EUROPEAN MATHEMATICAL SOCIETY

Nicolas Errandonea · Andres Gutierrez · Roylan Martinez · Sergi Sanjuan

DISCRETE MARKOV CHAIN APPLICATIONS

Abstract. A stochastic process without memory is called a Markov process. Despite the diversity of these processes, such generalization provides a theoretical framework that allows encompassing all those systems of states that depend solely on the current state. The universality of these systems relates the problem to a large number of areas of physics and mathematics. The objective of this review is to familiarize the reader with its usefulness in four different mathematical problems, providing a perspective on the conceptual importance of these chains. In turn, the mentioned areas will be presented in-depth, issuing a necessary theoretical context to highlight the need for the concept of the Markov chain.

Keywords. markov chains; markov process; stochastic; learning; hidden markov model; random walk; stochastic differential equations

1. Introduction

As described in the abstract, a stochastic process without memory is called a Markov process, it is usually represented via lineal algebra, by nodes among other mathematical representations, but with the same idea of transitions from a specific state to another, between a finite or countable number of states. Markov chains is particular stochastic process that matches the Markov property, which basically states that the probability of transitioning to any other particular state is dependent uniquely on the current state and in this paper we will only focus on the discrete applications and modelling.

Nicolas Errandonea: 46015;; ebani@alumni.uv.es

Andres Gutierrez: 46008;; angujai@alumni.uv.es

Roylan Martinez: 08203;; grmarvar@alumni.uv.es

Sergi Sanjuan: 03841;; ssansil@alumno.upv.es

Markov chains have many applications, including finance or economics —modeling stock prices—, epidemiology —modeling the spread of diseases— or natural language processing —modeling word sequences in text—- and therefore its applications vary from pure and theoretical to quite practical ones.

The Markov chains are backed with a state diagram, usually represented through a matrix that can be depicted as a graph, it contains nodes and edges — and it is a directed graph so in this case directed edges — in which each node represents a state and each directed edge represents a transition between states with a given probability. Finally the probabilistic dynamicity of the state is represented through a transition matrix which is nothing else than a square matrix where each entry represents the probability of transitioning from one state to another.

Markov chains can be classified in and under different parameters and features but in general terms they are separated by the regular Markov chains, absorbing Markov chains, ergodic Markov chains and the periodic Markov Chain among others but on general terms depending on the properties of the states and transitions.

So, in general terms, Markov chains are a powerful tool for modeling, studying, predicting, simulating and understanding complex behaviours and systems in the pure and applied applications accompanied then with many applications as we will review [1].

2. Reinforcement learning

The objective of this section is to present the usefulness of Markov chains in the area of machine learning as a fundamental tool to define control problems. Hand in hand with this concept, we will introduce reinforcement learning and provide tools and methods derived from its theoretical framework necessary to solve this type of problem.

The fundamental scenario of reinforcement learning consists of an agent interacting with a given enviroment through actions. Each time the agent performs an action, it receives a reward and an update on the status of the enviroment. The objective of the agent will be to find an optimal strategy for taking actions to maximize the rewards. From now on, we will call each one of the strategies policies.

In the case that the next states only depend on the current state and action, the problem can be posed as a Markov Decision Problem (MDP). If the properties of the enviroment are known, we will review how to solve the MDP using strategies known as dynamic programming. If the enviroment is unknown, we will present learning techniques based on stochastic approximation to solve the MDP. Finally, for large-dimensional problems where all preceding strategies fail due to high computational cost, we will present the Temporal Differences (TD) learning techniques as a plausible alternative for the problem resolution. The section will conclude with Tesauro's Backgammon problem [21] [22], a problem solved by the TD method that resulted in a turning point in the area.

2.1. Markov Decision Problems and Dynamic Programming

Firstly, we will introduce a Markov decision process and the initial theoretical framework following the development of the renowned book by Mohri [7], which will be necessary to develop the algorithms that solve the MDP.

Definition 2.1. *It is defined as a Markov decision process a model agent-enviroment described by the following:*

- *A set of states S.*
- *An indexing time set T.*
- *An initial state* s_0.
- *A set of actions A*
- *The transition probabilities* $P[s' \mid (s, a)]$.
- *The reward probabilities* $P[r \mid (s, a)]$.

We denote the process Markovian as the transition and reward probabilities depend solely on the current state and action. Along this review, we will always consider an infinite discrete-time process, with finite sets of states and actions. As the probabilities are not time-dependent, the Markov decision process considered will be stationary.

A MDP is defined as a control problem that can be described by a Markov decision process. Therefore, the resolution of the MDP will lie in determining the best course of action in each of the different states, or in other words, selecting a policy.

Definition 2.2. *A policy* π *is defined as a map* $\pi : S \longrightarrow \Delta(A)$*, where* $\Delta(A)$ *is the set of all possible probability distributions in A. A policy is deterministic if the mapping can be rewritten as* $\pi : S \longrightarrow A$*, that is, a policy where the action to be taken is always completely determined.*

Note that if the agent follows a deterministic policy π, the Markov decision process can be represented as a Markov chain. The agent no longer makes any decisions, since there are given by the policy, and the change from one state to another in the chain will be given by the transition matrix T_π, where $T_\pi(i, j) = P(s^j \mid (s^i, \pi(s^i))$ and s^i represents the state i (not to be confused with the state of the process at time i).

It is reasonable to deduce that eventually, a criterion to compare policies will be needed. Therefore it would be wise to associate the policies with a value or set of values. We proceed with the following definition:

Definition 2.3. *The discounted value* $V_\pi(s)$ *of a (deterministic) policy* π *whose initial state is s is defined as the expected value of the discounted rewards received when a MPD starts*

at s and the agent follows the policy π.

$$V_\pi(s) = \mathbb{E}_{a_t=\pi(s_t)}[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) \mid s_0 = s, a_0 = \pi(s_0)]. \tag{2.1}$$

The expectation is regarding the states s_t reached and the rewards $r(s_t, a_t) \in \Delta(R(s_t, a_t))$ obtained. Note that a parameter $\gamma \in (0, 1)$ is used to prevent any of these values from not being bounded so as to have a way to compare policies. Normally the value of γ is close to one.

It is natural to seek to apply policies with the largest possible $V_\pi(s)$ values. In fact, as we will see further in the section, for all MDP there will be optimal policies defined as follows:

Definition 2.4. *A deterministic policy π is optimal if for every other deterministic policy π^* it is true that $V_\pi(s) \geq V_{\pi^*}(s) \quad \forall s \in S$.*

Note that the previous definitions could have been generalized for any policy, and not only for deterministic policies. This said, in the following pages we will just consider deterministic policies, and we will build all subsequent definitions in this fashion. The reason behind it is that the notation becomes much simpler, and it is proven in [7] that there is always an optimal policy for all MDP that is deterministic.

Mohri first proves a series of properties that policies and optimal policies must abide, and then concludes (theorem 17.7[7]) that there is always an optimal deterministic policy. Therefore, and given that the reasoning to reach the latter conclusion is analogous considering general policies or only deterministic ones, we will proceed in the latter way and let the reader intuit how the demonstrations and properties that will be introduced could be generalized.

In order to prove the existence of optimal policies, the following definition will be necesary:

Definition 2.5. *Let $s \in S$ and $a \in A$. The state-action value $Q_\pi(s, a)$ associated with a certain policy π is defined as the expected value of the sum of rewards by taking the action a for the initial value s and following the policy π in the latter stages, that is:*

$$Q_\pi(s,a) = \mathbb{E}[r(s,a)] + \mathbb{E}_{a_t=\pi(s_t)}[\sum_{t=1}^{\infty} \gamma^t r(s_t, a_t) \mid s_0 = s, a_0 = a] = \mathbb{E}[r(s,a) + \gamma V_\pi(s_1) \mid s_0 = s, a_0 = a]. \tag{2.2}$$

Note that when $a = \pi(s)$ we will have $Q_\pi(s, \pi(s)) = V_\pi(s)$.

As we already have enough tools to prove that there is always an optimal policy, we will proceed with the following theorems that will lead to said result.

Theorem 2.1. *Let two policies be π and $\pi*$. It is true that :*

$$\forall s \in S \;\; Q_{\pi^*}(s, \pi(s)) \geq Q_{\pi^*}(s, \pi^*(s)) \longrightarrow V_{\pi}(s) \geq V_{\pi^*}(s) \;\; \forall s \in S. \tag{2.3}$$

As long as one of the inequalities on the left hand side is strictly fulfilled, we will have that at least one of the inequalities on the right hand side is also strictly fulfilled

Proof. We assume that the first condition is fulfilled. Therefore let $s \in S$ we have:

$V_{\pi^*}(s) = Q_{\pi^*}(s, \pi^*(s)) \leq Q_{\pi^*}(s, \pi(s)) = \mathbb{E}[r(s, \pi(s) + \gamma V_{\pi^*}(s_1) \mid s_0 = s] =$
$\mathbb{E}[r(s, \pi(s)) + \gamma Q_{\pi^*}(s_1, \pi^*(s_1)) \mid s_0 = s] \leq \mathbb{E}[r(s, \pi(s)) + \gamma Q_{\pi^*}(s_1, \pi(s_1)) \mid s_0 = s]$
$= \mathbb{E}[r(s, \pi(s)) + \gamma r(s_1, \pi(s_1)) + \gamma^2 V_{\pi^*}(s_2) \mid s_0 = s]$.

Following these inequalities T times it can be deduce that

$$V_{\pi^*}(s) \leq \mathbb{E}[\sum_{t=0}^{T} r(s_t, \pi(s_t)) + \gamma^{T+1} V_{\pi^*}(s_{T+1}) \mid s_0 = s],$$

where taking the limits to infinity we end up with

$$V_{\pi^*}(s) \leq \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t r(s_t, \pi(s_t)) \mid s_0 = s] = V_{\pi}(s).$$

Let us note that if for the state $s = s_0 \quad Q_{\pi^*}(s_0, \pi^*(s_0)) < Q_{\pi^*}(s_0, \pi(s_0))$ is strictly fulfilled we have just verified that the same thing happens for $V_{\pi^*}(s_0) < V_{\pi}(s_0)$.

■

Theorem 2.2. *A policy π is optimal if $\forall s \in S$ holds that*

$$\pi(s) \in argmax_{a \in A} Q_{\pi}(s, a).$$

Proof. $\longrightarrow$ We assume that the first condition is true. Therefore, $\forall \pi*$ and $\forall s \in S$ holds that $V_{\pi}(s) \geq V_{\pi*}(s)$. Let $s_0 \in S$. We have that the previous inequality holds in particular for all policies identical to π except in the state s_o. Therefore each of these policies π^* satisfies that $\forall s \in S \setminus \{s_0\} \quad Q_{\pi}(s, \pi^*(s)) \geq Q_{\pi}(s, \pi(s))$. By the previous theorem and its consequence, we can conclude that $\pi(s_0) \in argmax_{a \in A} Q_{\pi}(s_0, a)$.Otherwise π would not be an optimal policy.

$\longleftarrow$ The proof of this implication formulated by Mohri [7] is inconsistent. Therefore, we include the correct proof derived from Puterman's approach in [12]. This approach includes the use of some matrix notation that will be presented in more detail as the section progresses.

Suppose we have a policy π^* such that it satisfies the second condition. Let's see that for any other policy π we have $V_{\pi^*} \geq V_{\pi}$, where the inequality is component by component. First, let's note that $Q_{\pi^*}(s, \pi^*(s)) = max_{a \in A} Q_{\pi^*}(s, a)$ holds $\forall s \in S$, in particular

$V_{\pi^*}(s) = Q_{\pi^*}(s, \pi^*(s)) \geq Q_{\pi^*}(s, \pi(s)) \quad \forall s \in S$. Therefore, if we rewrite the expressions in matrix form we get $V_{\pi^*} \geq R_\pi + \gamma T_\pi V_{\pi^*}$, where T_π is the transition matrix associated with the policy π and R_π the expected reward vector, where element i is $\mathbb{E}[r(s^i, \pi(s^i)]$.

On the other hand, we also have the following matrix equality $V_\pi = R_\pi + \gamma T_\pi V_\pi$. Therefore, nesting the inequality n times, the equality infinite times, and subtracting both expressions we obtain the following inequality:

$$V_{\pi^*} - V_\pi \geq \gamma^n (T_\pi)^n V_{\pi^*} - \sum_{t=n}^{\infty} \gamma^t (T_\pi)^t R_\pi$$

Let $\epsilon > 0$. Since $(\gamma T_\pi)^t \to 0$, we can choose n large enough such that $\gamma^n ||(T_\pi)^n R_\pi||_\infty \leq \epsilon$. Therefore:

$$V_{\pi^*} - V_\pi \geq -\sum_{t=n}^{\infty} \gamma^t (T_\pi)^t R_\pi \geq -\frac{\epsilon\gamma}{1-\gamma} 1_S \quad \forall \epsilon > 0$$

We then conclude that $V_{\pi^*} \geq V_\pi \quad \forall \pi$, so π^* is optimal.

