob[L \geq (1+\varepsilon).\mu] \leq 1/n^2$$

Then

$$Prob[L \notin [1-\varepsilon, 1+\varepsilon]] \leq 2/n^2$$

Or:

$$Prob[L \in [1-\varepsilon, 1+\varepsilon]] \geq 1 - 2/n^2$$

Set $f(v) = \sqrt{d/k}.v'$ where v' is the projection of the point v on the first k dimensions. Let u and v be two given points. Let us call $X = u - v$ and X' its projection on the first k coordinates. Then:

$$|f(u) - f(v)|^2 \simeq d/k. \sum_{i=1,\ldots k} {X'_i}^2 = d/k.k/d. \sum_{i=1,\ldots d} {X_i}^2 = |u-v|^2$$

More precisely:

$$Prob[|f(u) - f(v)|^2/|u-v|^2 \notin [1-\varepsilon, 1+\varepsilon]] \leq 2/n^2$$

If we take the $\binom{n}{2}$ pairs (u, v), the probability that some pair has a large distortion is at most $n.(n-1)/2 \cdot 2/n^2 = 1 - 1/n$. Finally:

$$Prob[\forall u, v (1-\varepsilon)|u-v|^2 \leq (|f(u) - f(v)|^2 \leq (1+\varepsilon)|u-v|^2] \geq 1/n$$

The conclusion is that we can repeat several choices of a random projection: at some point we will find one which satisfies the low distortion property.

7.1.1 Random projections

The following program construct a projection matrix of size $(10000, 50)$ using values in $N(0, 1)$.

```
import numpy as np
import matplotlib.pyplot as plt

rn=np.random.standard_normal(10000*50)
rn=rn.reshape(11000,50)
transformation_matrix = np.asmatrix(rn)
```

If we mutliply the original matrix A of dimension (n, d) by the matrix rn of dimension (d, k), we obtain the new matrix A' of dimension n, k, much smaller.

7.1.2 Gaussian distributions

Recall the basic gaussian distribution $\mathcal{N}(0, 1)$:

$$\varphi(x) = \frac{e^{-x^2/2}}{\sqrt{2\pi}}$$

Recall

$$\int_{-\infty}^{\infty} \varphi(x)\,\mathrm{d}x = 1$$

We show this result using the polar coordinates r, θ.

$$(\int_{-\infty}^{\infty} \varphi(x)\,\mathrm{d}x)^2 = \int_{-\infty}^{\infty} \varphi(x)\,\mathrm{d}x. \int_{-\infty}^{\infty} \varphi(y)\,\mathrm{d}y = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} \varphi(x).\varphi(y)\,\mathrm{d}x\mathrm{d}y$$

$$\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} \frac{e^{-x^2/2}}{\sqrt{2\pi}}.\frac{e^{-y^2/2}}{\sqrt{2\pi}}\,\mathrm{d}x\mathrm{d}y = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} \frac{e^{(-x^2-y^2)/2}}{2\pi}\,\mathrm{d}x\mathrm{d}y$$

Introduce the polar coordinates r, θ, i.e. $r^2 = x^2 + y^2$ and $\mathrm{d}x\mathrm{d}y = r.\mathrm{d}r\mathrm{d}\theta$:

$$\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\frac{e^{(-x^2-y^2)/2}}{2\pi}\,\mathrm{d}x\mathrm{d}y=\int_0^{2\pi}\int_0^{\infty}\frac{e^{-r^2/2}}{2\pi}\,r.\mathrm{d}r\mathrm{d}\theta=\int_0^{\infty}r.e^{-r^2/2}\,\mathrm{d}r$$

Introduce $s=-r^2/2$, i.e. $\mathrm{d}s=-r\mathrm{d}r$:

$$\int_0^{\infty}r.e^{-r^2/2}\,\mathrm{d}r=-\int_0^{-\infty}e^s\,\mathrm{d}s=-[e^{-\infty}-e^0]=1$$

The specific property we will use is that if we pick x,y,z as three independent gaussian variables of $\mathcal{N}(0,1)$, let $r^2=x^2+y^2+z^2$ and consider $X=(x/r,y/r,z/r)$, then X is uniformly distributed on the sphere of dimension 3. The density function is:

$$\varphi(x,y,z)=\frac{e^{-x^2/2}}{\sqrt{2\pi}}.\frac{e^{-y^2/2}}{\sqrt{2\pi}}.\frac{e^{-z^2/2}}{\sqrt{2\pi}}=\frac{e^{-r^2/2}}{(2\pi)^{3/2}}$$

Notice that the density function only depends on r but not on θ.

7.2 PCA: Principal Components Analysis

The dimension reduction is a classical subject, called *Principal Components Analysis*. A typical dataset is not a uniform distribution of points, and in some cases we can project each point of dimension d on a space of smaller dimension. Consider 3 points in 4 dimensions,given by the following matrix.

$$A=\begin{pmatrix}0.1 & 2 & 2.0 & 1.3\\ 0.4 & 3 & 3.0 & 1\\ 0.1 & 4 & 4.1 & 1.2\end{pmatrix}$$

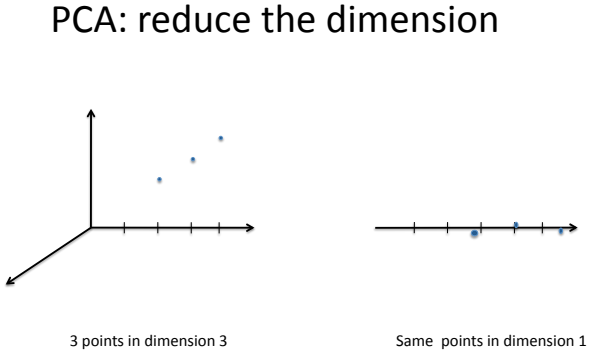


Figure 7.1: Projections on the main eigenvector

In a practical situation, $d=10^4$ and we can project each point on a space of dimension $d'=200$.

