ANÁLISIS DE SISTEMAS COMPLEJOS USANDO MACHINE LEARNING

Alejandro Luque Cerpa

ANÁLISIS DE SISTEMAS COMPLEJOS USANDO MACHINE LEARNING

Alejandro Luque Cerpa

Memoria presentada como parte de los requisitos para la obtención del título de Grado en Matemáticas por la Universidad de Sevilla.

Tutorizada por

Prof. Dr. Fernando Sancho Caparrini

Agradecimientos

Muchas gracias a mi tutor, el Prof. Dr. Fernando Sancho Caparrini, primero por haber influido en mi elección de este Trabajo de Fin de Grado al mostrarse como un excelente profesor en la asignatura de *Inteligencia Artificial*, y segundo, aunque no menos importante, por haberme guiado en su elaboración.

Muchas gracias también a mis padres, porque sin ellos no estaría aquí.

Agradecimientos también a mi antiguo profesor de matemáticas Curro Piñero, porque sin él las cosas hubieran sido muy distintas.

A Rafael Leal, por acompañarme en esta aventura desde el principio.

Índice general

English Abstract

Nowadays, we can find complex systems in a great variety of fields, such as biology, physics, engineering or social systems. From fluid dynamics to biomolecular interactions, we can find complex systems everywhere, and it is not an easy task to predict their behaviour. Cellular automata are considered great techniques of modelling complex systems. In cellular automata, some simple rules lead sometimes to emergent chaotic or complex behaviours that are interesting in the context of some real-life complex models. Machine Learning algorithms are well known for identifying underlaying patterns in complex structures. In this paper, cellular automata and some Machine Learning algorithms, such as Neural Networks or k-Nearest Neighbours are introduced, and after that some of them are used to analyze cellular automata. The aim of this paper is to define a way of working with cellular automata and machine learning that can be expanded later with new algorithms, new automata or more data.

1 Introducción

Actualmente, podemos encontrar sistemas complejos en una gran variedad de áreas, tales como Biología, Física, Ingeniería o Ciencias Sociales. Desde la dinámica de fluidos, hasta las interacciones biomoleculares, podemos encontrar sistemas complejos en todas partes, y no es una tarea fácil predecir su comportamiento futuro a partir de su evolución pasada y su situación actual.

Los Autómatas Celulares están considerados, simultáneamente, grandes ejemplos de sistemas complejos a la vez que buenos modelos para otros sistemas complejos que aparecen en el mundo real. En algunos autómatas celulares las reglas simples que los definen a veces dan lugar a comportamientos emergentes complejos o caóticos que son interesantes en el contexto de modelos complejos reales.

Los algoritmos de Aprendizaje Automático son bien conocidos por su capacidad para identificar patrones subyacentes en estructuras de datos complejas. En este trabajo se introducen conceptos fundamentales de Autómatas Celulares y de Aprendizaje Automático (que nombraremos por su nombre en inglés, Machine Learning, por ser mucho más común) como vía para abordar un estudio que no suele ser tan común, y es la aplicación de algoritmos de aprendizaje para el estudio experimental de patrones dinámicos de autómatas.

El objetivo principal es doble, por una parte, analizar hasta qué punto los estudios y clasificaciones actuales de la complejidad en Autómatas puede ser abordada y comprendida desde un punto de vista computacional por medio de Aprendizaje Automático y, por otra parte, buscar los límites que algunos algoritmos de Aprendizaje tienen para reconocer patrones cuando éstos superan una cierta complejidad.

Suele ser común situar los límites del aprendizaje automático en la presencia de ruido o procesos puramente aleatorios, sin embargo, es interesante observar hasta qué punto un proceso complejo no aleatorio (como las dinámicas deterministas y simples

que veremos en algunos autómatas) pueden estar fuera de la capacidad predictiva de los algoritmos más habituales (clásicos) de aprendizaje automático.

De forma más general, podemos identificar los siguientes objetivos en el trabajo que aquí se presenta:

- Entender y formalizar matemáticamente el concepto de Autómata Celular en su sentido más amplio.
- Describir las clasificaciones más habituales de la dinámica de los Autómatas Celulares unidimensionales, que se basan principalmente en la descripción fenomenológica del comportamiento observado en este tipo de autómatas dada por Wolfram.
- Entender el contexto formal matemático del Aprendizaje Automático (ML) como herramienta de la aproximación de funciones (en su sentido más amplio) desde un punto de vista computacional.
- Conocer las características y limitaciones más importantes de los algoritmos de aprendizaje clásicos.
- Generar una batería de experimentos siguiendo una metodología razonada para evaluar las hipótesis de partida: capacidad predictiva de los algoritmos clásicos de ML sobre dinámicas complejas.
- Ser capaz de manejar un lenguaje de programación científico (Python) para la realización efectiva de los experimentos planteados, haciendo uso de librerías especializadas existentes para tal fin (Scikit-learn).
- Extraer conclusiones solventes de los resultados obtenidos, y planificar líneas de trabajo prometedoras en función de éstos.

Debe indicarse que entre los objetivos de este trabajo no se busca obtener unos resultados específicos, sino que el objetivo principal es el propio desarrollo de la metodología. Debido a las características propias de un Trabajo Fin de Grado, no se dan las condiciones adecuadas para introducirse en el manejo de técnicas avanzadas de ML (como sería el uso de algoritmos de Deep Learning), por lo que se espera que los resultados obtenidos con las técnicas clásicas no sean satisfactorias. En lugar de un resultado de aproximación sorprendente, es más importante la justificación que se pueda extraer de que, para las técnicas clásicas, la complejidad determinista de los autómatas celulares suponen un límite que merece la pena explorar.

1.1 Estructura de la memoria

Con los objetivos anteriores en mente, se ha estructurado la memoria que aquí se presenta en los siguientes capítulos:

- **Introducción**: Este capítulo, donde se dan las motivaciones y objetivos principales del trabajo, y se expone la estructura de la memoria.
- **Autómatas Celulares**: Donde se expone la formalización matemática de este tipo de objeto matemático, y se repasan sus principales características y clasificaciones realizadas respecto a su dinámica (en el caso unidimensional).
- **Machine Learning**: Donde se muestra una aproximación matemática del aprendizaje automático y se hace un rápido repaso por algunos de los algoritmos/modelos de ML más habituales (los que serán usados en la parte experimental del proyecto).
- **Resultados Experimentales**: Donde se explican brevemente los experimentos realizados y se analizan los resultados obtenidos para algunos de los autómatas celulares más representativos de cada clase de complejidad dinámica.
- **Conclusiones**: Donde se desarrollan las conclusiones más relevantes que se han obtenido durante la ejecución del trabajo y se exponen algunas de las líneas de trabajo futuro que pueden resultar interesantes tras haber establecido las conclusiones sobre el trabajo aquí realizado.

2 Autómatas Celulares

2.1 Introducción

Los sistemas complejos tienen el problema de que no puede estudiarse individualmente cada una de sus componentes para averiguar el comportamiento global del sistema debido a la gran cantidad de interrelaciones que se dan entre estas. Esto obligó a que se buscaran nuevos enfoques para intentar modelizarlos, y en este contexto surgieron los autómatas celulares.

Los autómatas celulares son modelos matemáticos y computacionales sencillos que se siguen utilizando para modelar sistemas complejos en la actualidad en campos como la física, biología, química o ingeniería.