■

Theorem 2.3. *For all MDP (with the restrictions used in the definition) there is an optimal policy.*

Proof. Let α be $\alpha = argmax_\pi \sum_{s \in S} V_\pi(s)$, which we know exists because there are only $|A| \cdot |S|$ policies . We assume that α is not an optimal policy. Therefore, by the previous theorem, we can assume that there exists a s_0 and a a_0 such that $Q_\alpha(s_0, \alpha(s_0)) < Q_\alpha(s_0, a_0)$. Applying Theorem 2.1 for the policy π identical to α except in $\pi(s_0) = a_0$, we have that $\forall s \in S$ holds $V_\pi(s) \geq V_\alpha(s)$, where the inequality is surely strict in more than one term.We then conclude that $\sum_{s \in S} V_\pi(s) > \sum_{s \in S} V_\alpha(s)$, finding a contradiction. ■

The Bellman optimality equations can be defined based on the relationship between the discounted values V^* of the optimal policy π^* and the state-action values of the same policy that is deduced from Theorem 2.2

$$V^*(s) = Q^*(s, \pi^*(s)) = max_{a \in A} Q^*(s, a),$$

as well as on the very definition of these state-action values. From now on, V^* will be denoted as the optimal value, and the state-action values of the optimal policy will be denoted as $Q^*(s, a)$.

Definition 2.6 (**Bellman optimality ecuations**). *The optimal discounted values of a MDP satisfy the following system of equations:*

$$\forall s \in S \quad V^*(s) = max_{a \in A} \left\{ \mathbb{E}[r(s, a)] + \gamma \sum_{s' \in S} \mathbb{P}[s' \mid (s, a)] V^*(s') \right\}. \tag{2.4}$$

Furthermore, if we consider that for each policy π that $V_\pi(s) = Q_\pi(s, \pi(s))$

Definition 2.7 (**Bellman policy equations**). *The discounted values of a policy π satisfy the following linear system of equations:*

$$\forall s \in S \quad V_\pi(s) = \mathbb{E}[r(s, \pi(s))] + \gamma \sum_{s' \in S} \mathbb{P}[s' \mid (s, \pi(s))] V_\pi(s'). \tag{2.5}$$

Since the previous system is linear, it can be written as the following matrix system $V_\pi = R_\pi + \gamma T_\pi V_\pi$, where T_π is the transition matrix associated with the policy π and R_π the expected reward vector, where element i is $\mathbb{E}[r(s^i, \pi(s^i)]$.It is proven in [7] that the matrix $(I - \gamma T_\pi)$ is always invertible ,so we can conclude that:

Theorem 2.4. *The Bellman policy equations have a unique solution $V_\pi = (1 - \gamma T_\pi)^{-1} R_\pi$*

The Bellman equations form the core of dynamic programming, where algorithms use these equations to generate convergent methods to the optimal values V^* or the optimal policies π^*. The need for these algorithms derives from the impossibility of solving the optimality Bellman equations directly, as well as the computational cost of solving the system of equations associated with each of the policies to obtain V_π. However, these algorithms will need to know explicitly the MDP enviroment conditions to solve it, that is, the transition matrices T_π and rewards R_π, so they cannot be applied to problems where the enviroment is unknown. The most widely used algorithms are VI (value interaction) and PI (policy interaction).

The VI method is an algorithm that seeks to find the optimal policy obtaining the optimal values V^* by iteratively solving the Bellman optimality equations. The iterations are given by evaluating V values in the following function $\theta(V)$:

$$\forall s \in S \quad [\theta(V)](s) = max_{a \in A} \Big\{ \mathbb{E}[r(s, a)] + \gamma \sum_{s' \in S} \mathbb{P}[s' \mid (s, a)] V(s') \Big\},$$

which is nothing more than the right-hand side of Bellman's optimality equations. For each $s \in S$ it is selected the value $a \in A$ that maximizes the previous expression, that is, a policy. Therefore the function may be rewritten as $\theta(V) = max_\pi \{R_\pi + \gamma T_\pi V\}$ Let's see in **Algorithm 1** how the method is structured :

To prove that this method is actually convergent and appropriate for our problem we will have to check the following:

Theorem 2.5. *The three following statements are met:*

- *For every initial value V_0 we have that the sequence $V_{n+1} = \theta(V_n)$ converges to V^*.*
- *If $||V_{n+1} - V_n||_\infty \leq \frac{(1-\gamma)\epsilon}{\gamma}$ then $||V^* - V_{n+1}||_\infty \leq \epsilon$.*

Algorithm 1 VI algorithm

Initialize V_0 with a random values
Select $\gamma \in (0, 1)$
$V \longleftarrow V_0$
while $||V - \theta(V)||_\infty \geq \frac{(1-\gamma)\epsilon}{\gamma}$ **do**
  $V \longleftarrow \theta(V)$
end while
$\pi \in argmax_\pi \{ R_\pi + \gamma T_\pi V \}$
return π

- *If* $||V^* - V_n||_\infty \leq \epsilon$ *The policy* $\pi \in argmax_\pi \{ R_\pi + T_\pi V_n \}$ *has* $||V^* - V_\pi||_\infty \leq \frac{2\epsilon}{1-\gamma}$ *near-optimal values* .

Proof. First of all, it is evident that $V^* = \theta(V^*)$ since the optimal values satisfy Bellman's optimality equations. On the other hand, the function θ is γ-Lipschitz for the infinity norm (Theorem 17.11 [7]). Therefore, it can be deduced that for $n \in \mathbb{N}$

$$||V^* - V_{n+1}||_\infty = ||\theta(V^*) - \theta(V_n)||_\infty \leq \gamma||V^* - V_n||_\infty \leq \gamma^{n+1}||V^* - V_0||_\infty,$$

concluding that the sequence will converge to V^* regardless of the initial value V_0.

At the same time,

$$||V^* - V_{n+1}||_\infty \leq ||V^* - \theta(V_{n+1})||_\infty + ||\theta(V_{n+1}) - V_{n+1}||_\infty = ||\theta(V^*) - \theta(V_{n+1})||_\infty +$$
$$||\theta(V_{n+1}) - \theta(V_n)||_\infty \leq \gamma||V^* - V_{n+1}||_\infty + \gamma||V_{n+1} - V_n||_\infty.$$

Therefore regrouping terms we obtain that $||V^* - V_{n+1}||_\infty \leq \frac{\gamma}{1-\gamma}||V_{n+1} - V_n||_\infty$. It just has been proved that if $||V_{n+1} - V_n||_\infty \leq (1 - \gamma)\epsilon/\gamma$ then $||V^* - V_{n+1}||_\infty \leq \epsilon$. The ϵ-convergence of the algorithm will be of the order of $log(1/\epsilon)$, as it is shown in (Theorem 17.11 [7]).

Finally take the policy $\pi \in argmax_\pi \{ R_\pi + T_\pi V_n \}$ where $||V^* - V_n||_\infty \leq \epsilon$. It can be deduced that $R_\pi + T_\pi V^* \geq R_\pi + T_\pi(V_n - \epsilon) = V_{n+1} - \epsilon \geq V^* + 2\epsilon$, where the inequalities are component to component. It can be concluded that $\forall s \in S \quad Q^*(s, \pi(s)) + 2\epsilon \geq Q^*(s, \pi^*(s))$. Proceeding with a demonstration similar to the one used Theorem 2.1, let $s \in S$:

$$\mathrm{V}^*(s) = Q_{\pi^*}(s, \pi^*(s)) \leq Q_{\pi^*}(s, \pi(s)) + 2\epsilon = \mathbb{E}[r(s, \pi(s) + \gamma V^*(s_1) \mid s_0 = s] + 2\epsilon =$$
$$\mathrm{E}\ [\mathrm{r}(\mathrm{s},\pi(\mathrm{s})) + \gamma Q_{\pi^*}(s_1, \pi^*(s_1)) \mid s_0 = s] + 2\epsilon \leq \mathbb{E}[r(s, \pi(s)) + \gamma(Q_{\pi^*}(s_1, \pi(s_1)) +$$
$$2\epsilon) \mid s_0 = s] + 2\epsilon = \mathbb{E}[r(s, \pi(s)) + \gamma r(s_1, \pi(s_1)) + \gamma^2 V_{\pi^*}(s_2) \mid s_0 = s] + 2\epsilon + \gamma 2\epsilon.$$

Following these inequalities T times we can deduce that

$$V_{\pi^*}(s) \leq \mathbb{E}[\sum_{t=0}^{T} \gamma^t r(s_t, \pi(s_t)) + \sum_{t=0}^{T} \gamma^t 2\epsilon + \gamma^{T+1} V_{\pi^*}(s_{T+1}) \mid s_0 = s],$$

were taking the limits to infinity we end up with

$$V_{\pi^*}(s) \leq \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t r(s_t, \pi(s_t)) \mid s_0 = s] + \sum_{t=0}^{\infty} \gamma^t 2\epsilon = V_\pi(s) + \frac{2\epsilon}{1-\gamma}.$$

We conclude that $||V^* - V_\pi||_\infty \leq \frac{2\epsilon}{1-\gamma}$.

■

On the other hand, the PI algorithm consists of evaluating the value of certain policies, and from these obtained values building a new policy that is superior to its predecessor. Let's see the structure of PI in **Algorithm 2**:

Algorithm 2 PI algorithm

Initialize π_0 with a random values
$\pi \longleftarrow \pi_0$
$V_\pi = (I - \gamma T_\pi)^{-1} R_\pi$
while $V_\pi \neq max_{\pi^*}\{ R_{\pi^*} + \gamma T_{\pi^*} V_\pi \}$ **do**
    $\pi = argmax_{\pi^*}\{ R_{\pi^*} + \gamma T_{\pi^*} V_\pi \}$
    $V_\pi = (I - \gamma T_\pi)^{-1} R_\pi$
end while
    return π

Proving that this algorithm converges is relatively simpler than in the previous case. For each new policy π^* elaborated $\forall s \in S \;\; Q_\pi(s, \pi^*(s)) \geq Q_\pi(s, \pi(s))$, so it follows that $V_{\pi^*} \geq V_\pi$.In the case of not being able to update the policy, it will be true that $\forall s \in S \;\; \pi(s) = argmax_{a \in A} \; Q_\pi(s, a)$ so it is proven that the policy is optimal.

It is proven in Theorem 17.13 [7] that if the initial values of VI are V_{π_0} and the initial policy of PI is π_0 then PI will converge in a smaller number of steps. Despite this, each of these steps will be more computationally expensive, as inverse matrices have to be computed.

2.2. Stochastic approximation to solve the curse of modeling

From this point on we will consider a more general scenario for MDPs, where the enviroment might be unknown, that is, the agent will not have access to the transition matrices or the probability distributions of the rewards. Therefore, dynamic programming algorithms become useless for this type of problem. This situation is commonly known as the modeling curse.

So, how can the agent try to figure out which is the best policy in this context? As the reader already might have deduced, it is necessary for him to try to obtain information from the system by interacting with it. Within reinforcement learning, two types of approaches

are employed for this situation. Those algorithms prepared for obtaining the optimal policy directly are known as model-free approaches while the algorithms that try first to predict the enviromental conditions of the problem are defined as model-based approaches. In this section, only model-free algorithms will be introduced.

Since the algorithms used for reinforcement learning are related to stochastic approximation techniques, some necessary results will be presented that will guarantee the convergence of our algorithm. Let us remember that stochastic algorithms are used to solve optimization problems whose objective function has a single fixed point and is only accessible through noisy data or when the function is defined as an expected value of random variables.

More formally, in a stochastic optimization problem we seek to obtain the solution of a system $x = H(x)$ from which we cannot obtain the expression of H, but only samples i.i.d. with a certain noise $H(x_i) + w_i$.In our case, Q^*, the vector with all the values $Q^*(s, a)$, will be the fixed value of a certain function H.

The first result [14]is based on the law of large numbers and will allow the reader to get insight of the reason for the convergence of the algorithms that will be presented below.

Theorem 2.6 (**Robbins-Monro mean estimation**). *Let X be a random variable with values in $[0, 1]$ and x_1, x_2, x_m samples i.i.d. of X. Then the series $(\mu_m)_{m\in\mathbb{N}}$ with values:*

$$\mu_{m+1} = (1 - \alpha_m)\mu_m + \alpha_m(x_{m+1}),$$

that meets the following conditions $\mu_0 = x_0$, $\alpha_m \in [0, 1]$, $\sum_{n=0}^{\infty} \alpha_n = \infty$, $\sum_{n=0}^{\infty} \alpha_n^2 \leq \infty$ converges almost surely to the expected value of X.

The following theorem [7], more complex, will be the one that guarantees the convergence of the two algorithms that will be presented further in this section.