7.2.1 Covariance

The covariance of two random variables X_1 and X_2 whose means are μ_1 and μ_2 is

$$cov(X_1, X_2) = I\!E[(X_1 - \mu_1).(X_2 - \mu_2)]$$

Let σ_1 and σ_2 the standard deviations, i.e. $\sigma^2 = I\!E[(X - \mu)^2] = I\!E[X^2] - (I\!E[X])^2$. The covariance is also related to the *Pearson Correlation coefficient*:

$$\rho(X_1, X_2) = cov(X_1, X_2)/\sigma_1.\sigma_2 = I\!E[(X_1 - \mu_1).(X_2 - \mu_2)]/\sigma_1.\sigma_2$$

The covariance matrix B interprets the lines as random variables.

$$B(i, j) = cov(X_i, X_j)$$

In our example, the matrix has 3 lines and the covariance matrix is a $(3, 3)$ matrix. One can also write $B = A.A^t/n$. Unfortunately, sometimes B can also be defined as $A.A^t/n - 1$.

The basic Python instruction to compute the covariance is:

```
B=np.cov(A)
```

It assumes an *numpy* object *np*. We can also write:

```
B=np.cov(L,bias=0)
```

It computes the covariance normalized by $n - 1$. To compute the covariance normalized by n, we write:

```
B=np.cov(L,bias=1)
```

We can also interpret the columns as random variables and in this case we transpose the matrix A. The Python instruction is:

```
A1=A.T
```

Recall the notions of Eigenvalues and Eigenvectors from section 6.2.1. The *eigenvalues* are complex values (in C) but for a large class of matrices, called Positive Semi Definite, the values are reals and positive. Covariance matrices belong to this class.

The Python code, assuming an object *la* of the " linalg" library (linear algebra) is:

```
w,v=la.eig(A)
```

The vector w gives the n eigenvalues. The matrix v gives the n eigenvectors represented as columns.

7.2.2 Gram representation

Given n vectors $v_1, ...v_n$ of dimension n, the Gramian matrix is $G(i, j) = u_i.u_j$. If L is the matrix where the lines are the vectors $v_1, ...v_n$, then:

$$G(i, j) = L.L^t$$

For every positive semidefinite matrix A there is a matrix L such that $A = L.L^t$, in particular for the covariance matrix. The Cholesky decomposition construct such an L. The Python instruction is:

```
L = la.cholesky(A)
```

7.2.3 Principal components

The principal components refer to the eigenvectors associated with the large eigenvalues. If we neglect the small eigenvalues, we will find a close covariance matrix and an original matrix with fewers dimensions, i.e. fewer columns.

7.3 Python code

7.3.1 Covariance, Eigenvectors

```
import numpy as np
import scipy.linalg as la
A = np.array([ [0.1,2,2.0,1.3],[0.4,3,3.0,1],[0.1,4,4.1,1.2] ])
print("A=",A)

B=np.cov(A)
print("B=",B)


w,v=la.eig(B)

print("Eigenvalues of B",w)
print("Eigenvectors of B \n",v)
```

```
----------------------------------------
A= [[ 0.1  2.   2.   1.3]
 [ 0.4  3.   3.   1. ]
 [ 0.1  4.   4.1  1.2]]
B= [[ 0.80333333  1.11666667  1.69333333]
 [ 1.11666667  1.82333333  2.71666667]
 [ 1.69333333  2.71666667  4.05666667]]
Eigenvalues of B [  6.58837818e+00+0.j   9.43713208e-02+0.j   5.83832237e-04+0.j]
Eigenvectors of B
 [[ 0.33086905  0.9325094  -0.1447477 ]
 [ 0.52471327 -0.30928564 -0.79310679]
 [ 0.78434792 -0.18646345  0.5916331 ]]
```

Covariance of A^t

```
import numpy as np
import scipy.linalg as la
A = np.array([ [0.1,2,2.0,1.3],[0.4,3,3.0,1],[0.1,4,4.1,1.2] ])

C=np.cov(A.T)
print(C)


w1,v1=la.eig(C)

print("Eigenvalues of C",w1)
print("Eigenvectors of C \n",v1)
print("reduced A \n", np.dot(A,v1[:,:1]))
```

```
 Result: ------------------------------------------------
 [[ 0.03        0.         -0.005      -0.025     ]
 [ 0.          1.          1.05       -0.05      ]
 [-0.005       1.05        1.10333333 -0.04833333]
 [-0.025      -0.05       -0.04833333  0.02333333]]
Eigenvalues of C [2.10525971e+00+0.j 5.14069593e-02+0.j 8.82897147e-17+0.j
 3.27348862e-18+0.j]
Eigenvectors of C
 [[-0.00134235 -0.76387485 -0.15821894  0.6295915 ]
 [ 0.68915943 -0.04959599  0.71268639 -0.214213  ]
 [ 0.72384112  0.07523669 -0.68130144  0.23772499]
 [-0.03333935  0.63904217 -0.05360244  0.7079648 ]]
reduced A
 [[2.78252571]
 [4.20512536]
 [5.68424486]]
```

7.3.2 Gram's decomposition

```
import numpy as np
import scipy.linalg as la
A = np.array([ [0.1,2,2.0,1.3],[0.4,3,3.0,1],[0.1,4,4.1,1.2] ])
#A=A1.transpose()

B=np.cov(A)
print(B)


w,v=la.eig(B)

print("Eigenvalues of B",w)
print("Eigenvectors of B \n",v)
L = la.cholesky(B)


print("Cholesky decomposition of B \n")
print(L)
print("L.T * L= \n")
print(np.dot(L.T, L))
```