2.1.1 Contexto histórico

Se puede considerar que los Autómatas Celulares constituyen uno de los primeros modelos de Computación Natural, en donde se ha producido una inspiración en procesos naturales observados para generar un modelo de computación teórico completamente formalizado (y que, a veces, persigue objetivos distintos a los del modelo real en el que se inspira). Su origen se produce en los años 50, principalmente por parte de John Von Neumann (en la imagen 2.1) mientras intentaba desarrollar un modelo abstracto de auto-reproducción en biología, y sin el uso de ordenadores. Los trabajos desarrollados por él y por Stanislaw Ulam, que había considerado el problema de forma independiente, ayudaron más adelante a resultados tan sorprendentes como el descubrimiento del ADN.

Tras los primeros resultados formales obtenidos, y tras depurar las diversas va-

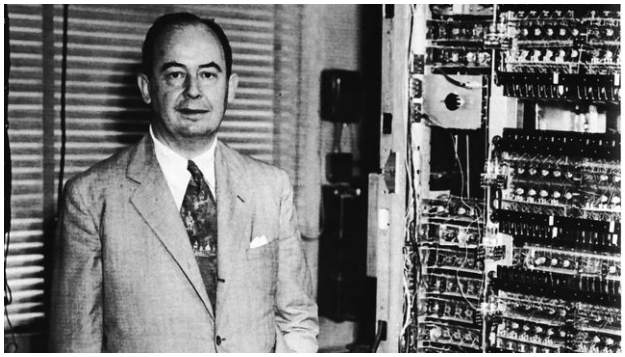


Figura 2.1: Von Neumann. Recuperada de www.eldiario.es

riantes que podían construirse de los primeros modelos formales definidos, a finales de los años 50 y principios de los 60, se obtuvieron una serie de resultados mucho más detallados y técnicos que probaban la capacidad computacional de los autómatas celulares que, bajo determinadas condiciones, era equivalente a los que ofrecían las máquinas de Turing. Debido al carácter dinámico de la evolución de los autómatas celulares, también se produjeron los primeros intentos de conectar éstos con Sistemas Dinámicos.

En 1968, a pesar de la poca investigación desarrollada sobre ellos, John Conway comenzó a trabajar, primero a mano y más tarde con ayuda de ordenadores, en una gran variedad de reglas para autómatas celulares de dos dimensiones. Más adelante, en 1970, expuso una serie de reglas a las que llamó "El juego de la vida" (una muestra en la figura 2.2), y que se popularizaron a través de la revista *Scientific American* por Martin Gardner. Estas reglas exhibían, a pesar de ser bastante simples, un amplio rango de comportamientos complejos.

Además, y de forma completamente independiente, algunos casos particulares de autómatas celulares 2D (2.3) se aplicaban desde los 50 a modo de filtros para eliminar el ruido en el procesamiento de imágenes digitales.

Mucho más adelante, en algunos artículos de Stephen Wolfram que también se recogen en [1], se define una clasificación de los autómatas celulares en cuatro clases según la complejidad que muestran en sus dinámicas. Esta es una de las clasificaciones más conocidas, y divide el conjunto de posibles autómatas celulares entre aquellos que se estabilizan en un estado, los que repiten una serie de estructuras periódicamente, los que evolucionan a patrones caóticos, y los que evolucionan a patrones complejos, algunos con larga duración. Los límites entre estas clases son difíciles de discernir a veces, especialmente entre los dos últimos.

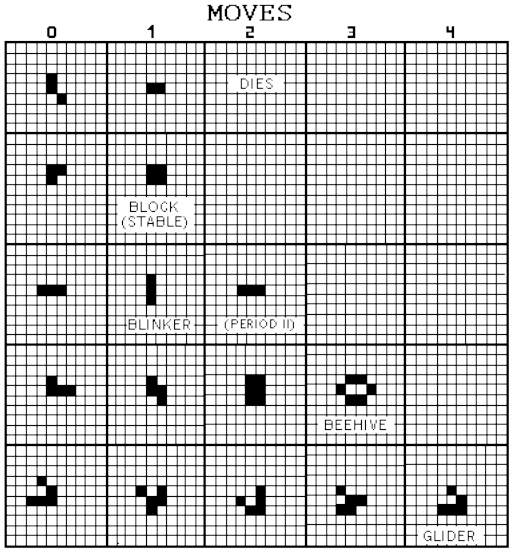


Figura 2.2: Juego de la vida. Recuperada de www.daviddarling.info

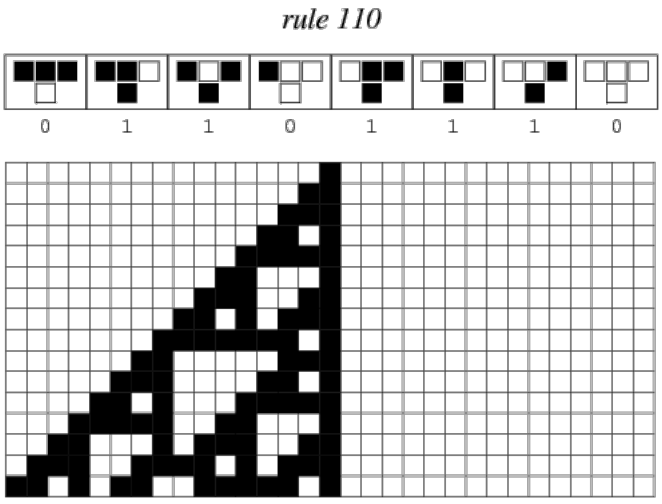


Figura 2.3: Ejemplo de la regla 110. Recuperada de mathworld.wolfram.com

Culik y Yu presentaron en 1988 [2] una clasificación formal que se corresponde con la de Wolfram y que será descrita más adelante en este mismo capítulo.

Otra clasificación surge de los trabajos de Li y Packard, que en [3] extienden el esquema de Wolfram a seis clases de complejidad, que también serán descritas más adelante.

2.1.2 Aplicaciones

Más allá del estudio teórico acerca de la complejidad que pueden alcanzar los autómatas celulares y su clasificación, existen bastantes aplicaciones a distintos problemas del mundo real. Algunos ejemplos son los siguientes:

- El crecimiento de algunos cristales, y en concreto los patrones en copos de nieve, pueden ser modelizados por autómatas simples en 2D (Figura 2.4).
- Los patrones encontrados en las conchas de los moluscos *Cymbiola innexa* son similares a los generados por la regla 30 en los autómatas celulares 1D (Figura 2.5).
- En criptografía, si se combinan distintas reglas para determinadas células dentro de un mismo autómata, y se usan reglas lo suficientemente caóticas, se pueden cifrar cadenas, siendo extremadamente costoso en tiempo intentar revertir el proceso sin saber qué reglas se han utilizado.
- Procesamiento de imágenes digitales en medicina.
- Modelización de la activación eléctrica del corazón en cada pulsación.
- Modelización de dinámica de fluidos, como por ejemplo los modelos de Lattice-gas y Lattice Boltzman.
- Modelización del flujo de tráfico en ciudades [4].