Theorem 2.7. *Let H be a mapping $H : \mathbb{R}^n \longrightarrow \mathbb{R}^n$, $(w_m)_{m\in\mathbb{N}} \in \mathbb{R}^n$ a series of random variables, $(\alpha_m)_{m\in\mathbb{N}} \in \mathbb{R}^+$ a sequence of real values , and $(x_m)_{m\in\mathbb{N}} \in \mathbb{R}^n$ a sequence defined as:*

$$\forall n \in \{1, ...N\} \quad x_{t+1}(s) = x_t(s) + \alpha_t(s)[H(x_t)(s) - x_t(s) + w_t(s)],$$

for a certain x_0 . It is defined F_t as $F_t = \{(x_{t'})_{t'\leq t}, (w_{t'})_{t'\leq t}, (a_{t'})_{t'\leq t}\}$ and it is assumed that the following conditions are met:

- $\exists K_1, K_2 \in \mathbb{R}^{\kappa} \quad \mathbb{E}[||w_t||^2(s) \mid F_t] \leq K_1 + K_2||x_t||^2$ *for some norm .*
- $\mathbb{E}[w_t \mid F_t] = 0$
- $\forall n \in \{1, ...N\} \quad \sum_{n=0}^{\infty} \alpha_n(s) = \infty \quad \sum_{n=0}^{\infty} \alpha_n^2(s) \leq \infty$
- *H is $||\ \ ||_\infty$ β-Lipschitz where $\beta \in (0, 1)$ with a fixed point x^*.*

Then, the sequence x_t converges almost certainly to x^*.

Reinforcement learning algorithms shown in this review consist of two basic units: the learning policy, which selects which state-action pairs are explored during execution, and the update rule, which defines the new estimate of the values that there are approximating in each iteration.

This said,let's go on and present two well-known stochastic algorithms of Reinforcement learning, where both algorithms approximate $Q^*(s,a)$ and are based on the formal definition of optimal state-action values, that is:

$$Q^*(a,s) = \mathbb{E}[r(s,a)] + \gamma \sum_{s' \in S} \mathbb{P}[s' \mid (s,a)] V^*(s') = \mathbb{E}[r(s,a) + \gamma max_{a \in A} Q^*(s',a)].$$

We will first present the Q-learnirg algorithm in **Algorithm 3**:

Algorithm 3 Q-learning algorithm

We initialize all $Q_0(s,a) = 0$
$Q \longleftarrow Q_0$
for t = 0 : T **do**
  Select $s \in S$ randomly
  for each step of t epoch **do**
    Select $a \in A$ using $\theta(Q,s)$ learning policy.
    Let $r(s,a)$ be the reward obtained
    Let s' be the next state of the enviroment
    $Q(s,a) \leftarrow Q(s,a) + \alpha_m(s,a)[r + \gamma max_{a' \in A} Q(s',a') - Q(s,a)]$
    $s \leftarrow s'$
  end for
end for
$\pi(s) = argmax_{a \in A} Q(s,a)$
  return π

This algorithm does not need to compute the properties of the enviroment, but only to be able to interact with it (which is exclusively the objective of the subsection).

First, the algorithm encounters the current state of the enviroment and selects action a following the learning policy $\theta(Q^*,s)$. Note that this algorithm learning policy is different from an MDP policy, as will be explained below. Once the action is selected, the enviroment will grant the agent a reward and a new state, and $Q^*(s,a)$ will be updated

$$Q^*(s,a) \leftarrow Q^*(s,a) + \alpha_m(s,a)[r + \gamma max_{a'} Q^*(s',a') - Q^*(s,a)]$$

as described, where m is the number of times $Q^*(s,a)$ has been updated. We take the new $s \leftarrow s'$ and repeat the process a certain number of times defined by the algorithm until it returns to the initial situation where it starts a new interaction with the enviroment.

Robbins Monro's theorem gives us an intuition of why the algorithm is convergent, although the theorem that justifies its convergence is Theorem 2.7. Theorem 17.18 of [7] provides a detailed proof, from which it can be concluded that the algorithm will be convergent only when for an infinite number of iterations T all $Q^*(s,a)$ are updated infinite times and :

$$\forall s \in S \; \forall a \in A \quad \alpha_m(s,a) \in [0,1] \quad \sum_{n=0}^{\infty} \alpha_n(s,a) = \infty \quad \sum_{n=0}^{\infty} \alpha_n(s,a)^2 \leq \infty.$$

The first condition directly affects the definition of our learning policy $\theta(Q^*, s)$, since it is obliged to permit access to all the possible actions.

The trivial example that meets the conditions would be to take $\alpha_m(s,a) = \frac{1}{m}$ and the learning policy $\theta(Q^*, s) = random\{a \in A\}$, that is, the policy that selects the action at random. This learning policy is purely exploratory, however it will converge very slowly. A more commonly used policy is the so-called ϵ-Greedy that for each pair (Q^*, s) selects the action $a = argmax_{a \in A} Q^*(s,a)$ with probability $1 - \epsilon$, and any other random action with probability ϵ.This policy favors following the most fruitful actions founded so far, but still meets the conditions and allows exploration.

Reinforcement learning algorithms whose update value is correlated with the learning policy will be called on-policy algorithms. SARSA, presented in **Algorithm 4** below, is a transformation of Q-learning to an on-policy algorithm.

Algorithm 4 SARSA-learning algorithm

We initialize all $Q_0(s,a) = 0$
$Q \longleftarrow Q_0$
for t = 0 : T **do**
  Select $s \in S$ randomly
  Select $a \in A$ using $\theta_{n_s}(Q, s)$ learning policy.
  for each step of t epoch **do**
    Let $r(s,a)$ be the reward obtained
    Let s' next state of the enviroment
    Select $a' \in A$ using $\theta_{n_{s'}}(Q, s')$ learning policy
    $Q(s,a) \leftarrow Q(s,a) + \alpha_m(s,a)[r + \gamma Q(s',a') - Q(s,a)]$
    $s \leftarrow s'$
    $a \leftarrow a'$
  end for
end for
$\pi(s) = argmax_{a \in A} Q^*(s,a)$
  return π

This algorithm will present convergence conditions almost identical to Q-learning, although it also requires the learning policy to be limiting greedy, that is, when taking a limit with

respect to the number of times the algorithm has passed through s, it must meet:

$$\forall s \in S \; lim_{n\to\infty}\theta_n(Q,s) = argmax_{a\in A}Q(s,a).$$

We will conclude this section by noting that as the dimensions of the problem increase, these algorithms will be computationally more efficient than those of dynamic programming, since it avoids the use of transition matrices and inverses.

2.3. Temporal differences learning

So far, we have seen how to solve MDPs by dynamic programming, as well as using stochastic approximation algorithms to overcome the modeling curse in the case where we do not know explicitly the enviroment of our problem or where it is too computationally expensive to use the transition matrices.

However, we can still run into another difficulty. If our MDP turns out to have a huge number of possible states or actions, stochastic approximation methods such as Q learning or SARSA will become too computationally expensive and will fail. This phenomenon is known as the curse of dimensionality. . To overcome this type of problems, temporal difference (TD) learning methods have been used as a tool for resolving MDPs.

In the following sections we will consolidate the basic concepts of TD learning and we will present its use in a widely recognised decision taking problem [22], unsolvable by the previously presented methods due to the high dimensionality of the problem.

The temporal difference (TD) method represents a branch of Machine Learning used for so called multi-step problems. In these problems, a series of events or 'steps' take place (with a time order) until reaching the point where the result we sought to predict is obtained. The TD model will compare the predictions in one step with the next and thus correct their predictions successively. A classic example to visualize this type of problem is Sutton's 'weatherman problem' [17] where a meteorologist tries to predict the weather on Sunday throughout the week.

Until the mid-1980s and early 2000s, these types of problems were solved using supervised learning. Going back to the 'weatherman problem', the supervised approach to this problem would be to generate a series of pairs prediction-result ('(Monday,Sunday)', '(Tuesday, Sunday)'...) , and adjust the predictions by some sort of regression. Despite obtaining partially satisfactory results, Sutton and Barto, among others[17] [18] [19], showed that alternative learning methods were computationally more efficient and even learned faster, that is, they converged faster to better results than the supervised approach.

Next, we will introduce the TD methods following the approach used by Sutton [17], and we will state the theorem that guarantees the convergence of this method for some particular conditions. Although the conditions are certainly very restrictive, this was the first theorem to prove a convergence for TD methods. This result allowed to consolidate the intuition that these methods effectively solved learning problems.

2.3.1. TD introduction and Sutton's theorem.

We will start by introducing a multi-step problem. These problems consist of sequences of elements $x_1, x_2, ... , x_t, z$ where each of the values x represents the temporally ordered steps of the sequence, and z represents the result obtained at the end of it.For this type of problems we will normally get a considerable number of sequences. Each element of a sequence will be associated with a prediction $P_t = P(x_t, w)$ of the corresponding z, dependent on the event x_t and some weights w, common in all the predictions. The learning goal is for the P_t to reasonable predict the value of z (more formally $\mathbb{E}[z \mid x_t]$) at the end of all the sequence.

The predictor function P can be anything from a linear function to a deep neural network. Therefore, the learning will consist of updating the weights w through increments Δw_t. Normally they will be applied at the end of each sequence, as indicated by the following expression:

$$w \longleftarrow w + \sum_{t=1}^{m} \Delta w_t. \tag{2.6}$$

If we take the supervised learning approach, we can state the problem as a series of state-end of sequence pairs (x_1, z),....,(x_t, z), noting that with this representation we have omitted the succeeding character of the sequence. In this approach, the increments or updates of w are formulated in the backpropagation way, first presented by Rumelhart [15]

$$\Delta w_t = \alpha(z - P_t)\nabla_w P_t, \tag{2.7}$$

where α represents a coefficient of variation and $\nabla_w P_t$ is the gradient of the function $P_w(x) = P(x, w)$ evaluated at x_t.

Therefore, we can rewrite the update of w for each sequence as

$$w \longleftarrow \sum_{t=1}^{m} \alpha(z - P_t)\nabla_w P_t. \tag{2.8}$$

Moreover, if we rewrite $z - P_1 = \sum_{t=1}^{m}(P_{t+1} - P_t)$ where $P_{m+1} = z$ we end up getting the following expression for Δw_t:

$$\Delta w_t = \alpha(P_{t+1} - P_t) \sum_{k=1}^{t} \nabla_w P_k. \tag{2.9}$$

This way of computing the increments is much more time and storing efficient than the previous one. We remind the reader that if we have a series of ordered sequences, and we are traversing it to update the w, for each sequence it is needed to compute $\sum_{t=1}^{m} \Delta w_t$. Once this expression is obtained, w will be updated, the information stored to calculate the sum of increments deleted and we will move on to the next sequence.

In the first case (2.7), it will be needed to store the values of P_t and $\nabla_w P_t$ for all the elements of the sequence, and only once we reach the end of said sequence will we be

able to perform all the addition of increments. On the other hand, in the second way (2.9) increments may be calculated and added step by step in the sequence, since for each Λw_t it is only necessary the sum of the previous gradients and the approximation of the next step. Therefore, we only need to store the sum of the above gradients $\sum_{k=1}^{t} \nabla_w P_k$ and the relevant approximation.

The last case will be called TD(1), and it will be part of the family of TD methods. TD(1) updates the weights of w in an identical way to supervised learning, but in a much more computationally efficient fashion. Given this, the TD methods are introduced as those methods that update the weights of w in the way initially mentioned, but their increments are now the following :

$$\Delta w_t = \alpha(P_{t+1} - P_t) \sum_{k=1}^{t} \lambda^{t-k} \nabla_w P_k \quad where \;\; \lambda \in [0, 1]. \tag{2.10}$$

As in the previous TD(1) case, we will be able to add the increments while traversing the sequence. Furthermore, it is only needed to store the weighted sum of the previous gradients and the relevant state, since it can be seen that

$$\sum_{k=1}^{t+1} \lambda^{t-k} \nabla_w P_k = \nabla_w P_{t+1} + \lambda \sum_{k=1}^{t} \lambda^{t-k} \nabla_w P_k.$$

Finally, we will introduce the absorbing Markov chains concept, since in order to apply the convergence theorem that Sutton presented, the problem must be able to be rewritten as sequences of a chain of this type. An absorbing Markov chain is a Markov chain where there exists at least one absorbing state, that is, a state that cannot be left.All states must be capable of reaching an absorbing state in a finite number of steps. Consequently, if we are traversing the chain we will eventually reach one of these states and restart the chain.

Moreover, for our particular problem, when we reach each of the absorbing states we will receive a reward z that will only depend on said state. Essentially, we require the multi-state problem to have no memory. Let us now continue and end the section with the statement of the theorem.