7.3.3 PCA: reconstruction

```
import numpy as np
import scipy.linalg as la
A = np.array([ [0.1,2,2.0,1.3],[0.4,3,3.0,1],[0.1,4,4.1,1.2] ])
#A=A1.transpose()

B=np.cov(A)
print(B)
```

```
w,v=la.eig(B)

print("Eigenvalues of B",w)
print("Eigenvectors of B \n",v)
vt=v[:,[0]]
vt1=vt.transpose()
print("Main Eigenvectors  \n",vt)
print("Main Eigenvectors transposed \n",vt1)
Ak=np.dot(vt1,A)
print("Reduced Ak \n",Ak)


print("-----")

C=np.cov(A.T)
print(C)


w1,v1=la.eig(C)

print("Eigenvalues of C",w1)
print("Eigenvectors of C \n",v1)
v1t=v1[:,[0]]
kA=np.dot(A,v1t)

print("Reduced kA \n",kA)
NA=np.dot(kA,Ak)
print(" A=\n",A)

print(" NA=\n",(1/7.2)*NA)

Result:------------------------------
[[ 0.80333333  1.11666667  1.69333333]
 [ 1.11666667  1.82333333  2.71666667]
 [ 1.69333333  2.71666667  4.05666667]]
Eigenvalues of B [  6.58837818e+00+0.j   9.43713208e-02+0.j   5.83832237e-04+0.j]
Eigenvectors of B
 [[ 0.33086905  0.9325094  -0.1447477 ]
 [ 0.52471327 -0.30928564 -0.79310679]
 [ 0.78434792 -0.18646345  0.5916331 ]]
Main Eigenvectors
 [[ 0.33086905]
 [ 0.52471327]
 [ 0.78434792]]
Main Eigenvectors transposed
 [[ 0.33086905  0.52471327  0.78434792]]
Reduced Ak
 [[ 0.32140701  5.37326958  5.45170437  1.89606054]]
-----
[[ 0.03        0.         -0.005      -0.025     ]
 [ 0.          1.          1.05       -0.05      ]
```

```
 [-0.005       1.05        1.10333333 -0.04833333]
 [-0.025      -0.05       -0.04833333  0.02333333]]
Eigenvalues of C [  2.10525971e+00+0.j   5.14069593e-02+0.j   8.82897147e-17+0.j
   3.27348862e-18+0.j]
Eigenvectors of C
 [[-0.00134235 -0.76387485 -0.15821894  0.6295915 ]
 [ 0.68915943 -0.04959599  0.71268639 -0.214213  ]
 [ 0.72384112  0.07523669 -0.68130144  0.23772499]
 [-0.03333935  0.63904217 -0.05360244  0.7079648 ]]
Reduced kA
 [[ 2.78252571]
 [ 4.20512536]
 [ 5.68424486]]
 A=
 [[ 0.1  2.   2.   1.3]
 [ 0.4  3.   3.   1. ]
 [ 0.1  4.   4.1  1.2]]
 NA=
 [[ 0.12421156  2.076564    2.10687605  0.73275517]
 [ 0.18771622  3.13823225  3.18404171  1.10738503]
 [ 0.25374391  4.24208055  4.30400313  1.49689894]]
```

7.3.4 PCA: reconstruction via the Gram matrix

```
import numpy as np
import scipy.linalg as la
A = np.array([ [0.1,2,2.0,1.3],[0.4,3,3.0,1],[0.1,4,4.1,1.2] ])
#A=A1.transpose()

B=np.cov(A)
print(B)


w,v=la.eig(B)

print("Eigenvalues of B",w)
print("Eigenvectors of B \n",v)
L = la.cholesky(B)


print("Cholesky decomposition of B \n")
print(L)
print("L.T * L= \n")
print(np.dot(L.T, L))


L1=L[:1,:]
print("L1= \n",L1)
```

```
print("reduced A \n", np.dot(L1.T, L1))

Result:------------------------
[[ 0.80333333  1.11666667  1.69333333]
 [ 1.11666667  1.82333333  2.71666667]
 [ 1.69333333  2.71666667  4.05666667]]
Eigenvalues of B [  6.58837818e+00+0.j   9.43713208e-02+0.j   5.83832237e-04+0.j]
Eigenvectors of B
 [[ 0.33086905  0.9325094  -0.1447477 ]
 [ 0.52471327 -0.30928564 -0.79310679]
 [ 0.78434792 -0.18646345  0.5916331 ]]
Cholesky decomposition of B

[[ 0.89628864  1.24587841  1.88927233]
 [ 0.          0.52069217  0.69688598]
 [ 0.          0.          0.04082483]]
L.T * L=

[[ 0.80333333  1.11666667  1.69333333]
 [ 1.11666667  1.82333333  2.71666667]
 [ 1.69333333  2.71666667  4.05666667]]
L1=
 [[ 0.89628864  1.24587841  1.88927233]]
reduced A
 [[ 0.80333333  1.11666667  1.69333333]
 [ 1.11666667  1.552213    2.3538036 ]
 [ 1.69333333  2.3538036   3.56934993]]
```

7.4 Recommendation Systems

Consider 4 customers and 12 products such that $A(i,j) = c$ if c is the intensity of the *like* of customer i for product j.

$$A = \begin{pmatrix} 0 & 1 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 2 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 2 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Notice that the first two lines are close, the last two lines are close and that a line of the first group is far from a line of the last group. We say that A is close to a matrix of rank 2. Remember that the rank of a matrix is the minimum number of independent lines, also equal to the minimum number of independent columns. A line is dependent if it is a linear combinations of other lines.

We could apply the dimension reduction to A. As a matter of fact, we can do something more surprising. Suppose we only know a small fraction of A, i.e. a small proportion of the values $A(i,j)$. Can we retrieve A, i.e. find the missing values? It turns out that is is possible, if we assume that A *is close to a low rank matrix*.

Suppose we hide some of the values of A, i.e. set them to $-$. For example:

$$A = \begin{pmatrix} 0 & 1 & - & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ - & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & - \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & - & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & - & 1 & 0 & - & 0 & 0 & 0 & 0 \end{pmatrix}$$

We want to keep the same norm as the original matrix, so we multiply all the elements by 1.7 and set the $-$ to $\bar{0}$.

$$A = \begin{pmatrix} 0 & 1.7 & \bar{0} & 0 & 0 & 0 & 0 & 0 & 0 & 1.7 & 1.7 & 0 \\ \bar{0} & 0 & 1.7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1.7 & \bar{0} \\ 0 & 0 & 1.7 & 0 & 0 & 1.7 & 1.7 & \bar{0} & 0 & 0 & 0 & 0 \\ 0 & 1.7 & 0 & 0 & \bar{0} & 1.7 & 0 & \bar{0} & 0 & 0 & 0 & 0 \end{pmatrix}$$

We take the following approach, for an (m, n) matrix A:

- Assume A' is the matrix A where only a fraction of A is known, but the norm of A is approximately known. We multiply all the values of A' by some constant, so that A and A' have approximately the same norm.
- We reduce the matrix along its lines to the k largest eigenvalues. We obtain an (k, n) matrix B.
- We reduce the matrix along its columns to the k largest eigenvalues. We obtain an (m, k) matrix C.
- Then A is close to $C.B$.