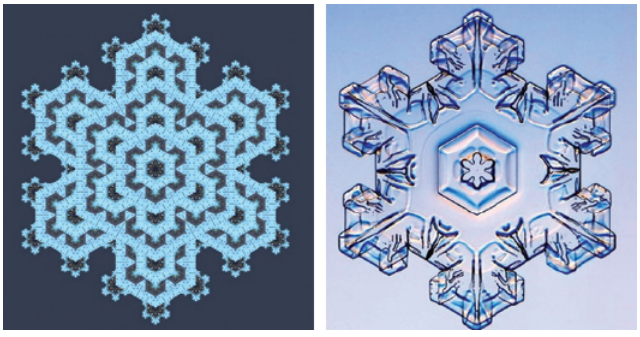


Figura 2.4: Autómata celular y copo de nieve. Recuperada de www.nap.edu

Figura 2.5: Cymbiola innexa. Recuperada de www.caledonianseashells.com

2.2 Definiciones y clasificación

Aunque hay una gran variedad de Autómatas Celulares según sus características, estudiaremos únicamente los más habituales y sencillos, que son autómatas celulares síncronos cuya topología es una cuadrícula rectangular infinita, de forma que cada célula se identifica con una casilla de la cuadrícula.

Definición 2.1. *Un autómata celular se define a partir de:*

- *Una matriz infinita de casillas de dimensión d. Cada casilla se denomina célula. Formalmente, podemos identificar esta matriz con $\mathbb{Z}^d$.*
- *Un conjunto finito de estados, S. Cada célula toma en cada momento un valor de este conjunto. Este conjunto se denomina alfabeto.*
- *Una configuración inicial, es decir, una asignación inicial de un estado a cada una de las células del espacio. Formalmente, la configuración es una función*

$$c : \mathbb{Z}^d \to S$$

  que especifica el estado de cada célula. Las funciones constantes se denominan configuraciones homogéneas. El conjunto de todas las configuraciones se denota por $C(d, S)$, o por C si d y S se conocen por el contexto.
- *La vecindad de cada célula, que es el conjunto de células que se consideran adyacentes a una dada. Es común especificar el conjunto de vecinos de una célula x por medio de un vector $N_x = (x_1, ..., x_n)$ de n elementos distintos de $\mathbb{Z}^d$, de forma que los vecinos son las células de las casillas $x + x_i$ con $i = 1, ..., n$. Normalmente, se considera el mismo vector N_x para todas las células del autómata.*
- *Una función de transición local*

$$f : S^n \to S$$

donde n es el tamaño del conjunto de células vecinas. Esta función toma como valores de entrada la vecindad de una célula, y devuelve el próximo estado de dicha célula. Con la función de transición local se puede construir una función de transición global, $G : C \to C$.

Las células cambian su estado de forma síncrona en intervalos de tiempo discretos. El siguiente estado depende del estado de las células vecinas (la condición de vecindad es distinta para cada autómata). Todas las células usan la misma regla para actualizar su estado. Se muestran algunos ejemplos en las Figuras 2.6, 2.7 y 2.8.

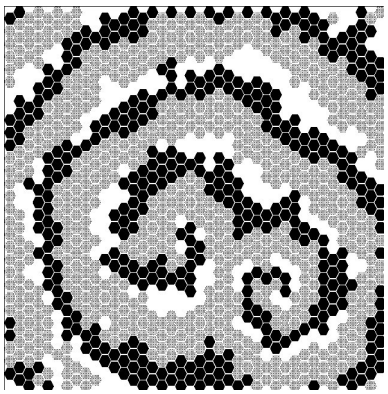


Figura 2.6: Autómata hexagonal. Recuperada de www.mced-ecology.org

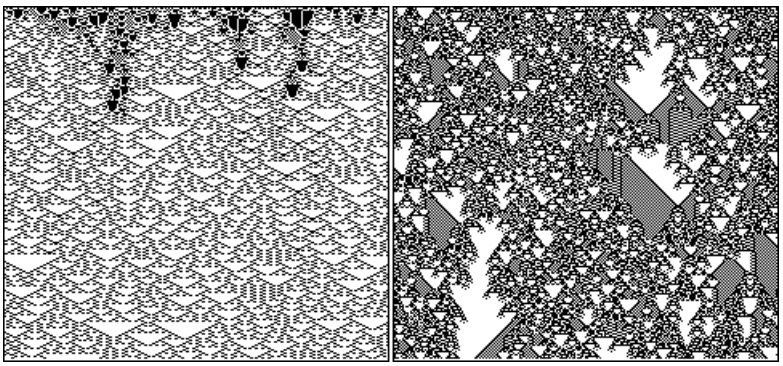


Figura 2.7: Ejemplos de autómatas 1-D. Recuperada de users.math.yale.edu

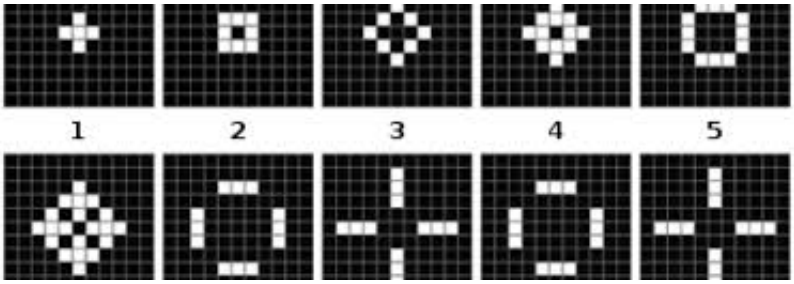


Figura 2.8: Ejemplo de autómata 2-D. Recuperada de culturacolectiva.com

A partir de este momento trabajaremos con autómatas celulares con dimensión $d = 1$, alfabeto $S = \{0, 1\}$, y considerando que la vecindad de una célula está compuesta por la célula que está a su izquierda y la que está a su derecha.

En este caso, será habitual considerar autómatas celulares en los que solo analizamos el comportamiento y evolución de una cantidad finita de células situadas de forma adyacente ocupando posiciones de la forma $\{1, \dots, n\}$.

Con el fin de seguir dando una definición homogénea del autómata es necesario distinguir qué ocurre en la frontera del mismo (es decir, en las células que ocuparían las posiciones extremas adyacentes, 0 y $n+1$). Lo habitual es considerar que las posibles vecinas a estos extremos toman valores que son dependientes de la zona considerada por nuestro conjunto finito de células, y podemos distinguir entre:

- *Frontera abierta*: las células frontera toman un valor fijo: $c(0) = c(n) = s_0$.
- *Frontera reflectora*: las células frontera toman los valores que están dentro, como si fuera un espejo: $c(0) = c(1),\ c(n+1) = c(n)$.
- *Frontera circular*: las células frontera copian el contenido del otro extremo: $c(0) = c(n),\ c(n+1) = c(1)$ (Figura 2.9).