Theorem 2.8. *Let be a stationary and absorbing Markov chain, with any distribution of initial probabilities μ_i, and expected values of the rewards associated with each absorbing state $\hat{z}_j$, where the states are represented as linearly independent vectors $\{x_i \mid i \in S\}$.Then there exists $\epsilon > 0$ such that for all $\alpha < \epsilon$ the predictions of the method TD(0) where P_w is a linear function converge on expected value to the ideal predictions. That is, $lim_{n \longrightarrow \infty} \mathbb{E}[P(x_i, w_n)] = \mathbb{E}[z \mid x_i] \;\; \forall i \in S$, where n is the number of sequences.*

2.4. The Tesauro's backgammon problem. Overcoming the curse of dimensionality

Backgammon is a classic board game where two players compete to remove all their pieces from the board before the rival. At each turn, one of the players rolls two dice, and depending on the result is able to move his pieces in different ways. Note that the next state on

the board only depends on the current state, so we are faced with a Markovian situation. If certain reasonable conditions are assumed and the game is approached from the perspective of a single player we can consider maximizing game-play in backgammon as a MDP. Lets see how:

Let the set of states S be all the possible combinations of distributions of pieces and dice that can exist in the game. Let the set of actions A_s be all the possible moves that the player can make if he finds himself in the position s. To assume the existence of the probability distribution $P(s' \mid (s, a))$ is needed to supose that the player is going against a consistent player or group of players. That is to say, a player that for each distribution of pieces that he receives and each roll of the dice that he obtains will always make the same movement (or set of movements, if we want to further complicate the problem). Finally, the rewards will be 0 for any pair (s, a) where both players continue to have chips on the board, 1 if only the rival has chips, and -1 when the opponent has no chips. Note that for these last two cases $P(0 \mid (s, a)) = 1$,that is, the game is restarted and the next state is the starting position.

Given that despite the fact that the rival player is consistent we do not know his game strategy, the transition matrices are unknown and therefore we can only solve the problem using stochastic approximations. We have seen that this type of technique converges theoretically when infinite iterations of the algorithm go through each state-action value $Q(s, a)$ an infinite number of times. Therefore, in order to obtain reasonable results in practice, we need to iterate our algorithm enough to reach each tuple a minimum of times. In this problem we have of the order of 10^{22} tuples, which rules out any possibility of using these techniques.In conclusion, new ways of approaching and solving the problem were needed.

Therefore, to solve this problem, Tesauro tried new methods [21] [22] derived form TD learning techniques, although he specified in [21] that there was no theoretical framework that guaranteed good results of this method for this problem. To begin with, the convergence results are postulated for prediction problems, when the problem as he posed it is a decision or prediction-decision problem (we can conclude the latter considering that if the player is able to predict the chances of winning in each position, he can always choose advantageous positions).

On the other hand, the proof of convergence in the TD(0) methods requires that the states be represented as linearly independent vectors, an impossible task given that he had on the order of 10^{22} states. Furthermore, Tesauro used deep neural networks and not linear functions as P, and he was not limited only to TD(0), but worked with any value of λ.

Despite all the arguments and warnings stated above, Tesauro tried to solve the problem with the following TD scheme: Let X be the set of all possible distributions of tiles on the board, where the states were represented as $x \in M(\mathbb{R})_{28\times 8}$. Let P be the function that approximates the value of the outcome of the sequence for each x and $z = 1$ the outcome if the first player wins, while $z = 0$ is obtained if the second player rises with victory. To start the algorithm, the weights of the function $P(x_t, w)$ will be initialized in a random way.

Let us note that by representing the states in a dependent way with a matrix structure that resembled the board, Tesauro sought that the algorithm could transfer the learning of some positions to other similar ones.This approach would avoid the collapse of the algorithm by not having to go through all the positions.

Being already stated the variables in play, let the game begin. A sequence will originate at x_0, the starting position, and a dice roll corresponding to the first player's turn. He will move to the new position x_1 allowed by the dice that maximizes P, since this function is an approximation of the average value of z at the end of the sequence, that is, the first player's chances of winning from that position. Now, the dice will be rolled and the second player will move, this time to the allowed position (x_2) that minimizes P, that is, to the position that according to the TD approximation minimizes the first player's chances of winning. They will keep moving successively until a player wins (reaches an ending situation x_t). Once this occurs, the weights w will be updated in a TD fashion (2.10) and a new game will start.

The objective of Tesauro was to make an algorithm that learns to play Backgammon only by playing against itself, that is, a self-taught algorithm, capable of learning to compete in the game only with the rules and without any previous experience, contrary to all machine learning methods based on supervised learning.It was an attempt on taking the learning even further.

The results achieved were phenomenal. The algorithm only playing against itself and without previous experience was able to reach an intermediate level of play, capable of beating any commercial supervised learning software trained with results from thousands of games. Furthermore, by introducing certain build in features to this algorithm, such as known game strategies for certain positions, it was able to bring its performance to that of a master, winning 13 games out of 31 games against the world champion.

The resounding success of Tesauro opened the door to the application of TD methods to other complex decision-making problems, such as many other MDPs of intractable dimensions.

3. The Hidden Markov model

This section introduces hidden Markov models, an inexpensive and intuitive method for modeling stochastic processes. The following part presents the motivation behind the technique with a simple example, thus showing its usefulness in our day to day.

The following sections explain the details of this approach following the development of the book by Rabiner [13]. First, Hidden Markov models will be presented as theoretical entities, and it will be shown how the state of a model can be estimated from the model definition and a history of observations, as well as how a model can be estimated from some observations. Second, the implementation of an HMM will be described, including an optimization into a sequence of simple operations to make it computationally efficient.

Finally, some of its applications will be shown, such as speech recognition (including Siri) or the analysis of biological sequences, particularly DNA.

3.1. A simple example

Imagine that we have a friend who lives far away and with whom we talk daily on the phone about what he did during the day. Our friend is interested in three activities: walking around the square, go shopping and cleaning his apartment. The activities that our friend is going to do depend on the weather.

To simplify the problem we are going to consider only two kinds of weathers, "rainy" and "sunny". Suppose that, by the number of times we have gone to visit him, the probability that a rainy day follows a rainy day is 0.7, and the probability that a sunny day follows a sunny day is 0.6. If we assume that these probabilities continue to hold, this information can be written as

	R	*S*
R	0.7	0.3
S	0.4	0.6

Tab. 1

where *R* is "rainy" and *S* is "sunny".

As we have said before, the activities that our friend is going to do depend on the weather. Suppose that this relationship is given by

	Walk	Shop	Clean
R	0.1	0.4	0.5
S	0.7	0.2	0.1

Tab. 2

where, for example, 0.1 is the probability that on a rainy day our friend will walk.

For this model, if we consider the *state* as the weather of the day, it is a *Markov process*, since the next state depends only on the current state, however, these states are "hidden" since we can't observe the weather of the day.

However, we can observe what activity our friend does every day. Thus, by Table 2, these data give us probabilistic information about the weather.

Since the states are hidden, this type of model is known as a *Hidden Markov model* (HMM). Our goal is to make use of the observable information, in this case the activities

that our friend does, to obtain information about the Markov process.

In this way we are going to present the *discrete Hidden Markov model* of this example. For this we have the *set of states* given by $Q = \{R, S\}$, and the *set of possible observations* given by $V = \{0, 1, 2\}$, where 0 represents "walk", 1 represents "shop" and 2 represents "clean".

By Table 1, the *transition probability matrix* is given by

$$A = \begin{bmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{bmatrix}, \tag{3.1}$$

and by Table 2, the *observation probability matrix* is given by

$$B = \begin{bmatrix} 0.1 & 0.4 & 0.5 \\ 0.7 & 0.2 & 0.1 \end{bmatrix}. \tag{3.2}$$

In this example we are going to assume that the probability that the initial day will be a rainy day is 0.6, while the probability that it will be a sunny day is 0.4. Thus, we have the *initial state distribution* as

$$\pi = \begin{bmatrix} 0.6 & 0.4 \end{bmatrix}. \tag{3.3}$$

As we can see, these three matrices (π, A, B) are stochastic by rows, that is, each row is a probability distribution.

Now let us consider a period of four days that we want to study, for which the sequence of activities (W, S, W, C) has been observed, therefore the *sequence of observations* is given by

$$O = (0, 1, 0, 2). \tag{3.4}$$

Given this model, we can question whether we could know the most probable weathers during the four-days period of interest, that is, whether we could determine the state sequence that maximizes the expected number of correct states given the observations 3.4 (*HMM probabilities*).

In the following sections, we will explain in detail the Hidden Markov model, as well as its notation. Then we will present the three fundamental problems, solving these and giving some efficient algorithms. (...)

3.2. Formal definition of a Hidden Markov model

A *Hidden Markov model* (HMM) is a statistical model in which it is assumed that the system to be modeled is a Markov process with unknown parameters ("hidden"). The objective is to determine the unobservable parameters of said chain from the observable parameters.

The HMM can perfectly be continuous, however, in this section we will be assumed to be *discrete*. To see the continuous HMM we recommend reading [10].

Definition 3.1. *A common notation for an HMM is the representation as a tuple* (Q, V, π, A, B, O) *where*

- Q *is the set of states of the Markov process,*
  $Q = \{q_0, q_1, \ldots, q_{N-1}\}$ *where* N *is the number of states in the model.*
- V *is the set of possible observations ,*
  $V = \{0, 1, \ldots, M-1\}$ *where* M *is the number of observations symbols.*
- π *is the initial state distribution,*
  $\pi = \{\pi_i\}$ *where* π_i *is the probability that the first state is state* q_i*.*
- A *is the transition probability matrix of dimension* $N \times N$*,*
  $A = \{a_{i,j}\}$ *with* $a_{i,j} = \mathbb{P}(\text{state } q_j \text{ at } t+1 | \text{state } q_i \text{ at } t)$*.*
- B *is the observation probability matrix of dimension* $N \times M$*,*
  $B = \{b_{j,k}\} = \{b_j(k)\}$ *with* $b_j(k) = \mathbb{P}(\text{observation } k \text{ at } t | \text{state } q_j \text{ at } t)$*.*
- O *is the observation sequence,*
  $O = (O_0, O_1, \ldots, O_{T-1})$ *where* T *the length of the observation sequence.*

The HMM is denoted by $\lambda = (A, B, \pi)$*.*

A generic hidden Markov model is shown in Figure 1, where $\{X_i\}$ represents the hidden state sequence and all other notations are as above. The Markov process, which is hidden above the dashed line, is determined by the current state X_0 and matrix A. We can only observe the O_i, which are related to the states of the Markov process hidden by matrix B.

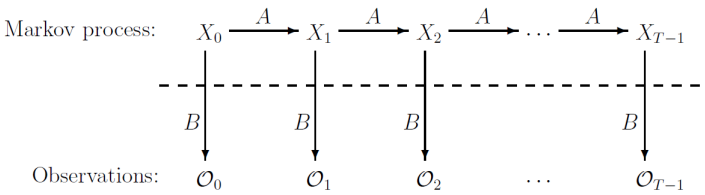


Fig. 1. Hidden Markov model.

For our example, we have $N = 2$, $M = 3$ and $T = 4$. In this case, A, B, π and O are given by 3.1, 3.2, 3.3 and 3.4, respectively.

So let's consider a generic state sequence of length four

$$X = (X_0, X_1, X_2, X_3)$$

with corresponding observations

$$O = (O_0, O_1, O_2, O_3).$$

We know that π_{X_0} is the probability of starting in state X_0, that $b_{X_0}(O_0)$ is the probability of starting observing O_0 given the state X_0, and that a_{X_0,X_1} is the probability of transiting from state X_0 to state X_1. Continuing, we see that the probability of the state sequence X is given by

$$\mathbb{P}(X, O) = \pi_{X_0} b_{X_0}(O_0) a_{X_0,X_1} b_{X_1}(O_1) a_{X_1,X_2} b_{X_2}(O_2) a_{X_2,X_3} b_{X_3}(O_3). \tag{3.5}$$

Continuing with the example with observation sequence $O = (0, 1, 0, 2)$, if we consider the sequence of states $X = (R, S, S, R)$, using (7) we have

$$\mathbb{P}((R, S, S, R), (0, 1, 0, 2)) = 0.6(0.1)(0.3)(0.2)(0.6)(0.7)(0.4)(0.5) = 0.000302.$$

In this way, we can calculate the probabilities of all sequences of states. We have listed these results in Table 2, where the normalized probabilities have also been calculated (the sum is equal to 1)

state	probability	normalized probability
$RRRR$	.000412	.042787
$RRRS$	.000035	.003635
$RRSR$	.000706	.073320
$RRSS$	.000212	.022017
$RSRR$	.000050	.005193
$RSRS$	.000004	.000415
$RSSR$	.000302	.031364
$RSSS$	.000091	.009451
$SRRR$	.001098	.114031
$SRRS$	.000094	.009762
$SRSR$	.001882	.195451
$SRSS$	.000564	.058573
$SSRR$	.000470	.048811
$SSRS$	.000040	.004154
$SSSR$	.002822	.293073
$SSSS$	.000847	.087963

Fig. 2. State sequence probabilities.