7.4.1 Python's code: Recommendation Systems

```
import numpy as np
from numpy import linalg as LA
Ainitial = np.array([ [0,1,2,0,0,0,0,0,0,1,1,0], [1,0,1,0,0,0,0,0,0,0,1,0],
[0,0,0,0,0,1,1,2,0,0,0,0],[0,0,0,0,1,1,0,2,0,0,0,0]])

A= np.array([ [0,1.7,0,0,0,0,0,0,0,1.7,1.7,0], [0,0,1.7,0,0,0,0,0,0,0,1.7,0],
[0,0,0,0,0,1.7,1.7,0,0,0,0,0],[0,0,0,0,0,1.7,0,0,0,0,0,0]])

print(A)
B=np.cov(A)
print("Covariance of A \n",B)
w, v = LA.eig(B)
print("Eigenvalues",w)
print("Eigenvectors \n",v)
vt=v[:,[0,1]]
vt1=vt.transpose()
print("Main Eigenvectors  \n",vt)
print("Main Eigenvectors transposed \n",vt1)
Ak=np.dot(vt1,A)
print("Reduced Ak \n",Ak)
print("----------- \n")

A1=A.transpose()

B1=np.cov(A1)
print("Covariance of A transpose \n",B1)
w1, v1 = LA.eig(B1)
```

```
print("Eigenvalues",w1)
#print("Eigenvectors \n",v1)
v1t=v1[:,[0,2]]
kA=np.dot(A,v1t)

print("Reduced kA \n",kA)
NA=np.dot(kA,Ak)
print(" A=\n",A)
print(" Ainitial=\n",Ainitial)
print(" NA=\n",NA)

Result:------------------------
[[ 0.   1.7  0.   0.   0.   0.   0.   0.   0.   1.7  1.7  0. ]
 [ 0.   0.   1.7  0.   0.   0.   0.   0.   0.   0.   1.7  0. ]
 [ 0.   0.   0.   0.   0.   1.7  1.7  0.   0.   0.   0.   0. ]
 [ 0.   0.   0.   0.   0.   1.7  0.   0.   0.   0.   0.   0. ]]
Covariance of A
 [[ 0.59113636  0.13136364 -0.13136364 -0.06568182]
 [ 0.13136364  0.43787879 -0.08757576 -0.04378788]
 [-0.13136364 -0.08757576  0.43787879  0.21893939]
 [-0.06568182 -0.04378788  0.21893939  0.24083333]]
Eigenvalues [ 0.80385779  0.44406254  0.36129026  0.09851667]
Eigenvectors
 [[ 0.67497853  0.58637578  0.44680097 -0.03059932]
 [ 0.40654238  0.20834975 -0.88923574 -0.02394721]
 [-0.52852955  0.63825584 -0.07716004 -0.55437562]
 [-0.31588572  0.45319236 -0.06062189  0.83135906]]
Main Eigenvectors
 [[ 0.67497853  0.58637578]
 [ 0.40654238  0.20834975]
 [-0.52852955  0.63825584]
 [-0.31588572  0.45319236]]
Main Eigenvectors transposed
 [[ 0.67497853  0.40654238 -0.52852955 -0.31588572]
 [ 0.58637578  0.20834975  0.63825584  0.45319236]]
Reduced Ak
 [[ 0.          1.1474635   0.69112205  0.          0.         -1.43550596
  -0.89850024  0.          0.          1.1474635   1.83858555  0.        ]
 [ 0.          0.99683883  0.35419457  0.          0.          1.85546193
   1.08503492  0.          0.          0.99683883  1.3510334   0.        ]]
-----------

Covariance of A transpose
 [[ 0.          0.          0.          0.          0.          0.          0.
   0.          0.          0.          0.          0.        ]
 [ 0.          0.7225     -0.24083333  0.          0.         -0.48166667
  -0.24083333  0.          0.          0.7225      0.48166667  0.        ]
 [ 0.         -0.24083333  0.7225      0.          0.         -0.48166667
  -0.24083333  0.          0.         -0.24083333  0.48166667  0.        ]
 [ 0.          0.          0.          0.          0.          0.          0.
   0.          0.          0.          0.          0.        ]
```

```
 [ 0.          0.          0.          0.          0.          0.          0.
   0.          0.          0.          0.          0.        ]
 [ 0.         -0.48166667 -0.48166667  0.          0.          0.96333333
   0.48166667  0.          0.         -0.48166667 -0.96333333  0.        ]
 [ 0.         -0.24083333 -0.24083333  0.          0.          0.48166667
   0.7225      0.          0.         -0.24083333 -0.48166667  0.        ]
 [ 0.          0.          0.          0.          0.          0.          0.
   0.          0.          0.          0.          0.        ]
 [ 0.          0.          0.          0.          0.          0.          0.
   0.          0.          0.          0.          0.        ]
 [ 0.          0.7225     -0.24083333  0.          0.         -0.48166667
  -0.24083333  0.          0.          0.7225      0.48166667  0.        ]
 [ 0.          0.48166667  0.48166667  0.          0.         -0.96333333
  -0.48166667  0.          0.          0.48166667  0.96333333  0.        ]
 [ 0.          0.          0.          0.          0.          0.          0.
   0.          0.          0.          0.          0.        ]]
Eigenvalues [  3.01000269e+00 +0.00000000e+00j   4.32163351e-01 +0.00000000e+00j
   1.37450062e+00 +0.00000000e+00j  -8.22128312e-17 +9.37970250e-17j
  -8.22128312e-17 -9.37970250e-17j   1.73461671e-49 +0.00000000e+00j
   0.00000000e+00 +0.00000000e+00j   0.00000000e+00 +0.00000000e+00j
   0.00000000e+00 +0.00000000e+00j   0.00000000e+00 +0.00000000e+00j
   0.00000000e+00 +0.00000000e+00j   0.00000000e+00 +0.00000000e+00j]
Reduced kA
 [[-2.15014853+0.j -1.38982446+0.j]
 [-1.25417116+0.j  1.41395382+0.j]
 [ 1.48629494+0.j -0.50900676+0.j]
 [ 0.93169817+0.j -0.28761664+0.j]]
 A=
 [[ 0.   1.7  0.   0.   0.   0.   0.   0.   0.   1.7  1.7  0. ]
 [ 0.   0.   1.7  0.   0.   0.   0.   0.   0.   0.   1.7  0. ]
 [ 0.   0.   0.   0.   0.   1.7  1.7  0.   0.   0.   0.   0. ]
 [ 0.   0.   0.   0.   0.   1.7  0.   0.   0.   0.   0.   0. ]]
 Ainitial=
 [[0 1 2 0 0 0 0 0 0 1 1 0]
 [1 0 1 0 0 0 0 0 0 0 1 0]
 [0 0 0 0 0 1 1 2 0 0 0 0]
 [0 0 0 0 1 1 0 2 0 0 0 0]]
 NA=
 [[ 0.00000000+0.j -3.85264794+0.j -1.97828333+0.j  0.00000000+0.j
   0.00000000+0.j  0.50778465+0.j  0.42390089+0.j  0.00000000+0.j
   0.00000000+0.j -3.85264794+0.j -5.83093127+0.j  0.00000000+0.j]
 [ 0.00000000+0.j -0.02963157+0.j -0.36597057+0.j  0.00000000+0.j
   0.00000000+0.j  4.42390765+0.j  2.66106235+0.j  0.00000000+0.j
   0.00000000+0.j -0.02963157+0.j -0.39560214+0.j  0.00000000+0.j]
 [ 0.00000000+0.j  1.19807150+0.j  0.84692377+0.j  0.00000000+0.j
   0.00000000+0.j -3.07802791+0.j -1.88772647+0.j  0.00000000+0.j
   0.00000000+0.j  1.19807150+0.j  2.04499527+0.j  0.00000000+0.j]
 [ 0.00000000+0.j  0.78238222+0.j  0.54204490+0.j  0.00000000+0.j
   0.00000000+0.j -1.87111999+0.j -1.14920512+0.j  0.00000000+0.j
   0.00000000+0.j  0.78238222+0.j  1.32442711+0.j  0.00000000+0.j]]
```