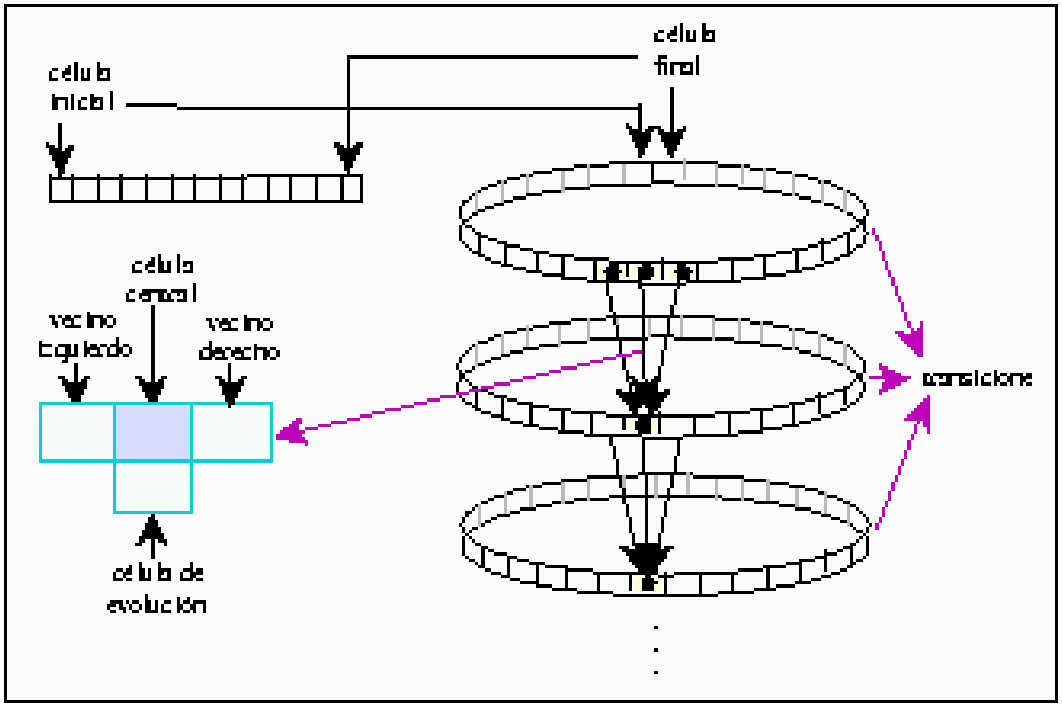


Figura 2.9: Frontera circular. Recuperada de www.cs.us.es

En el caso de que el autómata sea infinito se denomina *sin frontera* y no existen células frontera. En dicho caso, cada célula ocupa una posición de $\mathbb{Z}$.

En lo que sigue, consideraremos que nuestros autómatas tienen frontera circular.

Proposición 2.1. En las condiciones expuestas anteriormente ($d = 1$ y $|N| = 3$) existen únicamente 256 funciones de transición local posibles.

Demostración. Esto se debe a que la función de transición transforma el estado de tres células (la que estudiamos y su vecindad) en un nuevo estado. Esto significa que cada función de transición transforma cada uno de las 8 posibles combinaciones de estados (3 células con 2 posibles estados cada una) en un valor que puede ser 0 o 1. En consecuencia, tenemos 2^8 posibilidades. ∎

Notación 2.1. Cada función de transición la notaremos por un número entre 0 y 255 siguiendo el esquema que se especifica a continuación:

Dado el siguiente orden prefijado a las posibles configuraciones locales del autómata (que se corresponde con el orden natural de la representación decimal de las configuraciones):

$$c_1 = 000,\ c_2 = 001,\ c_3 = 010,\ c_4 = 011,\ c_5 = 100,\ c_6 = 101,\ c_7 = 110,\ c_8 = 111$$

y supuesto que la regla r se escribe en binario como $r = r_1 \dots r_8 \in \{0,1\}^8$, la regla r asigna a la configuración local c_i el valor r_i.

Por ejemplo, la regla 30, que en binario es 00011110, proporciona la función de transición siguiente:

$$r_{30}(000) = 0,\ r_{30}(001) = 0,\ r_{30}(010) = 0,\ r_{30}(011) = 1$$

$$r_{30}(100) = 1,\ r_{30}(101) = 1,\ r_{30}(110) = 1,\ r_{30}(111) = 0$$

Definición 2.2. *Sea A un autómata celular con función de transición local f. Podemos definir las siguientes transformaciones (donde $\overline{x} = 1 - x$):*

- *Transformación espejo:*

$$\mathcal{M}_f(x_{i-1}, x_i, x_{i+1}) = f(x_{i+1}, x_i, x_{i-1}) \forall x_i \in S$$

- *Transformación 0-1:*

$$\mathcal{C}_f(x_{i-1}, x_i, x_{i+1}) = \overline{f}(\overline{x_{i-1}}, \overline{x_i}, \overline{x_{i+1}}) \forall x_i \in S$$

- *Transformación conjunta:*

$$\mathcal{J}_f(x_{i-1}, x_i, x_{i+1}) = \overline{f}(\overline{x_{i+1}}, \overline{x_i}, \overline{x_{i-1}}) \forall x_i \in S$$

Nota 2.1. Con la representación binaria introducida anteriormente, si f viene dada por la cadena $(f_0, f_1, f_2, f_3, f_4, f_5, f_6, f_7)$, las reglas que se obtienen con las transformaciones anteriores son:

$$\begin{aligned} \mathcal{M}_f &= (f_0, f_4, f_2, f_6, f_1, f_5, f_3, f_7) \\ \mathcal{C}_f &= (\overline{f_7}, \overline{f_6}, \overline{f_5}, \overline{f_4}, \overline{f_3}, \overline{f_2}, \overline{f_1}, \overline{f_0}) \\ \mathcal{J}_f &= (f_7, f_3, f_5, f_1, f_6, f_2, f_4, f_0) \end{aligned}$$

Proposición 2.2. Utilizando las transformaciones de la definición anterior, podemos extraer únicamente 88 reglas independientes.

La tabla 2.1 muestra las distintas clases independientes que existen, y en las que podemos clasificar el conjunto de los 256 autómatas simples 1D. Por otra parte, en el apéndice (ver a partir de 6.1) habrá muestras de las 88 reglas.

Demostración. Sea r una regla de un autómata celular. Consideraremos los posibles resultados que se pueden dar al aplicar las anteriores transformaciones sobre la regla r:

- 1. $\mathcal{C}(r) = \mathcal{M}(r) = r$. Esto implica que $\mathcal{J}(r) = r$.
- 2. $\mathcal{C}(r) = \mathcal{M}(r) \neq r$. Esto implica que $\mathcal{J}(r) = \mathcal{C}(r) = \mathcal{M}(r) \neq r$.
- 3. $\mathcal{C}(r) = r \neq \mathcal{M}(r)$. Esto implica que $\mathcal{J}(r) = \mathcal{M}(r)$.
- 4. $\mathcal{C}(r) \neq r = \mathcal{M}(r)$. Esto implica que $\mathcal{J}(r) = \mathcal{C}(r)$.
- 5. $\mathcal{C}(r) \neq \mathcal{M}(r) \neq r$. Esto implica que $\mathcal{J}(r) \neq \mathcal{C}(r) \neq \mathcal{M}(r) \neq r$ (en el sentido de que todos son diferentes al resto). Se puede probar usando que $\mathcal{O}(\mathcal{O}(r)) = r$ con $\mathcal{O}$ una de las operaciones

El caso 1 da conjuntos de una sola regla. Los casos 2, 3 y 4 dan conjuntos con dos reglas. El caso 5 da conjuntos de 4 reglas. Por tanto, si n_1, n_2, n_3, n_4 y n_5 son el número de elementos en los distintos conjuntos respectivamente, entonces el total de reglas independientes es:

$$n_1 + \frac{(n_2 + n_3 + n_4)}{2} + \frac{n_5}{4}$$

Determinemos ahora los valores de n_i:

1. Se debe cumplir que $t_0 = \overline{t_7}$, $t_2 = \overline{t_5}$, $t_1 = t_4 = \overline{t_6} = \overline{t_3}$. Una vez fijados t_0, t_1 y t_2, el resto quedan determinados, luego hay 2^3 posibilidades, y por tanto $n_1 = 8$.
2. 2. Se debe cumplir que $t_0 = \overline{t_7}$, $t_4 = \overline{t_6}$, $t_2 = \overline{t_5}$ y $t_1 = \overline{t_3}$. Tenemos $2^4 = 16$ posibilidades, pero eliminando las del caso 1 tenemos $n_2 = 8$ posibilidades.
3. 3. Se debe cumplir que $t_0 = \overline{t_7}$, $t_1 = \overline{t_6}$, $t_2 = \overline{t_5}$ y $t_3 = \overline{t_4}$. Tenemos $2^4 = 16$ posibilidades, pero eliminando las del caso 1 tenemos $n_3 = 8$ posibilidades.
4. 4. Se debe cumplir que $t_1 = t_4$ y $t_3 = t_6$. Tenemos por tanto $2^6 = 64$ posibilidades, pero eliminando las del caso 1 tenemos $n_4 = 64 - 8 = 56$ posibilidades.
5. 5. El resto, que son $n_5 = 256 - 8 - 8 - 8 - 56 = 176$.