It can be seen that the sequence with the highest probability is $SSSR$, but this is the optimal sequence in the dynamic programming sense (explained in the previous section), not in the HMM sense. To find the optimal sequence in the HMM sense, we choose the most probable symbol at each position. To this end we sum the probabilities in Table 1 that have an R in the first position. Doing so, we find the normalized probability of R in the first position is 0.18817 and hence the probability of S in the first position is 0.81183. The HMM therefore chooses the first element of the optimal sequence to be S. We repeat this for each element of the sequence, obtaining the probabilities in Table 3.

	element			
	0	1	2	3
$\mathbb{P}(R)$	0.188182	0.519576	0.228788	0.804029
$\mathbb{P}(S)$	0.811818	0.480424	0.771212	0.195971

Fig. 3. HMM probabilities.

As we can see, the optimal sequence in the HMM sense of this example is $SRSR$, which does not match the sequence with the highest probability, but with the sequence of states that maximizes the expected number of correct states given the observations 3.4.

3.3. The three problems

Problem 1. *Given the model $\lambda = (A, B, \pi)$ and a sequence of observations O, find $\mathbb{P}(O|\lambda)$. Here, we want to determine a score for the observed sequence O with respect to the given model λ.*

Solution. Let $\lambda = (A, B, \pi)$ be a HMM and let $O = (O_0, O_1, \ldots, O_{T-1})$ be a observation sequence.

Let $X = (X_0, X_1, \ldots, X_{T-1})$ be a state sequence. Then by definition of B we have

$$\mathbb{P}(O|X, \lambda) = b_{X_0}(O_0) b_{X_1}(O_1) \cdots b_{X_{T-1}}(O_{T-1}) \tag{3.6}$$

and by the definition of π and A it follows that

$$\mathbb{P}(X|\lambda) = \pi_{X_0} a_{X_0,X_1} a_{X_1,X_2} \cdots a_{X_{T-2},X_{T-1}}. \tag{3.7}$$

Since

$$\mathbb{P}(O, X|\lambda) = \frac{\mathbb{P}(O \cap X \cap \lambda)}{\mathbb{P}(\lambda)}$$

then

$$\mathbb{P}(O, X|\lambda) = \frac{\mathbb{P}(O \cap X \cap \lambda)}{\mathbb{P}(\lambda)} \cdot \frac{\mathbb{P}(X \cap \lambda)}{\mathbb{P}(X \cap \lambda)}$$

$$= \frac{\mathbb{P}(O \cap X \cap \lambda)}{\mathbb{P}(X \cap \lambda)} \cdot \frac{\mathbb{P}(X \cap \lambda)}{\mathbb{P}(\lambda)} = \mathbb{P}(O|X, \lambda) \cdot \mathbb{P}(X|\lambda).$$

Thus, by 3.6 and by 3.7 we obtain

$$\mathbb{P}(O|\lambda) = \sum_X \mathbb{P}(O, X|\lambda) = \sum_X \mathbb{P}(O|X, \lambda) \cdot \mathbb{P}(X|\lambda)$$

$$= \sum_X \pi_{X_0} b_{X_0}(O_0) a_{X_0,X_1} b_{X_1}(O_1) a_{X_1,X_2} \cdots a_{X_{T-2},X_{T-1}} b_{X_{T-1}}(O_{T-1}). \tag{3.8}$$

■

Calculation of $\mathbb{P}(O|\lambda)$ as shown is impractical, since $2TN^T - 1$ operations are required. This means that for a model with only 10 states and 10 observations, 10^{11} operations are needed. To reduce this complexity, some algorithms such as forward algorithm or backward algorithm are used which we are going to show below.

Algorithm 1 (*Forward algorithm (α-pass)*). For $t = 0, 1, \ldots, T-1$ and $i = 0, 1, \ldots N-1$, define

$$\alpha_t(i) = \mathbb{P}(O_0, O_1, \ldots, O_t, X_t = q_i|\lambda). \tag{3.9}$$

Given the model λ, $\alpha_t(i)$, also called *forward probability*, is the probability of observing $O_0, O_1, \ldots, O_t$ and be in the state q_i at time t.

For the forward calculation of the probability of a sequence of observations, $\alpha_t(i)$ can be computed recursively as follows.

(1) Initialization:
Let $\alpha_0(i) = \pi_i b_i(O_0)$, for $i = 0, 1, \ldots N-1$.

(2) Recurrence:
For $t = 1, \ldots, T-1$ and $i = 0, 1, \ldots N-1$, compute

$$\alpha_t(i) = \left[\sum_{j=0}^{N-1} a_{ij}\alpha_{t-1}(j)\right] b_i(O_t).$$

(3) Termination:
For 3.8 and 3.9 it's enough to calculate

$$\mathbb{P}(O|\lambda) = \sum_{i=0}^{N-1} \alpha_{T-1}(i).$$

Algorithm 2 (*Backward algorithm (β-pass)*). For $t = 0, 1, \ldots, T-1$ and $i = 0, 1, \ldots N-1$, define

$$\beta_t(i) = \mathbb{P}(O_{t+1}, O_{t+2}, \ldots, O_{T-1}, X_t = q_i|\lambda). \tag{3.10}$$

Given the model λ, $\beta_t(i)$, also called *backward probability*, is the probability of observing $O_{t+1}, O_{t+2}, \ldots, O_{T-1}$ (from time instant $t+1$ to the end) and be in the state q_i at time t.

For the backward calculation of the probability of a sequence of observations, $\beta_t(i)$ can be computed recursively as follows.

(1) Initialization:
Let $\beta_{T-1}(i) = 1$, for $i = 0, 1, \ldots N-1$.

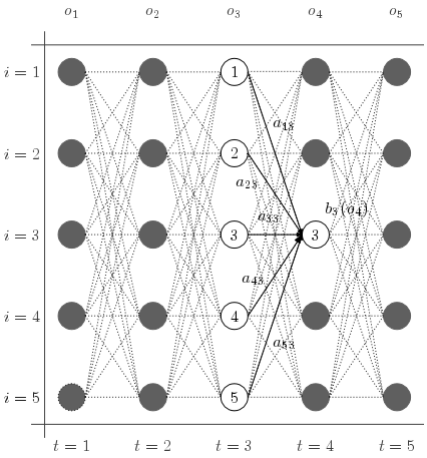
(2) Recurrence:
For $t = T-2, T-3, \ldots, 0$ and $i = 0, 1, \ldots N-1$, compute

$$\beta_t(i) = \sum_{j=0}^{N-1} a_{ij}\beta_{t+1}(j)b_j(O_{t+1}).$$

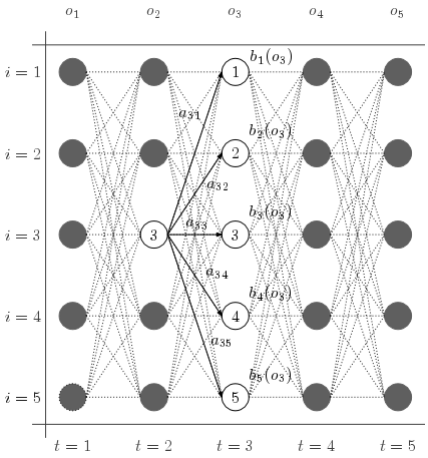
(3) Termination:
For 3.8 and 3.10 it's enough to calculate

$$\mathbb{P}(O|\lambda) = \sum_{i=0}^{N-1} \beta_0(i)\pi_i b_i(O_0).$$

The following schemes shows the states and probabilities necessary to calculate $\alpha_4(3)$ and $\beta_b(3)$ for a model with 5 states and a sequence of observations of length 5.



(a) Calculation of $\alpha_4(3)$



(b) Calculation of $\beta_b(3)$

Both forward algorithm and backward algorithm require on the order of N^2T operations, much less than the $2TN^T - 1$ operations that are necessary if $\mathbb{P}(O, X|\lambda)$ is calculated for all possible states sequences X of the model.

Problem 2. *Given $\lambda = (A, B, \pi)$ and an observation sequence O, find an optimal state sequence for the underlying Markov process. In other words, we want to uncover the hidden part of the Hidden Markov Model. This type of problem is discussed in some detail in the example, above.*

Solution. Given $\lambda = (A, B, \pi)$ and an observation sequence O, the optimal state sequence for the underlying Markov process is given by

$$\arg\max_X \mathbb{P}(X|O, \lambda) = \arg\max_X \mathbb{P}(X|O)\mathbb{P}(\lambda) = \arg\max_X \frac{\mathbb{P}(X, O)}{\mathbb{P}(O)}\mathbb{P}(\lambda)$$

and since O is fixed

$$\arg\max_X \mathbb{P}(X|O, \lambda) = \arg\max_X \mathbb{P}(X, O)\mathbb{P}(\lambda) = \arg\max_X \mathbb{P}(X, O, \lambda). \tag{3.11}$$

■

This task requires finding a maximum over all possible state sequences, and can be solved efficiently by the Viterbi algorithm.

Algorithm 3 (*Viterbi algorithm*). For $t = 0, 1, \ldots, T-1$ and $j = 0, 1, \ldots N-1$, define

$$\delta_t(j) = \max_{X_0,\ldots,X_t} \mathbb{P}(X_0, X_1, \ldots, X_t = q_j, O_0, O_1, \ldots, O_t, \lambda) \tag{3.12}$$

and

$$\varphi_t(j) = \arg\max_{X_t} \mathbb{P}(X_0, X_1, \ldots, X_t = q_j, O_0, O_1, \ldots, O_t, \lambda). \tag{3.13}$$

Given the model λ and an observation sequence O, $\delta_t(j)$ is highest probability of any sequence reaching state q_j at time t after emitting $O_0, O_1, \ldots, O_t$, and $\varphi_t(j)$ is the last state (X_t) in highest probability sequence reaching state q_j at time t after emitting $O_0, O_1, \ldots, O_t$.

For the Viterbi calculation, we will calculate the best sequence with the same recursive approach as in forward and backward algorithms (algorithms 1 and 2).

(1) Initialization:
For $j = 0, 1, \ldots N-1$, let

$$\delta_0(j) = \pi_j b_j(O_0)$$

and

$$\varphi_0(j) = 0.$$

(2) Recurrence:
For $t = 1, \ldots, T-1$ and $j = 0, 1, \ldots N-1$, compute

$$\delta_t(j) = \max_{0 \le i \le N-1} \delta_{t-1}(i) a_{ij} b_j(O_t)$$

and

$$\varphi_t(j) = \arg\max_{0 \le i \le N-1} \delta_{t-1}(i) a_{ij}.$$

(3) Termination:
First, compute the optimal last state

$$\hat{X}_{T-1} = \arg\max_{0 \le i \le N-1} \delta_{T-1}(i).$$

Then, for $t = 0, \ldots, T-2$, compute

$$\hat{X}_t = \varphi_{t+1}(\hat{X}_{t+1}),$$

where $\hat{X} = \{\hat{X}_t\}$ is the optimal state sequence in the HMM sense.
Finally, compute this probability

$$P(\hat{X}) = \max_{0 \le i \le N-1} \delta_{T-1}(i).$$

The following scheme shows an example of the most probable states sequence for a model with 5 states and a sequence of observations of length 5.

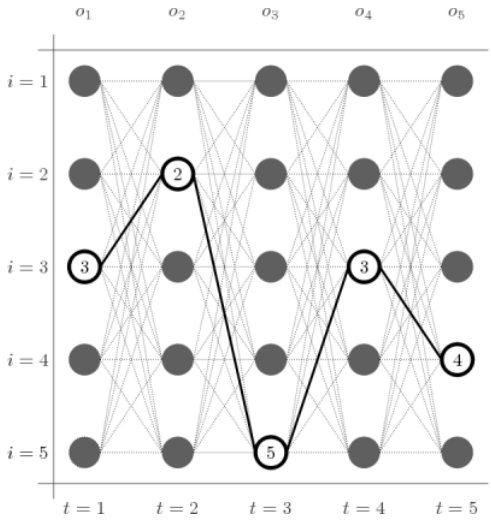


Fig. 5. Example of the most probable states sequence

This algorithm require on the order of N^2T operations, same as forward and backward algorithms.

In practice, many times this algorithm is used in *log-space*. That is because in the models, usually, the number of observations is very large, therefore, the individual probabilities are very small, that is, the products of such probabilities will be very very very small. In such cases, our machines may run out of precision, hence yielding unreliable computations. Working in the log-space is a common "trick" that solves the problem.

Problem 3. *Given an observation sequence O and the dimensions N and M, find the model $\lambda = (A, B, \pi)$ that maximizes the probability of O. This can be viewed as training a model to best fit the observed data.*

Mathematically, this problem can be written as

$$\max_{\lambda} \quad \mathbb{P}(O|\lambda)$$

$$\text{s.t.} \quad O, N, M.$$

Unfortunately, it is not possible to find such a model analytically and therefore an iterative algorithm such as Baum and Welch's is necessary, which allows estimating the parameters of a model that maximize the probability of an observations sequence.