7.5 Applications

MNIST dataset: https://www.kaggle.com/c/digit-recognizer/data
Applied random projections: https://ashokharnal.wordpress.com/tag/random-projections-tutorial/
Gaussian distributions: https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.randn.html

```
import numpy as np
import PIL.Image as pil
import matplotlib.pyplot as plt
# m is the number of dimensions
# n is the number of points
#  k is the dimension of the projection
m=100
n=50
k=20
#  a is the matric n,m
a = np.random.randn(n, m)
#  k is the dimension of the projection
#  z is the length of the projection
#  r is the sequence of  n lengths  of the projections
#  X is [1,2,....n]
#  x is the length of each row vector
#  we plot the graph  X,r
r=[]
X=[]
for i in range(0,n):
x=0
for j in range(0,m):
x=x+a[i,j]*a[i,j]
z= 0
for j in range(0,k):
z=z+(a[i,j]*a[i,j])/x
r.append(z)
# print(r)
for x in range(0, n):
    X.append(x+1)
plt.plot(X,r)
plt.show()
```

7.6 SVD decompositions

Chapter 8

Learning

Learning is the paradigm where an algorithm can be created from examples, provided by the original data. One distinguishes:

- Supervised learning: in this case each example specifies a class, or a target value y for an input x. We have n examples viewed as $f(x_1) = y_1$, $f(x_2) = y_2$, $...f(x_n) = y_n$ for some unknown function f. We look for a function g such that $f \simeq_\epsilon g$.

- Unsupervised learning: we just have n points $x_1, ...x_n$. We look for a function g into $\{1, 2, ...k\}$ such that if $x_i \simeq x_j$ then $g(x_i) = g(x_j)$ with high probability.

- Reinforcement learning: in this case we are looking for a strategy, i.e. a decision which given some history decides what to do at each step. Games such as Chess and Go are typical examples. We are looking for for a function which decides each move of the game, and an adversary follows a similar strategy.

8.1 Neural Networks

A logistic regression is the combination of a linear function with a non linear function. A neural network is the composition of several logistic regressions, as a circuit. A *deep neural network* is the circuit whose depth is large.

8.1.1 Basic neuron

The basic unit is a *neuron*, a node with incoming and outcoming edges as in Figure 8.1. This model is motivated by the brain cells, although brain cells remain mysterious.

The useful non linear functions are: either the sigmoid function or the Relu (Rectified Linear Unit) function:

- The sigmoid function $f(x) = \frac{1}{1=e^{-x}}$

- The Relu function: $f(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$

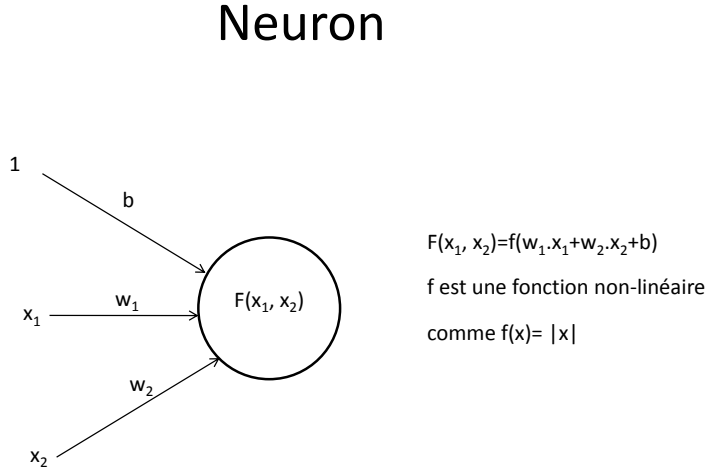


Figure 8.1: A basic neuron:

8.1.2 MLP: Multilayers percepton

Neurons can be organized in layers, which alternate linear and non linear functions. The input layer feeds hidden layers which feed the result of the network, an acyclic directed graph (DAG) as in Figure 8.3. Each node in the hidden and output layer applies a linear transformation from the previous layer and a non linear function f. If X_0 is the intial vector, let X_1 be the 1st layer, X_2 the second layer and X_i the i-th layer. There exists a matrix A_1 such that $X_1 = f(A_1.X_0)$, i.e. we apply the linear transformation followed by the non linear one (f). In general there exists a matrix A_i such that $X_{i+1} = f(A_i.X_i)$.