Finalmente, la cantidad de reglas independientes es:

$$8 + (8 + 8 + 56)/2 + 176/4 = 8 + 36 + 44 = 88$$

Esta demostración se ha extraído de [5].

Definición 2.3. *Definimos los siguientes conceptos:*

- *Se dice que $q \in S$ es un estado fijo si $f(q, ..., q) = q$.*
- *Una configuración c es fija si $c(x) = q$ para todo $x \in \mathbb{Z}^d$.*
- *Una configuración c es finita si solo un número finito de células no son fijas, es decir, si $\{x \in \mathbb{Z}^d | c(x) \neq q\}$ es finito.*

Clase	Reglas	Clase	Reglas
0	255	56	98,185,227
1	127	57	99
2	16,191,247	58	114,163,177
3	17,63,119	60	102,153,195
4	223	62	118,131,145
5	95	72	237
6	20,159,215	73	109
7	21,31,87	74	88,173,229
8	64,239,253	76	205
9	65,111,125	77	-
10	80,175,245	78	92,141,197
11	47,81,117	90	165
12	68,207,221	94	133
13	69,79,93	104	233
14	84,143,213	105	-
15	85	106	120,169,225
18	183	108	201
19	55	110	124,137,193
22	151	122	161
23	-	126	129
24	66,189,231	128	254
25	61,67,103	130	144,190,246
26	82,167,181	132	222
27	39,53,83	134	148,158,214
28	70,157,199	136	192,238,252
29	71	138	174,208,244
30	86,135,149	140	196,206,220
32	251	142	212
33	123	146	182
34	48,187,243	150	-
35	49,59,115	152	188,194,230
36	219	154	166,180,210
37	91	156	198
38	52,155,211	160	250
40	96,235,249	162	176,186,242
41	97,107,121	164	218
42	112,171,241	168	224,234,248
43	113	170	240
44	100,203,217	172	202,216,228
45	75,89,101	178	-
46	116,139,209	184	226
50	179	200	236
51	-	204	-
54	147	232	-

Cuadro 2.1: Tabla de clases independientes

- *Denotaremos por* $\mathcal{C}_F(d, S)$ *al conjunto de las configuraciones finitas.*

Proposición 2.3 (Clasificación de Wolfram). Los autómatas celulares pueden clasificarse en cuatro familias según su comportamiento. Esta clasificación se debe a Wolfram y se encuentra en [6]:

- **Clase I**: Casi todas las configuraciones iniciales evolucionan a un estado estable y homogéneo. Cualquier aleatoriedad desaparece.
- **Clase II**: Casi todas las configuraciones iniciales evolucionan rápidamente a estructuras estables u oscilantes.
- **Clase III**: Casi todas las configuraciones iniciales evolucionan de forma pseudoaleatoria o caótica. Cualquier estructura estable desaparece por el ruido.
- **Clase IV**: Casi todas las configuraciones iniciales evolucionan en estructuras que interaccionan de forma compleja. Wolfram conjeturó que muchos, cuando no todos, de los autómatas de esta clase son computadores universales. Esto se ha probado para la regla 110 y el juego de la vida de Conway.

Aunque en los diagramas de la figura 2.10 hay autómatas celulares de una dimensión pero diferentes órdenes, sirve para ilustrar las clases de Wolfram.

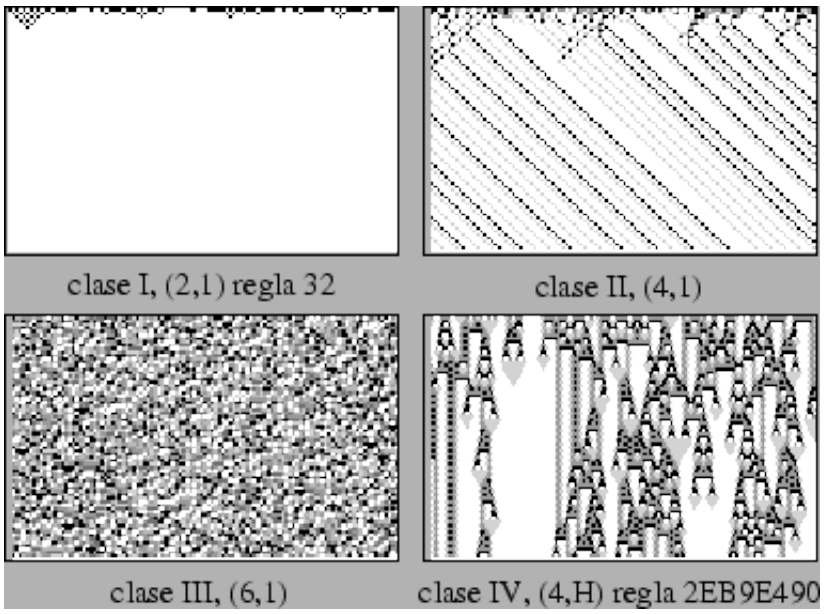


Figura 2.10: Ejemplos de las clases de Wolfram. Recuperada de delta.cs.cinvestav.mx

Proposición 2.4. Determinar a qué familia pertenece un autómata es un problema indecidible, es decir, no existe un algoritmo que permita clasificar cada Autómata

Celular en la familia a la que corresponde.

Este resultado fue desarrollado por Culik y Yu utilizando una clasificación más precisa, que definiremos a continuación. Es de especial interés en el campo que estudiamos, ya que aunque utilizando Machine Learning no podremos determinar con exactitud a qué clase pertenece, podremos intentar determinar la clase con cierta probabilidad.

Demostración. Una prueba de este resultado se encuentra en [2]. ∎

Esta otra clasificación se debe a Culik y a Yu, e intenta formalizar la clasificación de Wolfram con el fin de estudiar más concienzudamente algunas propiedades de los autómatas celulares. Esta clasificación se encuentra en [2].

El objetivo de esta clasificación es separar los autómatas que se mantienen estables y los que oscilan de los que presentan patrones más complejos, y luego separar los más complejos entre los que pueden simular una máquina de Turing universal (clase 4) y aquellos que no (clase 3) con el fin de estudiar sus propiedades.

Las clases de Culik-Yu se corresponden por tanto con las clases de Wolfram.