Algorithm 4 (*Baum and Welch algorithm*). For $t = 0, 1, \ldots, T-2$ and $i, j \in \{0, 1, \ldots N-1\}$, define

$$\gamma_t(i) = \mathbb{P}(X_t = q_i | O, \lambda) = \frac{\mathbb{P}(X_t = q_i, O, \lambda)}{\mathbb{P}(O, \lambda)} = \frac{\alpha_t(i)\beta_t(i)}{\sum_{k=0}^{N-1} \alpha_t(k)\beta_t(k)} \tag{3.14}$$

and

$$\begin{aligned}\xi_t(i,j) = \mathbb{P}(X_t = q_i, X_{t+1} = q_j | O, \lambda) &= \frac{\mathbb{P}(X_t = q_i, X_{t+1} = q_j, O, \lambda)}{\mathbb{P}(O, \lambda)} \\ &= \frac{\alpha_t(i)a_{ij}b_j(O_{t+1})\beta_{t+1}(j)}{\sum_{k=0}^{N-1} \alpha_t(k)\beta_t(k)}.\end{aligned} \tag{3.15}$$

where $\alpha_t(i)$ is the forward probability, 3.9, and $\beta_t(i)$ is the backward probability, 3.10.

Given the observation sequence O and the dimensions N and M, $\gamma_t(i)$ is the probability of being at state q_i at time t, and $\xi_t(i,j)$ is the probability of moving from state q_i at time t to state q_j at time $t+1$. Note that

$$\gamma_t(i) = \sum_{j=0}^{N-1} \xi_t(i,j). \tag{3.16}$$

Re-estimation is an iterative process. First, we initialize $\lambda = (A, B, \pi)$ with a best guess or, if no reasonable guess is available, we choose random values such that $\pi_i \approx \frac{1}{N}$ and $a_{ij} \approx \frac{1}{N}$ and $b_j(k) \approx \frac{1}{M}$. It's critical that π, A and B be randomized, since exactly uniform values will result in a local maximum from which the model cannot climb. As always, π, A and B must be row stochastic.

The solution to Problem 3 can be summarized as follows.

(1) Initialization:
For $j = 0, 1, \ldots N-1$, let

$$\delta_0(j) = \pi_j b_j(O_0)$$

and

$$\varphi_0(j) = 0.$$

(2) Recurrence:
For $i, j \in \{0, 1, \ldots N-1\}$ and $j = 0, 1, \ldots, M-1$, compute

$$\hat{\pi}_i = \frac{\text{Expected frequency in state } q_i \text{ at time } t=0}{} = \gamma_0(i), \tag{3.17}$$

$$\hat{a}_{i,j} = \frac{\text{Expected number of transitions from } q_i \text{ to } q_j}{\text{Expected number of transitions from } q_i} = \frac{\sum_{t=0}^{T-2} \xi_t(i,j)}{\sum_{t=0}^{T-2} \gamma_t(i)} \tag{3.18}$$

and

$$\hat{b}_{j,k} = \frac{\text{Expected number of emissions of } k \text{ from } q_j}{\text{Expected number of visits to } q_j} = \frac{\sum\limits_{\substack{\{t:0\le t\le T-1,\\ O_t=k\}}} \gamma_t(j)}{\sum\limits_{t=0}^{T-1} \gamma_t(j)}. \tag{3.19}$$

(3) Termination:
Re-estimate the model $\lambda = (A, B, \pi)$ with the $\hat{\pi}_i$, $\hat{a}_{i,j}$ and $\hat{b}_{j,k}$ calculated in step (2). Then, calculate $\mathbb{P}(O|\lambda)$ with the forward or backward algorithm (algorithms 1 and 2). If this value increases, go to step (1), and if not, the iteration has finished.
In practice, a threshold and/or a maximum number of iterations is set, and if $\mathbb{P}(O|\lambda)$ does not increase by at least that threshold and/or reaches that maximum number of iterations, the iteration stops.

The following scheme shows a partial diagram of the elements necessary for the calculation of $\xi_3(2,4)$ for a model with 5 states and a sequence of observations of length 5.

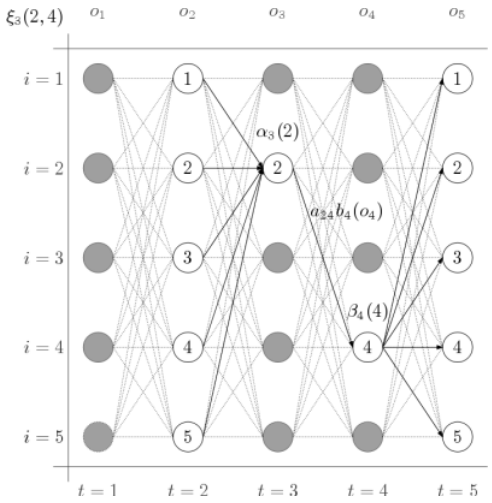


Fig. 6. Calculation of $\xi_3(2,4)$

This algorithm is the most used to solve this problem, although does not guarantee a global maximum, but a local maximum.

3.4. Applications of Hidden Markov models

Unfortunately, a comprehensive book devoted to Hidden Markov models does not yet exist. There are, however, several books intended for a reader with a specific background. The

most famous areas of HMM application are speech recognition and bioinformatics, and books devoted to these research areas often have chapters covering HMM. It is interesting to note that speech recognition uses continuous HMMs, but bioinformatics uses discrete HMMs for gene recognition and representation of protein families.

Consider the problem of speech recognition (which just happens to be one of the best-known applications of HMMs). We can use the solution to Problem 3 to train an HMM, say, λ_0 to recognize the spoken word "no" and train another HMM, say, λ_1 to recognize the spoken word "yes". Then given an unknown spoken word, we can use the solution to Problem 1 to score this word against λ_0 and also against λ_1 to determine whether it is more likely "no", "yes", or neither. In this case, we don't need to solve Problem 2, but it is possible that such a solution (which uncovers the hidden states) might provide additional insight into the underlying speech model. To know this application in detail we recommend the book by Rabiner [13].

Now, if we consider the problem of gene recognition, We can use the solution to Problem 3 to train the HMM model given some observations of genes of a similar species so that it can capture the characteristics of the genes of the same species to maximize the chances of detecting the genes of the studied species. After the model is trained, each of the genes found in the genome must be evaluated and their probability calculated (Problem 1). With said probability and the gene, post-processing is applied in order to eliminate those genes that, according to the knowledge of the model, are not possible genes. To know this application in detail we recommend the article [16].

To finish this section, we show you other of the many applications that HMMs have, as well as some articles that show these applications:

- Speech synthesis, [23].
- Machine translation, [8].
- Cryptanalysis, [4].
- Neuroscience, [2].
- Computational finance, [6].

4. Random walks

The random walk can be defined in simple terms as a stochastic process with the property of being studied from individual steps given by a specific distribution. The random walks can perfectly be continuous, however, in this section they will be assumed to be discrete and therefore there will be N discrete points where $N \geq 0$. The random walk can be derived from a specific Markov Chain as the following defintion states.

Definition 4.1 (A sequence of random variables $X_0, X_1, ... \subseteq \omega$ is a Markov chain with state space ω if for all possible states $x_{t+1}, x_t, \ldots, x_1$ it is given).

$$\mathbb{P}[X_{t+1} = x_{t+1} | X_t = x_t \wedge X_{t-1} = x_{t-1} \wedge \cdots \wedge X_0 = x_0] = \mathbb{P}[X_{t+1} = x_{t+1} | X_t]. \tag{4.1}$$

The markov chain would be refered as time-homogenous if the possibility is also independent of t.

So, given 4.1, we can define the random walk by stating that it will start at a point X_0^d —usually assumed to be $X_0^d = (0_1, \ldots, 0_d)$— and by definition $X \in \mathbb{Z}^d$. So, a random walk process S_N will be defined as:

$$S_N = X_0 + \cdots + X_N = \left(\sum_{i=0}^{0} X_0^i, \ldots, \sum_{i=0}^{N} X_N^i \right). \tag{4.2}$$

The random walks have a specific distribution, in this case it will be assumed to be uniform, so if $\xi \sim \textbf{unif}\{-1, 1\}$ then:

$$\forall i \geq 1, \forall N \geq 0, \forall d \in \mathbb{N} : X_N^{i+1} = X_N^i + \xi^i \sim \textbf{unif}\{-1, 1\}. \tag{4.3}$$

Note then that equation 4.3 implies that if $X_0^0 = 0$ then:

$$X_0^0 = 0 \implies \mathbb{P}(Z_0^1 = 1) = \mathbb{P}(Z_0^1 = -1) = \frac{1}{2}. \tag{4.4}$$

Assumed now a random walk of one dimension, so $d = 1$, N steps where $N \geq 0$, a random position $m \in \mathbb{Z}$ and where $m \leq N$, then what would be the probability that after N steps $X_N = m$? Note the probability of each steps is uniform and independent. Furthermore, in each step n either $X_n > X_{n-1}$ or $X_n < X_{n-1}$ because, given the distribution defined in 4.3 X_n will either increase 1 or reduce 1 with respect to (w.r.t) X_{n-1}. So, given N steps we know by definition there will be n_1 jumps in which $X_n < X_{n-1}$ and n_2 steps in which $X_n > X_{n-1}$ for any $n = 1, \ldots, N$. Therefore,

$$N = n_1 + n_2 \iff n_1 = \frac{1}{2}(N + m) \iff n_2 = \frac{1}{2}(N - m). \tag{4.5}$$

Note that by 4.4 we can say that given a step n and $q := \mathbb{P}(Z^{n+1} = -1)$ then the opposite probability $p := \mathbb{P}(Z^{n+1} = 1)$ add up 1 so:

$$\mathbb{P}(Z^{n+1} = 1), \mathbb{P}(Z^{n+1} = -1) \sim \textbf{unif}\{-1, 1\} \implies q + p = 1. \tag{4.6}$$

Therefore if after every specific step the change to increase 1 or reduce -1 w.r.t the last step is p or q respectively, then the accumulated probability of the path given by the increases and decreases after n steps is given by P^*:

$$P^* = \prod_{i=1}^{n_1} p \prod_{i=1}^{n_1} q = q^{\frac{1}{2}(N+m)} p^{\frac{1}{2}(N-m)}. \tag{4.7}$$

η steps where $\eta \leq n$ the paths to the left and right change and therefore P^* must be multiplied by the total amount of paths with $n1$ steps to the left and $n2$ steps to the right. This is related with the ways to combine n_1 things given a total of N and similarly for n_2. As you might expect, this is clearly related with the N choose K problem. However, the amount of combinations the object can be ordered is reduced the more objects are chosen, so after one out of N is chosen there will be $N-1$ and so on [5]. Consequently, the total amount of combinations given n_1 objects is $N(N-1)\cdots(N-n_1-1)$ and therefore the whole probability of being at point m after N steps —answering therefore the initial question— is given by:

$$\begin{aligned}\mathbb{P}(m|N) &= \frac{N!}{\left(\frac{N+m}{2}\right)!\left(\frac{N-m}{2}\right)!} q^{\frac{1}{2}(N+m)} p^{\frac{1}{2}(N-m)} \\ &= \binom{N}{n} q^{\frac{1}{2}(N+m)} p^{\frac{1}{2}(N-m)}. \end{aligned} \tag{4.8}$$

,

Since equation 4.8 represents the probability of every specific point of the 1-dimensional lattice it therefore forms the whole distribution.

Given the distribution $\mathbb{P}(m|N)$ we can therefore compute all its moments, so that given:

$$(pu+q)^N = \sum_{n=0}^{N} \binom{N}{n} q^{\frac{1}{2}(N+m)} p^{\frac{1}{2}(N-m)}. \tag{4.9}$$

,

We deduce $\mathbb{P}(m|N)$ is the coefficient in the binomial expansion of 4.9, so:

$$\sum_{n=0}^{N} \mathbb{P}(m|N) = [(pu+q)^N]_{u=1} = 1. \tag{4.10}$$

Since the distribution is normalized to 1 we can then compute, for instance, the expectation of n, also known in terms of moments as the first moment of n, as:

$$\begin{aligned}\mathbb{E}[n] &= \sum_{n=0}^{N} n\mathbb{P}(m|N) = \sum_{n=0}^{N} n \left[\binom{N}{n} u^n p^n q^{N-n}\right]_{u=1} \\ &= \sum_{n=0}^{N} n \left[\binom{N}{n} u \frac{d}{du}(u^n p^n q^{N-n})\right]_{u=1} = u\frac{d}{du}\sum_{n=0}^{N} n \left[\binom{N}{n} u^n p^n q^{N-n}\right]_{u=1} \\ &= \left[u\frac{d}{du}(pu+q)^N\right]_{u=1} = Np. \end{aligned} \tag{4.11}$$

In the same way we can derive the square of the deviations from the point of origin or in terms of moments the second moment such as:

$$\mathbb{E}[n^2] = \left[\left(u\frac{d}{du}\right)^2 (pu+q)^N\right]_{u=1} = Np + N(N-1)p^2. \tag{4.12}$$

The variance of the variable will be defined based on the equation of 4.12 and therefore as a function of the standard deviation of the distribution which in this context is nothing else than the some sort of measure of the distribution:

$$\mathrm{Var}[n] = \sigma^2 = Npq \tag{4.13}$$

Based on the standard deviation concept in this context, we can use 4.11 and 4.13 to get some kind of relative amplitude in the distribution such that:

$$\frac{\sigma}{\mathbb{E}[n]} = \sqrt{\frac{q}{p}}\frac{1}{\sqrt{N}}. \tag{4.14}$$

Note some interesting features about 4.14, first whenever N increases, $\sqrt{N}$ increases and therefore this relative amplitude of the distribution is reduced and note similar scenarios occur if the ratio difference in $\sqrt{\frac{q}{p}}$ decreases and the other way around if it increases and the distribution is therefore *self-averaging*.