Assume we choose random weights for the $a_{i,j}$ at the beginning and we try to classify from examples. Each example specifies X_0 and the class as output $Y = 1$ or $Y = 0$ if we have two classes. On the first assume, assume tha class is $Y = 1$. Most likely both outputs will be close to 0.5. We will then change the weights so that the value of the first output will increase and the value of the second node will decrease.

A classical procedure is called *Back Propagation*. It can be understood as a modification of each of the weights as to minimize the final error E defined as $E^2 = (Y - 1)^2$.

8.1.3 Back Propagation

We can view E as a function of the weights $E(a_{i,j})$. If we take the edge $a_{2,2}$, it influences E via the paths indicated on the Figure 8.4.

The variation ΔE can be expressed using the partial derivative of E:

$$\Delta E = \frac{\partial E}{\partial a_{2,2}}.\Delta a_{2,2}$$

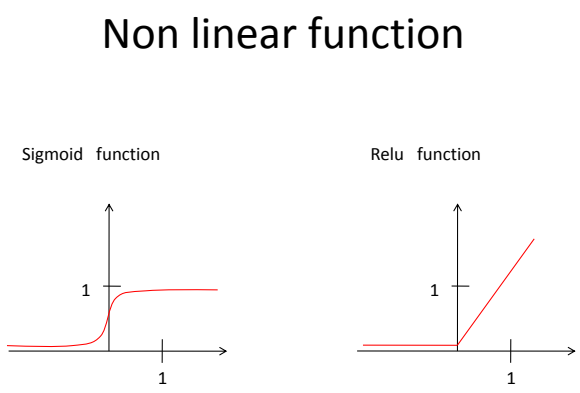


Figure 8.2: Two non linear functions: sigmoid and Relu

Similarly for the node x_i^j reached by $a_{2,2}$:

$$\Delta x_i^j = \frac{\partial x_i^j}{\partial a_{2,2}} . \Delta a_{2,2}$$

Along one path where x_i^j is the value of the i-th node in the j-th layer, we can similarly write :

$$\Delta x_i^{j+1} = \frac{\partial x_i^{j+1}}{\partial x_i^j} . \Delta x_i^j$$

Hence:

$$\Delta x_i^{j+1} = \frac{\partial x_i^{j+1}}{\partial x_i^j} . \frac{\partial x_i^j}{\partial a_{2,2}} . \Delta a_{2,2}$$

For the top path of Figure 8.4, we can write:

$$\Delta E = \frac{\partial E}{\partial x_i^{j+1}} . \frac{\partial x_i^{j+1}}{\partial x_i^j} . \frac{\partial x_i^j}{\partial a_{2,2}} . \Delta a_{2,2}$$

As there are two possible paths:

$$\Delta E = \sum_{2 \ paths} \frac{\partial E}{\partial x_i^{j+1}} . \frac{\partial x_i^{j+1}}{\partial x_i^j} . \frac{\partial x_i^j}{\partial a_{2,2}} . \Delta a_{2,2}$$

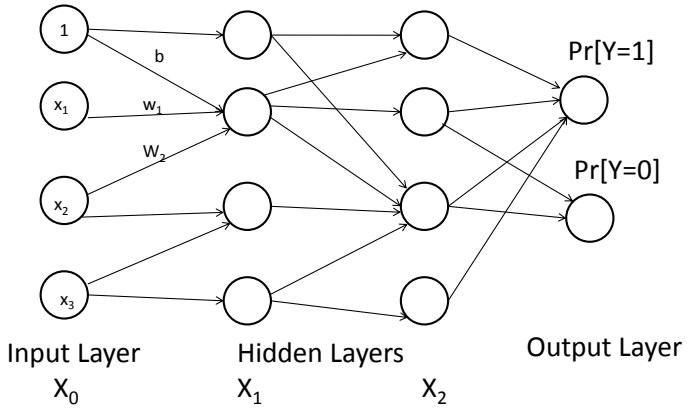


Figure 8.3: Multi layers

8.2 Reinforcement Learning

8.2.1 Markov Decision Processes and Probabilistic Automata

All the MDPs and automata that we will consider in the paper are one the same finite alphabet Σ fixed once and for all.

Definition 8 *A Markov decision Process (MDP) is a 4-uple $\mathcal{S} = (S, \alpha, A, P)$ where S is a finite set of states, α is an* initial distribution on states*, A is a set of* actions*, and $P : S \times A \times S \to [0; 1]$ is the transition relation. $P(s, a, t)$, also writen $P(t|s, a)$, is the probability to arrive in t in one step when the current state is s and action $a \in \Sigma$ is chosen for the transition.*

A *probabilistic automaton (PA)* $\mathcal{A}$ is an MDP with an extra set of final states $F \subseteq S$. MDPs with finite state space and finite action sets have been studied in [12, 5] and probabilistic automata. Statistical notions on the runs generated by a MDP have also been studied in [**?**]. We will generalize these notions in our context in order to obtain approximate algorithms for Membership and Equivalence problems.

We detail now the different types of schedulers or *policies* to resolve the non-determinism of an MDP.

A *history* on $\mathcal{S}$ is a finite or infinite alternating sequence of states and actions, which begins with a state and ends with a state when finite. We write Ω^* for the set of finite histories, and Ω for the set of infinite histories on $\mathcal{S}$.

A *scheduler* on $\mathcal{S}$ is a function $\sigma : \Omega^* \to \bigcup_{s \in S} \Delta(A(s))$ such that for all history $h = (s_1, a_1, ..., , a_{i-1}, s_i)$ on $\mathcal{S}$, $\sigma(h) \in \Delta(A(s_i))$. That is, a scheduler resolves the non determinism of the system by chosing a distribution on the set of available actions from the last state of the given history. We write HR for the set of history dependent and randomized schedulers. A scheduler is *deterministic* when for all history $h = (s_1, a_1, ..., , a_{i-1}, s_i)$ on $\mathcal{S}$, $\sigma(h) \in A(s_i)$. We write HD for the set of history dependent deterministic schedulers.