Definición 2.4 (Clasificación de Culik-Yu). *Las clases de Culik-Yu se establecen como:*

- ***CY1**: Todas las configuraciones finitas evolucionan a una configuración fija Q, es decir, para cada $c \in C_F$ existe un $n \geq 1$ tal que $G^n(c) = Q$. Algunos ejemplos son los autómatas 0, 4 y 16.*
- ***CY2**: Todas las configuraciones finitas son eventualmente periódicas, esto es, para cada $c \in C_F$ existen m y n, $m \neq n$ tales que $G^m(c) = G^n(c)$. Algunos ejemplos son los autómatas 8, 24 y 40.*
- ***CY3**: Existe un algoritmo que determina si una configuración finita dada pertenece a la órbita de otra configuración finita dada, es decir, si es decidible el problema de, dados $c_1,\ c_2 \in C_F$ decidir si $G^n(c_1) = c_2$ para algún $n \geq 1$. Un ejemplo es la regla 30.*
- ***CY4**: El resto de autómatas celulares. Un ejemplo de Autómata Celular Universal se encuentra en [7]*

Esta clasificación se debe a Li y Packard, y reparte los autómatas en seis clases en lugar de cuatro como hacen las anteriores. Se puede encontrar en [3].

Definición 2.5 (Clasificacion de Li-Packard). *Las clases de Li-Packard se establecen como:*

- ***Null**: Se alcanza una configuración homogénea, con todas las células teniendo el mismo valor.*
- ***Punto fijo**: La configuración final es invariante después de aplicar la regla una vez. También se incluyen en esta clase las reglas que simplemente desplazan los patrones.*
- ***Dos ciclos**: La configuración final es invariante después de aplicar la regla dos veces.*
- ***Periódicos**: La configuración final es invariante después de aplicar la regla un total de k veces, con el ciclo k independiente (o débilmente dependiente) del número de células.*
- ***Complejos**: Pueden tener algunas configuraciones finales periódicas, pero el tiempo requerido para encontrarlas puede ser enorme. Este tiempo aumenta linealmente con el número de células.*
- ***Caóticos**: Tienen lugar dinámicas no periódicas, caracterizados por su inestabilidad respecto a perturbaciones en la configuración inicial.*

En realidad, lo que hace la clasificación de Li-Packard es separar la clase II de Wolfram en tres trozos: Punto fijo, Dos ciclos y Periódicos. Las otras tres clases se conservan: la clase I se corresponde con Null, la clase III con Complejos y la clase IV con Caóticos.

3 Machine Learning

3.1 Introducción

El término *Machine Learning* (ML), o *aprendizaje automático* en español, hace referencia a un área de la Inteligencia Artificial que tiene como objetivo desarrollar técnicas que permitan a las computadoras aprender. Desde un punto de vista pragmático, este ambicioso objetivo se ha transformado en dar procedimientos para la detección automática de patrones significativos en un conjunto de datos o de generalización de comportamientos a partir de una información suministrada en forma de ejemplos.

El término Machine Learning fue usado por primera vez en 1959[10], y se debe a Arthur L. Samuel, quien desarrolló en 1952 un programa que aprendía a jugar a las damas. El ordenador que utilizaba mejoraba cuanto más jugaba, analizando las mejores jugadas que se habían dado en partidas anteriores e incorporándolas a su juego. Samuel siguió mejorando su programa hasta mediados de los 70, momento en que ya era capaz de ganar a un jugador amateur experto.

En 1951 Marvin Minsky creó el SNARC: la primera red neuronal de la historia. El SNARC (*Stochastic neural analog reinforcement calculator*) tenía 40 neuronas como la de la figura 3.1, y simulaba el comportamiento de ratones que se movían por un laberinto. Minsky vio que por un error en el diseño electrónico podía incluir dos o tres "ratones" al mismo tiempo en su máquina[11], y que los ratones aprendían de los otros a la hora de encontrar la salida del laberinto.

Un logro reciente del Machine Learning ha sido el desarrollo de AlphaGo, capaz de derrotar a 60 jugadores profesionales de Go seguidos en 2016, batiendo además al mejor jugador del mundo, Ke Jie, tras ganarle cada una de las cinco partidas que jugaron. El Go está considerado como el juego de tablero más complejo del mundo, superando en dificultad al ajedrez.

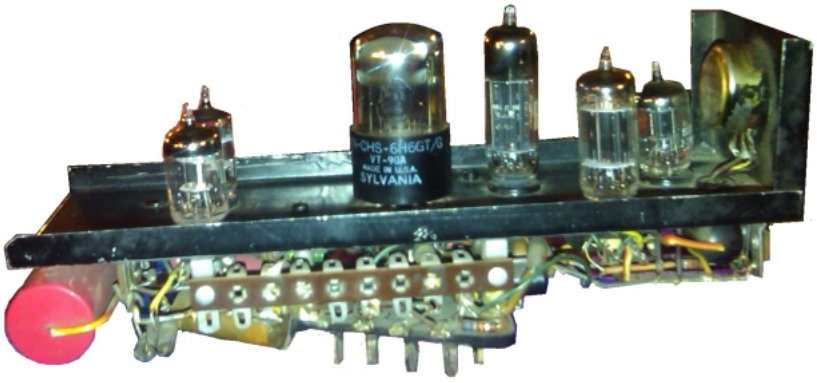


Figura 3.1: Una de las neuronas del SNARC. Recuperada de cyberneticzoo.com

En los últimos años el Machine Learning ha demostrado ser una herramienta extremadamente útil en diversos ámbitos: los motores de búsqueda proporcionan cuáles son los mejores resultados para nuestras búsquedas, además de buscar cuáles son los anuncios más acordes a nuestros intereses; las cámaras aprenden a detectar caras, los teléfonos móviles aprenden a reconocer comandos de voz, etc... Un ejemplo interesante es el publicado en [12]. Para dicho estudio, se han entrenado algunos modelos de aprendizaje con casi 300.000 imágenes de fondos de ojo. Estos modelos, al introducirle nuevas imágenes, son capaces de predecir correctamente características como el género el 97 % de las veces, si fuman el 71 % de las veces, y si ha habido algún evento cardíaco de importancia el 70 % de las veces.

Una característica común a todas estas aplicaciones es que, normalmente, debido a la complejidad de los patrones subyacentes a los problemas que se intentan resolver, un programador humano no podría especificar de forma explícita cómo deben resolverse estos problemas. En el ejemplo descrito el párrafo anterior, se pensaba que no había diferencia entre el ojo masculino y el femenino, pero si la máquina es capaz de distinguirlos con una probabilidad de 0.97, significa que está detectando algo que no somos capaces de discernir.

La idea que engloba a los métodos de Machine Learning es, como su nombre indica, que los programas "aprendan" de los distintos casos que se les plantean, y adapten sus respuestas a éstos, de una forma similar a como el ser humano (y otros animales superiores) lo hacen a lo largo de su vida respecto a los problemas que encontramos. El mecanismo básico se muestra en la figura 3.2.