So, now, not in terms of steps but in terms of the position after N steps it will be then given by:

$$\mathbb{E}[m] = N(p - q). \tag{4.15}$$

In the same way the second moment and therefore square of the deviations from the point of origin will be given by:

$$\mathbb{E}[m^2] = 4\mathbb{E}[n^2] - 4\mathbb{E}[n^2]N + N^2 = 4\sigma^2 + \mathbb{E}[m]^2. \tag{4.16}$$

The variance is then the standard deviation squared so:

$$\sigma_m^2 = 4\sigma^2 = 4Npq. \tag{4.17}$$

Note that if by symmetry $p = 1 = \frac{1}{2}$ then $\sigma_m^2 = N$ which is known simply as the *free difusion*.

As we see the random walks are nothing else than a non-decreasing stochastic process that fluctuates in each step and this is essential to introduce the stochastic differential equations.

4.1. Distributions

The binomial distribution for $Np \to \infty$ in the random walk can be used to approximate the factorials in the binomial distribution as:

$$ln(N!) = \left(N + \frac{1}{2}\right) ln(N) - N + \frac{1}{2} ln(2\pi) + O(\frac{1}{N}). \tag{4.18}$$

From 4.19 we can derive $ln(\mathbb{P}(m|N))$ such as:

$$ln(\mathbb{P}(m|N)) = \left(N + \frac{1}{2}\right) ln(N) - \left(\frac{N+m}{2} + \frac{1}{2}\right) ln\left(\frac{N+m}{2}\right) - \left(\frac{N-m}{2} + \frac{1}{2}\right) ln\left(\frac{N-m}{2}\right) + \frac{N+m}{2} ln(p) + \frac{N-m}{2} ln(q) - \frac{1}{2} ln(2\pi). \tag{4.19}$$

When $N \to \infty$ we see, or we expect, the average of the distribution to be at the peak of the distribution because it is gaussian and therefore *bell-shaped* so we can approximate it as:

$$m = N(p-q) + \delta m. \tag{4.20}$$

From which can be derived the following equality:

$$\frac{N+m}{2} = Np + \frac{\delta m}{2} \text{ and } \frac{N-m}{2} = Nq + \frac{\delta m}{2}. \tag{4.21}$$

And using 4.22 and 4.19 we get:

$$\begin{aligned} ln(\mathbb{P}(m|N)) &= \left(N + \frac{1}{2}\right) ln(N) - \frac{1}{2} ln(2\pi) - \left(Np + \frac{1+\delta m}{2}\right) ln\left[Np\left(1 + \frac{\delta m}{2Np}\right)\right] \\ &- \left(Nq + \frac{1-\delta m}{2}\right) ln\left[Np\left(1 + \frac{\delta m}{2Np}\right)\right] + \left(Np + \frac{\delta m}{2}\right) ln(p) + \left(Nq + \frac{\delta m}{2}\right) ln(q) \\ &= -\frac{1}{2} ln(2\pi Npq) - \left(Np + \frac{1+\delta m}{2}\right) ln\left(1 + \frac{\delta m}{2Np}\right) - \left(Nq + \frac{1-\delta m}{2}\right) ln\left(1 - \frac{\delta m}{2Np}\right). \end{aligned} \tag{4.22}$$

Then if we expand the logarithm:

$$ln(1 \pm x) = \pm x - \frac{1}{2}x^2 + O(x^3). \tag{4.23}$$

And using 4.23 we get:

$$\begin{aligned} &ln(\mathbb{P}(m|N)) - \frac{1}{2} ln(2\pi) Npq + \frac{(p-q)}{4Npq}\delta m - \frac{1}{2}\frac{1}{4Npq}(\delta m)^2 \\ &= -\frac{1}{2} ln(2\pi) Npq + \frac{(p-q)}{\sigma_m^2}\delta m - \frac{1}{2}\frac{1}{2\sigma_m^2}(\delta m)^2. \end{aligned} \tag{4.24}$$

Note that in 4.24 was used the equality $\sigma_m^2 = 4Npq$. Also in the second equality it is of $O((Np)^{-1/2})$ and therefore can be removed when $Np \to \infty$, such that:

$$\mathbb{P}(m|N) \to \frac{2}{\sqrt{2\pi\sigma_m^2}} exp\left(-\frac{1}{2}\frac{(\delta m)^2}{\sigma_m^2}\right). \tag{4.25}$$

Finally when we are interested in to know the probability to find the random walk in an specific interval $2\Delta x$ around a specific position x in a particular time t,if we define $\Delta x \to 0$, $\Delta t \to 0$, $D = 2pq\frac{(\Delta x)^2}{\Delta t}$, $\upsilon = (p-q)\frac{\Delta x}{\Delta t}$, then we can find it as:

$$\mathbb{P}(x|t) = \frac{1}{\sqrt{4\pi Dt}} exp\left(-\frac{(x-\upsilon t)^2}{4Dt}\right). \tag{4.26}$$

Finally just note in 4.26 we assume the prerequirement $p(x,t) = \delta(x)$ [9].

4.2. Random walks and stochastic differential equations

The random walks introduced in the last part are really related with the stochastic differential equations. Both are really important concepts in many areas, as probability theory, stochastic theory, mathematical finance among other topics. Random walks can be used, for instance, to model the behavior of a series of processes with uncertainty, as the stock prices, and can be used to make predictions about future movements or states of the system as we did in the last section.

It is here where stochastic differential equations become relevant since they can be used to improve the study of predictions. Stochastic differential equations are used to model systems that evolve over time and are influenced by random forces. As the markov chains derivation in stochastic differential equations are really important tools for understanding and study of predictions. [24]

4.3. Stochastic differential equations

A stochastic differential equation is a type of differential equation that contains one or more random variables or stochastic processes and it is just and extension of the ordinary differential equation. An ordinary differential equation is defined as:

$$\frac{dx(t)}{dt} = f(t,x), dx(t) = f(t,x)dt. \tag{4.27}$$

with a given initial condition $x(0) = x_0$, so it can be written as an integral such that:

$$x(0) = x_0 + \int_0^t f(s, x(s))ds, \tag{4.28}$$

note that if 4.28 had solution, it would simply be $x(t) = x(t, x_0, t_0)$ assuming $x(0) = x_0$.

Now let's take any random differential equation, *random in the sense of any*, as for instance one similar to 4.28:

$$\frac{dx(t)}{dt} = a(t)x(t), x(0) = x_0. \tag{4.29}$$

Note that in 4.29 we do not see anything beyond a normal ordinary differential equation. Now, if we assume $a(t)$ is not a deterministic parameter but rather a stochastic variable, we can get in exchange a stochastic differential equation. We can then define $a(t)$ as:

$$a(t) = f(t) + h(t)\xi(t). \tag{4.30}$$

Note $\xi(t)$ is simply a not deterministic value —assume it is a white noise process—. We then get:

$$\frac{dX(t)}{dt} = f(t)X(t) + h(t)X(t)\xi(t). \tag{4.31}$$

then if we write it in the differential form, and we use $dW(t) = \xi(t)dt$ where $W(t)$ is a brownian motion we get:

$$dX(t) = f(t)X(t)dt + h(t)X(t)dW(t). \tag{4.32}$$

From 4.32 we derive then the usual form of a stochastic differential equation:

$$dX(t) = f(t, X(t, \omega))dt + g(t, X(t, \omega))dW(t, \omega). \tag{4.33}$$

In this case ω just implies that $X(t, \omega) = 0$ by definition. So, from 4.31 we derive:

$$dY(t, \omega) = \mu(t)dt + \sigma(t)dW(t, \omega). \tag{4.34}$$

Finally if $f(t, X(t, \omega)), g(t, X(t, \omega)), W(t, \omega) \in \mathbb{R}$ we can express 4.33 in integral form [11] as:

$$X(t, \omega) = X_0 + \int_0^t f(s, X(s, \omega))ds + \int_0^t g(s, X(s, w))dW(s, w). \tag{4.35}$$

5. Text generation algorithms

5.1. Introduction and Motivation

Let's take a brief detour to talk about a recent event in the artificial intelligence world: On November 30th 2022, the OpenAI research laboratory released a new chatbot prototype called ChatGPT. This is an artificial intelligence language processing model that simulates a conversation between the user and another person.

This model quickly gained attention due to its realistic and natural answers, making most replies seem written by a human instead of an algorithm. Its language processing model allows it to seemingly understand your prompt and generate an appropriate response. These two impressive feats have made ChatGPT a very well known chatbot in both social media and professional settings.

Although at first glance upon using the prototype it may look like the amount of progress in the artificial intelligence field has reached science fiction territory, there are two key components that one must remember:

Firstly, even if the response might look like the results of an online browser search, what it is doing is something completely different. The program is not copying the best internet posts to the question, but rather it is generating a completely new text from scratch, word for word, until it answers the question. Note that this word for word text generation implies some level of randomness in the response, as can be seen by asking the same thing several times and receiving different responses.

Secondly, even if it might look like there is some semblance of logic in the replies due to the usual coherence of the sentences, there is no such thing to make sure the overall text makes sense. Asking for non-direct problems that need a few steps to solve makes this clear, for example asking "I am 7 years older than my sister, she has 2 less years than half my age. How old is my sister?" will usually result in a completely wrong and nonsensical answer. Needless to say inputting anything more complicated than that gets even worse.

Even with its flaws, ChatGPT is an incredible prototype that has reached and astonished countless people, with many wondering how is it possible to build such a program. In this section we will explain the basic ideas behind the first versions of such models, and some of the improvements that have been made to make them more realistic.

Overall, a language model can be divided in two main parts: The language processing, which tries to understand the input to give an adequate response, and the text generation, which creates the text that is given as a response. Some of the basis behind these two parts can be found [3], as well as some other applications such as text encoding or speech recognition which are largely based on the same models. In this section we will focus on the text generation algorithms.

5.2. Basic text generation model

In order to turn a text into a Markov chain there are two of its properties that we have to remember to keep in mind:

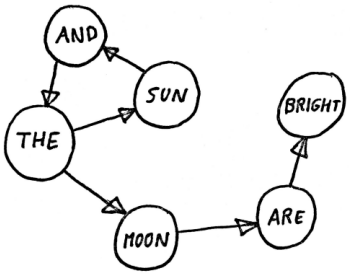
The first one is that Markov chains are a memory-less process. This means that the next state of the chain only depends on what the current state is, independently of what the process has gone through before.

The second one is that, throughout this section, we will only work with discrete time Markov chains.

The question we pose now is: How do we build a Markov chain so that it can represent a given text?

We can start by considering each different word a possible state of the chain. One word X will lead to another word Y if, in the training text provided, there is some point where word X is followed by word Y. The probability of the chain switching to this state depends on the amount of instances where word Y follows word X in proportion to how many other words follow word X.

As an example, let's use the training text: *The sun and the moon are bright*. In this case the Markov chain associated to this text has 6 possible states, one for each different word. The connections between the states look as follows:



We now need to establish what the probability of going to each state is. To do this, we first build a vector for each different state listing how many times it precedes each of the other words in the training text. For example, the vector associated to the state *the* would look like this:

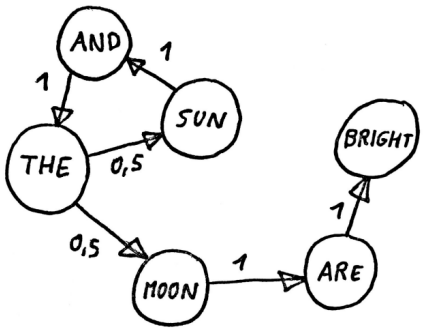
	(the)	(sun)	(and)	(moon)	(are)	(bright)
the	0	1	0	1	0	0

We can build this vector for each of the possible states, and have each of them be the rows of what we will call the instances matrix:

	the	sun	and	moon	are	bright
the	0	1	0	1	0	0
sun	0	0	1	0	0	0
and	1	0	0	0	0	0
moon	0	0	0	0	1	0
are	0	0	0	0	0	1
bright	0	0	0	0	0	0

Finally, if we normalize the instances matrix' rows we obtain the probability matrix that is used to define the Markov chain's chances to go from one state to the next:

$$\begin{pmatrix} \nearrow & the & moon & and & sun & are & bright \\ the & 0 & 0.5 & 0 & 0.5 & 0 & 0 \\ moon & 0 & 0 & 1 & 0 & 0 & 0 \\ and & 1 & 0 & 0 & 0 & 0 & 0 \\ sun & 0 & 0 & 0 & 0 & 1 & 0 \\ are & 0 & 0 & 0 & 0 & 0 & 1 \\ bright & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$



Now that we have a complete Markov chain we can start generating text with it. The process to do so is simple: Start with a random word from the training text, and use that word as the starting state in the chain to generate a new word. Now we have a generated text with two words; we define the current state of the Markov chain as the last word in the generated text so far. We can use this newly defined state to generate a new word, which we add to the generated text and increase its length by one. This process can be repeated until we obtain a text of the desired length.