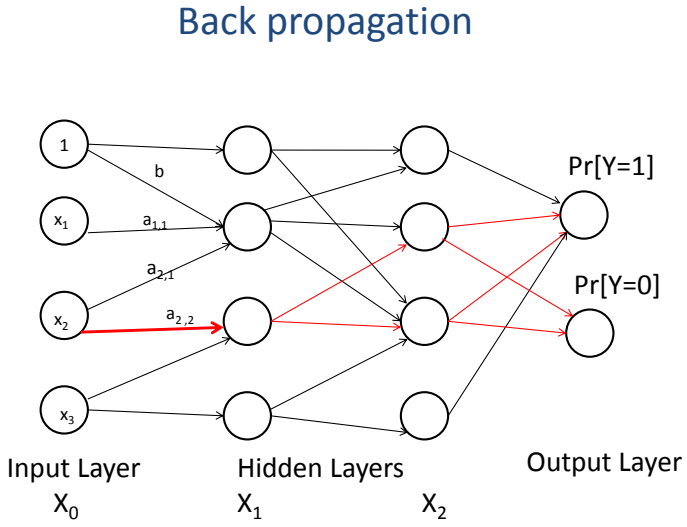


Figure 8.4: Two paths from the edge $a_{2,2}$ to each output

If $k \in \mathbf{N}$, a scheduler σ is said to have *memory* k if for any history $h = (s_1, a_1, ..., , a_{i-1}, s_i)$ of length at least k we have $\sigma((s_1, a_1, ..., , a_{i-1}, s_i)) = \sigma((s_{i-k}, a_{i-k}, ..., , a_{i-1}, s_i))$. we write $MR(i)$ for the set of randomized schedulers which have memory less than i. A scheduler is *stationary*, or *memoryless*, if it has memory 0. That is for any history $h = (s_1, a_1, ..., , a_{i-1}, s_i)$ we have $\sigma(h) = \sigma(s_i)$. We write SR for the set of stationary and randomized schedulers, and we write SD for the set of stationary and deterministic schedulers.

State-Actions Frequencies

We present now the more specific notions of statistics on runs for a MDP. In the following $\mathcal{S}$ is a MDP with initial distribution α. The empirical state-action frequency vectors are random variables.

Definition 9 (Expected state action frequency vector) *Let σ be a scheduler on $\mathcal{S}$ and $T \geq 0$. Let $x^T_{\sigma,\alpha}$ be the vector of $\mathbf{R}^{|S\times\Gamma|}$ whose components are:*

$$x^T_{\sigma,\alpha}(s,a) = \frac{1}{T+1} \sum_{t=0}^{T} P^\sigma_\alpha(X_t = s \wedge Y_t = a),\ \forall s \in S, a \in \Gamma$$

$x^T_{\sigma,\alpha}$ is a distribution on $S \times \Gamma$, hence a vector in $[0;1]^{|S\times\Gamma}$ whose components sum to one. In words, $x^T_{\sigma,\alpha}(s,a)$ is the expected frequency, up to time T of taking state-action (s,a), given the initial distribution of the system is α. Let σ be a scheduler on $\mathcal{S}$. $x^\infty_{\sigma,\alpha}$ is the empty set if $x^T_{\sigma,\alpha}$ does not converge as $T \to +\infty$, and the limit point if $x^T_{\sigma,\alpha}$ converges. We define the following subsets of $\mathbf{R}^{|S\times A|}$:

$$H^{HR}(\alpha) = \bigcup_{\sigma\in HR} x^\infty_{\sigma,\alpha}$$
$$H^{SR}(\alpha) = \bigcup_{\sigma\in SR} x^\infty_{\sigma,\alpha}$$
$$H^{SD}(\alpha) = \bigcup_{\sigma\in SD} x^\infty_{\sigma,\alpha}$$

For any initial distribution α on $\mathcal{S}$, $H^{HR}(\alpha) = \overline{(H^{SR}(\alpha))} = \overline{(H^{SD}(\alpha))}$, where $\overline{X}$ stands for the closed convex hull of X.

The polytope H for communicating MDPs

An MDP is *communicating* if there exists a randomized stationary policy which induces a recurrent Markov chain on its state space. Let $H(\mathcal{S})$ be the set of vectors x in $\mathbf{R}^{|S\times A|}$ that satisfy:

$$x(s,a) \geq 0,\ \forall s,a \in S \times A.\ (E1)$$
$$\sum_{s\in S}\sum_{a\in A} x(s,a) = 1.\ (E2).$$
$$\sum_s \sum_a P(s'|s,a)\cdot x(s,a) = \sum_{a'} x(s',a'),\ \forall s' \in S.\ (E3)$$

We know that for a weakly communicating MDP $\mathcal{S}$ we have $H(\mathcal{S}) = H^{HR}(\alpha)$, for all initial distribution α.

8.2.2 Existence of strategies and Equivalence

Let $\mathcal{S}$ be an MDP, $\epsilon > 0$ and $\lambda > 0$ be given. Given w_n be word of length n, x be its statistic vector of order k, we want to decide if there is a scheduler σ such that when following σ, $\mathcal{S}$ generates runs r_n which is ε-close to w_n or which have statistics ε-close to x, with probability at least $\lambda \in [0;1]$, a fixed threshold value.

Existence of a strategy. *Let w_n be word of length n. Is there a scheduler σ on $\mathcal{S}$ such that the run r_n is such that $\mathbf{P}^n_\sigma[|r_n - w_n| \leq \epsilon] > \lambda$?*

We may also consider infinite runs, given a statistic vector.

Existence of a strategy for a statistic. *Let $x \in \mathbf{R}^{|\Sigma|^k}$ be a statistic vector. Is there a scheduler σ on $\mathcal{S}$ such that: $\mathbf{P}_\sigma[|ustat_k(r) - x| \leq \epsilon] > \lambda$?*

We write $\mathcal{S} \models^{n,\lambda}_\epsilon w_n$ when the answer to the first problem is YES and $\mathcal{S} \models^{\lambda}_\epsilon x$ when the answer to the second is YES. Let $\mathcal{S}_1, \mathcal{S}_2$ be two MDPs on the same alphabet Σ, $\epsilon > 0$ and $\lambda > 0$ be given.