Interpretado el Machine Learning como una herramienta computacional de bús-

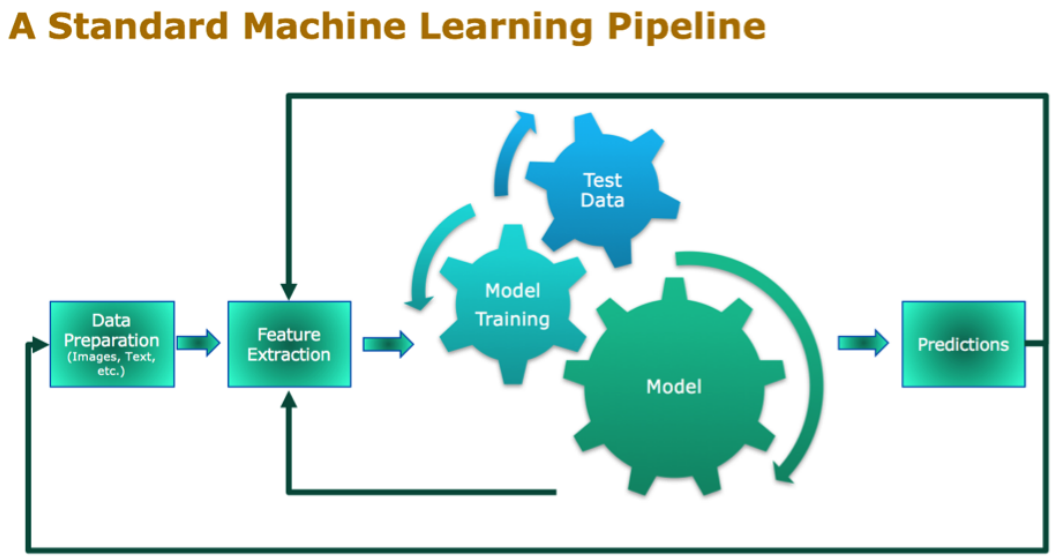


Figura 3.2: Esquema del funcionamiento de un algoritmo de ML. Recuperada de www.datanami.com

queda de patrones complejos, podemos justificar claramente la razón por la que en este trabajo se introduce un capítulo de fundamentos de esta área: debido a las características de los Sistemas Complejos, como los autómatas celulares vistos en el capítulo anterior, las herramientas de Machine Learning pueden resultar una gran ayuda al aportar potencia suficiente para detectar patrones que para nosotros sería imposible reconocer.

Atendiendo a los objetivos que tienen los diversos algoritmos de ML que se han desarrollado, se suele considerar una clasificación que agrupa estos algoritmos en dos grandes tipos (aunque también es normal considerar algunas otras agrupaciones y otros tipos de clasificaciones): los de *aprendizaje supervisado*, y los de *aprendizaje no supervisado*. Los algoritmos de aprendizaje supervisado se llaman así porque se entrenan con un conjunto de datos para los que se sabe el resultado, y el algoritmo puede usar este valor de salida para optimizar su funcionamiento, y devuelven una función que predecirá el resultado correcto a partir de ese momento con mayor o menor precisión. Los de aprendizaje no supervisado se llaman así porque sólo se introducen datos de entrada, y no se conoce ningún resultado para estos, por lo que suelen buscarse agrupaciones en los datos, o patrones específicos que atiendan a la estructura interna de los mismos.

Por la naturaleza de nuestro problema (somos capaces de calcular iteraciones de

los autómatas celulares a partir de una regla y una configuración inicial), utilizaremos fundamentalmente algoritmos de aprendizaje supervisado, aunque también podrían utilizarse algunos de aprendizaje no supervisado para reconocer, por ejemplo, clasificaciones interesantes entre las familias y clasificaciones de autómatas que vimos en el capítulo anterior.

En este capítulo se explicarán los fundamentos de los distintos algoritmos de aprendizaje que aplicaremos más adelante a los autómatas celulares. El contenido de este capítulo no intenta ser, ni mucho menos, exhaustivo, sino que se proporciona únicamente con el fin de acotar el contexto de ML del que haremos uso a lo largo del resto del trabajo, así como la notación usada. En las referencias que se proporcionan a lo largo de las siguientes secciones se puede encontrar direcciones para quien desee disponer de un desarrollo más amplio de los temas aquí tratados.

En los experimentos concretos que haremos en capítulos sucesivos, se hará uso de las implementaciones de los algoritmos señalados que vienen en la librería de código libre *scikit-learn* para Python.

3.2 Aproximación Matemática al Aprendizaje Automático

El objetivo de esta sección es doble: formalizar nuestro modelo de aprendizaje, y probar bajo qué condiciones nuestros algoritmos nos proveerán de soluciones aproximadas lo suficientemente buenas.

El primer objetivo es definir un modelo formal de aprendizaje en el que basar los algoritmos que se estudiarán más tarde. Un modelo formal es el siguiente, el cual se encuentra en [8]:

Definición 3.1 (Modelo formal de aprendizaje estadístico). *Nuestros algoritmos se definen mediante los siguientes elementos:*

- ***Entrada del algoritmo:***
  - ***Dominio:*** *Un conjunto arbitrario, $\mathcal{X}$. Este es el conjunto de objetos que deseamos etiquetar. Normalmente, cada elemento de $\mathcal{X}$ representa un vector de características.*
  - ***Conjunto de etiquetas:*** *Un conjunto $\mathcal{Y}$ con el que se pueden etiquetar cada uno de los elementos del dominio. Normalmente $\mathcal{Y} = \{0, 1\}$, aunque puede tomar otros*

valores.

- ***Conjunto de entrenamiento:*** $S = \left(\left(x_1, y_1\right), ..., \left(x_m, y_m\right)\right)$ *es una secuencia de pares de* $\mathcal{X}x\mathcal{Y}$. *Esto es, es una secuencia de puntos del dominio ya etiquetados. Estos son los datos de entrada a los que tiene acceso nuestro algoritmo de aprendizaje.*

- ***Salida del algoritmo:*** *El algoritmo debe devolver un predictor,* $h : \mathcal{X} \to \mathcal{Y}$. *Esta función puede usarse para etiquetar nuevos puntos del dominio. Denotaremos* $A(S)$ *al predictor que devuelve el algoritmo* A *tras entrenarlo con el conjunto de entrenamiento* S.

- ***Un modelo de generación de datos:*** *Se asume en primer lugar que los elementos del dominio están generados por alguna distribución de probabilidad. Denotamos la distribución sobre* $\mathcal{X}$ *por* $\mathcal{D}$. *Asumimos también que existe una función de etiquetado "correcta", es decir, una función* $f : \mathcal{X} \to \mathcal{Y}$ *de modo que* $y_i = f\left(x_i\right)$ *para cada* i.
  Es importante observar que no es necesario que $\mathcal{D}$ *ni* f *sean conocidos por el algoritmo (de hecho,* f *es lo que se pretende averiguar).*

- ***Medida de error:*** *Definimos el error de un predictor como la probabilidad de que, para un elemento de* $\mathcal{X}$, *el predictor no sea capaz de determinar correctamente su etiqueta.*
  Formalmente, dado $A \subset \mathcal{X}$, *la distribución* $\mathcal{D}$ *asigna un número* $\mathcal{D}(A)$, *el cual determina la probabilidad de observar un punto* $x \in A$.
  Se define el error de un predictor $h : \mathcal{X} \to \mathcal{Y}$ *como:*

$$L_{D,f}(h) = \mathbb{P}_{x\sim D}\left[h(x) \neq f(x)\right] = \mathcal{D}\left(\left\{x : h(x) \neq f(x)\right\}\right)$$

Debido a que normalmente no se conocen $\mathcal{D}$ ni f, el verdadero error no puede calcularse. Por este motivo, se buscará minimizar el error empírico, que se puede medir de varias formas, aunque la forma más corriente sea:

$$L_S(h) = \frac{\left|\left\{i \in [m] : h\left(x_i\right) \neq y_i\right\}\right|}{m}$$

con $[m] = 1, .., m$

Definición 3.2 (ERM). *Se llama Minimización de Error Empírico (en inglés, ERM: Empirical Risk Minimization), al paradigma de aprendizaje basado en encontrar un predictor* h_S *que minimice* $L_S(h)$.