Using this model and a long enough training text to fit the length of the text we want to generate, we can create a completely new text that replicates the style and contents of the training text.

At least that's in theory. The reality is that with such a simple model the results are in no way realistic, just some random words put together. An example of what we can get using this model with a relatively long training text is the following:

the subject he was all the request seemed to a long anti-religious poem in a later this poem in front of the chequered figure in May which affected Berlioz alone alone was saying was so powerful

As we can see, although we have been able to use Markov chains to generate a completely new text, the results leave much to be desired. In the next section we will explore a few ways to improve this model and obtain a more natural text.

5.3. Model improvements

5.3.1. Input token length. The first idea one could have to make more coherent sentences is to try and keep some of the context behind each word. For example, in a random text the word *upon* can be followed by multiple options, all of them viable in a normal setting. However, if the previous word is *once* it's probably referring to the expression *once upon a time* and the probability of the next generated word being *a* should rise accordingly.

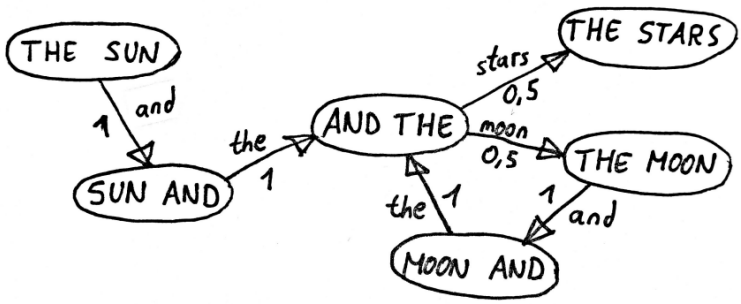
However, it is important to remember that Markov chains are a memory-less process, so we can't check what the previous chain state was. The solution to this is to redefine the current state of the Markov chain not as the last word, but as the last X words of the generated text. By considering sets of more than one word we are able to keep some of the context like we wanted while keeping the process memory-less. We will call these new states of the chain the input tokens.

As an example, let's use the training text: *The sun and the moon and the stars*, and let's use input tokens of length 2. We can build the same vectors as before checking pairs of words instead of single words, and checking the word after said pair in the training text. For example, the vector associated to the input token *and the* would be:

	(the)	(sun)	(and)	(moon)	(stars)
and the	0	0	0	1	1

Just like in the previous section we can build a matrix with the rows being the vector corresponding to each pair of words, and then normalize each row to obtain the probability matrix defining the new model:

$$\begin{pmatrix} \nearrow & the & sun & and & moon & stars \\ the\ sun & 0 & 0 & 1 & 0 & 0 \\ sun\ and & 1 & 0 & 0 & 0 & 0 \\ and\ the & 0 & 0 & 0 & 0.5 & 0.5 \\ the\ moon & 0 & 0 & 1 & 0 & 0 \\ moon\ and & 1 & 0 & 0 & 0 & 0 \\ the\ stars & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$



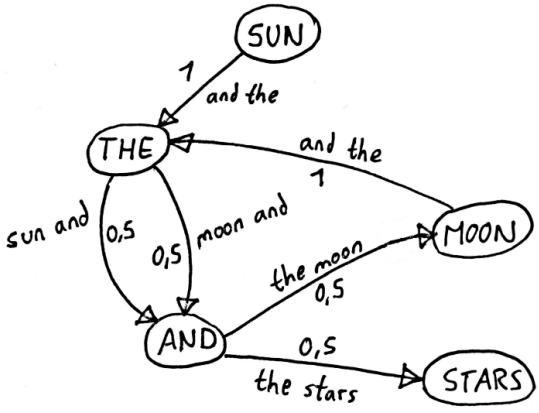
The results of implementing this idea turn out to be much less random and the words that are put together make more sense. An example of what we can get using this model with a relatively long training text and an input token length of 2 is the following:

The man was seven feet tall but narrow in the shoulders, incredibly thin. Ivan Nikolayich had written this poem in record time, but unfortunately the editor had commissioned the poet to write a long anti-religious poem for one of the strangest appearance.

Note that this implementation can also bring a grave problem if not calibrated correctly: If the input token length is too long, the word sets of that length will be so unique that it will in most cases appear only once in the training text. This means that there will usually be one option for the Markov chain to take when generating new text, which will make most of the generated text a copy of the training text. This is not what we aim to achieve.

5.3.2. Output token length. In a similar way to the last section we can consider what happens when, instead of changing the amount of words in the Markov chain definition, we change the amount of words generated each time. Let's build the matrix associated to the same example training text as in the last section, *The sun and the moon and the stars*, and see what the text generation process looks like:

$$\begin{pmatrix} \nearrow & \textit{the sun} & \textit{sun and} & \textit{and the} & \textit{the moon} & \textit{moon and} & \textit{the stars} \\ \textit{the} & 0 & 0.5 & 0 & 0 & 0.5 & 0 \\ \textit{sun} & 0 & 0 & 1 & 0 & 0 & 0 \\ \textit{and} & 0 & 0 & 0 & 0.5 & 0 & 0.5 \\ \textit{moon} & 0 & 0 & 1 & 0 & 0 & 0 \\ \textit{stars} & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$



This change in the amount of generated words has an interesting effect on the resulting text. Note that the process in this last model has more overall connections between the states than the prior ones, although due to the small size of the example it may be hard to see. While it still generates some chunks of text with sense due to the amount of words generated each step, it is more probable that it will jump to more unusual states than in the previous section's model.

Considering how we want to build a realistic and natural looking text this characteristic may look almost useless. However, although changing the input token size helps improve the results it may also make large chunks of the generated text be identical or very similar to the training text, which we want to avoid. This is why changing the length of both the input and output tokens could be a useful approach: The first one used to control the amount of context kept behind each word, the latter one to randomise the generated text and make it less similar to the original one.

5.3.3. Implementation details. Some readers may have noticed that we are glossing over some details when generating the new text, such as punctuation, capitalization, etc.

In the case of capitalization, we can simply lowercase all words before calculating the instances matrix to build the Markov chain. This will also avoid any issues where the code considers two of the same word as different states because of capitalization. Once the text has been generated we can add some simple post-processing code that capitalizes the character after a point or a semicolon accordingly.

The case of punctuation is a more interesting one. The most common way to approach this issue is to consider punctuation symbols as their own separate words and add them as possible states to the chain. This may result in some sentences having out of place commas

and points, but overall it has the same realistic result as the text itself and should not be the biggest cause of problems.

An alternative to this solution to the punctuation problem would be to add some sort of grammatical rules to the algorithm that detect when to add a comma or end a sentence. One could also implement such an algorithm together with the Markov chain model to obtain a more realistic and natural result. Overall it seems like an interesting branch of research to improve this basic text generation model.

Some further detail on the input and output token length modifications can be found in [20].

Another issue that arises is if the last token in the text is unique and does not appear anywhere else. In this case if the chain reaches that state it will not be able to generate a new word, since the process has no options to go to from that state.

There are several ways to go about this problem. The simpler one is to stop generating the text and send the user a message explaining what has happened. Another viable option is, were this case to show up while running the algorithm, to add a random token from the text. Considering the usual length of the training texts used for such programs, the probability of encountering this extreme case is so low that it will not affect the grand scheme of the generated text.

5.4. Test program

In order to provide the reader with an interactive way to test out and experiment with the Markov chain text generation algorithm described through these sections a Matlab script has been created and is publicly available through GitHub. The repository contains two files:

The first one is a text file called `text.txt`. Here is where one can put the training text that will be used to generate the Markov chain used in the text generator.

The second one is the Matlab script that builds the text generation model. In the initial part of the scripts one can find parameters for the length of the output text, as well as the length of the input and output tokens, so that the user can tweak and experiment with them as they wish.

Please note that, while the program already has all the model and token length options mentioned in the previous sections, some quality of life and general improvements will be made in the future.

The test program can be found at the following link: `shorturl.at/cDR45`

5.5. Text generation overview

In this application of the Markov chains we have seen the origins of the current text processing and response artificial intelligence programs, which have been rising in popularity

in the last few months. We have seen how to convert a training text into a Markov chain an how to generate text using this basic model. We have then made some improvements to the basic model by changing the amount of words read at once and the amount of words generated at once, and seen that changing these parameters can substantially increase the realism of the result.

Although the generated text with the presented model still does not look like something a person, it is important to remember where such advanced programs come from, and how starting with such simple ideas and models one can slowly but surely improve them until they reach the masterpieces that are nowadays artificial intelligence language processing models.

6. Conclusion

As we reviewed, Markov chains are really powerful not only to model or study systems and probabilities but also to innovate in text generation or to model random walks. This is the most interesting fact about Markov chains, it is absolutely powerful because in theoretical terms its base is always really similar, we always can start by studying the transitioning from specific states to others — since by definition others are not affected by current states —. This property makes Markov chains a useful tool for predicting future states and understanding the long-term behavior of a large amount of pure and applied situations and problems. With the use of Markov chains, many real-world problems can be effectively modeled and analyzed, leading to more accurate predictions and better decision making.

We also saw how the combination with additional tools to as the stochastic differential equations can help to improve even more the randomness and uncertainty.

As we see, the possibilities are yet to explore, however, not because of that limited but the other way around probably unlimited just as the transitions in a Markov chain matrix.

7. Acknowledgements

We are specially thankful for the resources provided by the universities and the professors of both the Valencia Polytechnic University and the University of Valencia from which we intensively used.

References

[1] Ehrhard Behrends. Introduction to markov chains with special emphasis on rapid mixing. vieweg, 2000.

[2] Blaettler Florian, Kollmorgen Sepp, Herbst Joshua, and Hahnloser Richard. Hidden markov models in the neurosciences. *Hidden Markov Models, Theory and Applications, ed P. Dymarski (Rijeka: IntTech)*, pages 169–186, 2011.

[3] Frederick Jelinek. Markov source modeling of text generation. In *The impact of processing techniques on communications*, pages 569–591. Springer, 1985.

[4] Chris Karlof and David Wagner. Hidden markov model cryptanalysis. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 17–34. Springer, 2003.

[5] Gregory F Lawler and Vlada Limic. *Random walk: a modern introduction*, volume 123. Cambridge University Press, 2010.

[6] Rogemar S Mamon and Robert James Elliott. *Hidden Markov models in finance*, volume 4. Springer, 2007.

[7] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of machine learning*. MIT press, 2018.

[8] Sudha Morwal, Nusrat Jahan, and Deepti Chopra. Named entity recognition using hidden markov model (hmm). *International Journal on Natural Language Computing (IJNLC) Vol*, 1, 2012.

[9] John N Mundy. Random walk. In *Defect and Diffusion Forum*, volume 353, pages 1–7. Trans Tech Publ, 2014.

[10] Valery A Petrushin. Hidden markov models: Fundamentals and applications. In *Online Symposium for Electronics Engineer*, 2000.

[11] Philip E Protter. Stochastic differential equations. In *Stochastic integration and differential equations*, pages 249–361. Springer, 2005.

[12] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[13] Lawrence R Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[14] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

[15] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[16] Mario Stanke and Stephan Waack. Gene prediction with a hidden markov model and a new intron submodel. *Bioinformatics*, 19(suppl_2):ii215–ii225, 2003.

[17] Richard S Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44, 1988.

[18] Richard S Sutton. Gain adaptation beats least squares. In *Proceedings of the 7th Yale workshop on adaptive and learning systems*, volume 161, page 166, 1992.

[19] Richard S Sutton, Andrew G Barto, and Ronald J Williams. Reinforcement learning is direct adaptive optimal control. *IEEE control systems magazine*, 12(2):19–22, 1992.

[20] G Szymanski and Z Ciota. On-line text generation using markov models. In *Proceedings of the International Conference Modern Problems of Radio Engineering, Telecommunications and Computer Science, 2004.*, pages 339–341. IEEE, 2004.

[21] Gerald Tesauro. Temporal difference learning of backgammon strategy. In *Machine Learning Proceedings 1992*, pages 451–457. Elsevier, 1992.

[22] Gerald Tesauro. Td-gammon, a self-teaching backgammon program, achieves master-level play. *Neural computation*, 6(2):215–219, 1994.

[23] Keiichi Tokuda, Yoshihiko Nankaku, Tomoki Toda, Heiga Zen, Junichi Yamagishi, and Keiichiro Oura. Speech synthesis based on hidden markov models. *Proceedings of the IEEE*, 101(5):1234–1252, 2013.

[24] Nicolaas G Van Kampen. Stochastic differential equations. *Physics reports*, 24(3):171–228, 1976.