Simulation. *Do we have that for any $x \in \mathbf{R}^{|\Sigma|^k}$: $\mathcal{S}_1 \models^{n,\lambda}_\epsilon x \Longrightarrow \mathcal{S}_2 \models^{n,\lambda}_\epsilon x$?*

We write $\mathcal{S}_1 \prec^{n,\lambda}_\epsilon \mathcal{S}_2$ when this is the case. The *Approximate Equivalence* is the condition: $\mathcal{S}_1 \prec^{n,\lambda}_\epsilon \mathcal{S}_2$ and $\mathcal{S}_2 \prec^{n,\lambda}_\epsilon \mathcal{S}_1$.

Chapter 9

Python

Python main ideas:

- Objects and methods
- Dynamic typing x=1; x='abc'
- Numeric types: int, float, complex, bool
- Data structures: Tuples, Lists and Dictionaries
- Classes: arrays, matrices, graphs,.....
- Main, import

More on the main data structures:

- Tuples

  Tuples are fixed lists. They are often keys to dictionaries.

```
t=(1,2)
t1=('ab',3)

print (t[1] )
```

- Lists

```
nums = [0, 1, 2, 3, 4]
squares = []
for x in nums:
    squares.append(x ** 2)
print(squares)   # Prints [0, 1, 4, 9, 16]

#Alternative

nums = [0, 1, 2, 3, 4]
squares = [x ** 2 for x in nums]
print(squares)   # Prints [0, 1, 4, 9, 16]
```

- Dictionaries

```
d = {'person': 2, 'cat': 4, 'spider': 8}
for animal in d:
    legs = d[animal]
    print('A %s has %d legs' % (animal, legs))

#Alternative

nums = [0, 1, 2, 3, 4]
even_num_to_square = {x: x ** 2 for x in nums if x % 2 == 0}
print(even_num_to_square)  # Prints "{0: 0, 2: 4, 4: 16}"


#Keys as tuples

d = {(x, x + 1): x for x in range(10)}  # Create a dictionary with tuple keys
t = (5, 6)        # Create a tuple
print(type(t))    # Prints "<class 'tuple'>"
print(d[t])       # Prints "5"
print(d[(1, 2)])  # Prints "1"
```

Other types, as classes:

- New Classes

```
class Greeter(object):

    # Constructor
    def __init__(self, name):
        self.name = name  # Create an instance variable

    # Instance method
    def greet(self, loud=False):
        if loud:
            print('HELLO, %s!' % self.name.upper())
        else:
            print('Hello, %s' % self.name)

g = Greeter('Fred')  # Construct an instance of the Greeter class
g.greet()            # Call an instance method; prints "Hello, Fred"
g.greet(loud=True)   # Call an instance method; prints "HELLO, FRED!"
```

- Arrays by Numpy

9.1 Random projections

There are 3 main techniques to construct a projection matrix P:

$f(x) = \frac{e^{-x^2/2}}{\sqrt{2\pi}}$
For three independent trialson x, y, z, then:

$$f(x, y, z) = \frac{e^{-x^2/2}}{\sqrt{2\pi}} \cdot \frac{e^{-y^2/2}}{\sqrt{2\pi}} \cdot \frac{e^{-z^2/2}}{\sqrt{2\pi}} = \frac{e^{-|v|^2/2}}{(2\pi^{3/2}}$$

https://ashokharnal.wordpress.com/tag/random-projections-tutorial/ MNIST https://www.kaggle.com/c/digit-recognizer/data

9.2 Json to Dictionary

```
import requests
import json

dct = {"1": "a", "3": "b", "8": {"12": "c",
"25": "d"}}

for key in dct.keys():
 print(key, dct[key])
```

1 a
3 b
8 '12': 'c', '25': 'd'

```
import requests
import json

dct = {"1": "a", "3": "b", "8": {"12": "c",
"25": "d"}}

print(dct["8"] ["12"])
```

c

9.2.1 Json API

```
import requests
import json

r = requests.get('https://api.coindesk.com/v1/bpi/currentprice.json')
bitcoin_data = dict(r.json())
bitcoin_value = bitcoin_data["bpi"]["USD"]["rate_float"]
print(bitcoin_value)
```

Chapter 10

R

Bibliography

[1] Noga Alon and Michael Krivelevich. Testing k-colorability. *SIAM J. Discrete Math.*, 15(2):211–227, 2002.

[2] Sanjeev Arora and Shmuel Safra. Probabilistic checking of proofs: A new characterization of np. *J. ACM*, 45(1):70–122, 1998.

[3] M. Blum and S. Kannan. Designing programs that check their work. *Journal of the ACM*, 42(1):269–291, 1995.

[4] M. Blum, M. Luby, and R. Rubinfeld. Self-testing/correcting with applications to numerical problems. *ACM Symposium on Theory of Computing*, pages 73–83, 1990.

[5] C. Derman. *Finite State Markovian Decision Processes*. Academic Press, Inc. Orlando, FL, USA, 1970.

[6] P. Erdös and A Renyi. On the evolution of random graphs. In *Publication of the mathematical institute of the Hungarian Academy of Sciences*, pages 17–61, 1960.

[7] O. Goldreich, S. Goldwasser, and D. Ron. Property testing and its connection to learning and approximation. *Journal of the ACM*, 45(4):653–750, 1998.

[8] Oded Goldreich and Luca Trevisan. Three theorems regarding testing graph properties. *Random Struct. Algorithms*, 23(1):23–57, 2003.

[9] Michael Molloy and Bruce Reed. The size of the giant component of a random graph with a given degree sequence. *Comb. Probab. Comput.*, 7(3):295–305, September 1998.

[10] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[11] Mark Newman. *Networks: An Introduction*. Oxford University Press, Inc., 2010.

[12] M.L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc. New York, NY, USA, 1994.

[13] R. Rubinfeld and M. Sudan. Robust characterizations of polynomials with applications to program testing. *SIAM Journal on Computing*, 25(2):23–32, 1996.

[14] Jeffrey S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, 1985.