Aunque el paradigma ERM parece natural, si no se tiene cuidado pueden darse situaciones en las que nuestros algoritmos fallen. Una de estas situaciones es que se de sobreajuste, o *overfitting* en inglés.

El sobreajuste se da cuando nuestro algoritmo aprende demasiado bien los datos del conjunto de entrenamiento, y se ajusta tanto a ellos que no predice correctamente el comportamiento del resto de elementos del dominio.

Por ejemplo, el predictor:

$$h_S(x) = \begin{cases} y_i & si \quad \exists i \in [m] : x_i = x \\ 0 & cc \end{cases}$$

cumple que $L_S\left(h_S\right) = 0$, pero su predicción del resto del dominio será bastante pobre.

En [8] se prueba que si tomamos una clase de predictores $\mathcal{H}$ finita a la que restringir la búsqueda de predictores para nuestro algoritmo y un conjunto de entrenamiento lo suficientemente grande, entonces el predictor que obtenemos es probablemente correcto (como aproximación).

La importancia de este hecho reside en que cualquier predictor que podamos programar en un computador está limitado por un número de bytes dado por el computador. Esto hace que nuestra clase de funciones sea finita, y por tanto se apliquen estos resultados.

Con esta idea en mente, definimos el siguiente tipo de aprendizaje:

Definición 3.3 (Supuesto de realización). *Una clase de hipótesis $\mathcal{H}$ cumple el supuesto de realización si para algún $h \in \mathcal{H}$, se tiene que $L_{(D,f)}(h) = 0$.*

Definición 3.4 (Aprendizaje PAC). *Una clase de hipótesis $\mathcal{H}$ es PAC (Probably Approximately Correct) aprendible si existe una función $m_{\mathcal{H}} : (0,1)^2 \to \mathbb{N}$ y un algoritmo de aprendizaje con la siguiente propiedad:*

Para cada $\epsilon, \delta \in (0,1)$, para cada distribución $\mathcal{D}$ sobre $\mathcal{X}$, y para cada función de etiquetado $f : \mathcal{X} \to 0,1$, si se cumple el supuesto de realización respecto a $\mathcal{H}, \mathcal{D}, f$, entonces cuando se ejecuta el algoritmo sobre $m \geq m_{\mathcal{H}}(\epsilon, \delta)$ ejemplos i.i.d. por $\mathcal{D}$ y etiquetados por f, entonces el algoritmo devuelve un predictor h tal que, con probabilidad al menos $1 - \delta$ sobre la elección de los ejemplos, cumple que:

$$L_{(D,f)}(h) \leq \epsilon$$

Proposición 3.1. Toda clase finita de hipótesis es PAC aprendible con

$$m_{\mathcal{H}}(\epsilon, \delta) \leq \left\lceil \frac{log\left(|\mathcal{H}|/\delta\right)}{\epsilon} \right\rceil$$

Demostración. Se encuentra en [8]. ∎

Llegados a este punto, es interesante estudiar qué ocurre cuando eliminamos alguna de las condiciones. Por ejemplo, ¿qué ocurre al eliminar el supuesto de realización?

Definición 3.5 (Aprendizaje PAC agnóstico). *Una clase de hipótesis $\mathcal{H}$ es PAC agnóstica aprendible si existe una función $m_{\mathcal{H}} : (0,1)^2 \to \mathbb{N}$ y un algoritmo de aprendizaje con la siguiente propiedad:*

Para todos $\epsilon, \delta \in (0,1)$ y para cada distribución $\mathcal{D}$ sobre Z, cuando se ejecuta el algoritmo de aprendizaje sobre $m \geq m_{\mathcal{H}}(\epsilon, \delta)$ ejemplos i.i.d. generados por $\mathcal{D}$, el algoritmo devuelve $h \in \mathcal{H}$ tal que, con probabilidad al menos $1 - \delta$ sobre la elección de los m ejemplos de entrenamiento, se tiene que:

$$L_D(h) \leq min_{h' \in D} L_D(h') + \epsilon$$

Esto nos indica que, en caso de que no se cumpla el supuesto de realización, entonces no podremos garantizar que nuestro algoritmo de aprendizaje tenga un error arbitrariamente pequeño, pero sí podemos garantizar que el error de este no sea muy grande en comparación al mejor predictor que podemos obtener.

Hasta este momento hemos usado la función de error L que consideramos al principio. Sin embargo, el aprendizaje PAC agnóstico puede generalizarse a cualquier función que mida el error.

Definición 3.6 (Funciones de error). *Sea $\mathcal{H}$ una familia de predictores y Z un dominio. Llamamos función de error (loss function) a cualquier función $\ell : \mathcal{H}xZ \to \mathbb{R}_+$.*

Algunos ejemplos de funciones de error son:

- **Error 0-1:** Para esta función $Z = \mathcal{X}x\mathcal{Y}$, y se tiene que:

$$\ell\left(h,(x,y)\right) = \begin{cases} 0 & si \quad h(x) = y \\ 1 & si \quad h(x) \neq y \end{cases}$$

- **Error cuadrático:** Para esta función $Z = \mathcal{X}x\mathcal{Y}$, y se tiene que:

$$\ell\left(h,(x,y)\right) = (h(x) - y)^2$$

Definición 3.7 (Aprendizaje PAC agnóstico generalizado). *Una clase de predictores $\mathcal{H}$ es PAC agnóstica aprendible respecto a un conjunto Z y a una función de error $\ell : \mathcal{H}xZ \to \mathbb{R}_+$, si existe una función $m_{\mathcal{H}} : (0,1)^2 \to \mathbb{N}$ y un algoritmo de aprendizaje con la siguiente propiedad:*

Para cada $\epsilon, \delta \in (0,1)$ y cada distribución $\mathcal{D}$ sobre Z, cuando se ejecuta el algoritmo sobre $m \geq m_{\mathcal{H}}(\epsilon, \delta)$ i.i.d. ejemplos generados por $\mathcal{D}$, el algoritmo devuelve $h \in \mathcal{H}$ tal que, con probabilidad al menos $1-\delta$ sobre la elección de los m ejemplos de entrenamiento, es

$$L_D(h) \leq min_{h' \in \mathcal{H}} L_D(h') + \epsilon$$

donde $L_D(h) = \mathbb{E}_{z \sim D}[\ell(h, z)]$.

Una vez presentados los fundamentos anteriores, existe una pregunta fundamental que surge de manera natural: ¿existe un algoritmo de aprendizaje universal? Es decir, ¿existe un algoritmo de aprendizaje que pueda aprender bien cualquier problema? La respuesta, como era de esperar, es que no. Esto queda probado por el teorema *No-free-lunch* que enunciamos a continuación:

Teorema 3.1 (No-free-lunch). *Sea A un algoritmo de aprendizaje cualquiera para la tarea de clasificación binaria respecto al error 0-1 sobre un dominio $\mathcal{X}$. Sea m un número menor que $|\mathcal{X}|/2$, el cual representa el tamaño de un conjunto de entrenamiento. Entonces existe una distribución $\mathcal{D}$ sobre $\mathcal{X}x\{0,1\}$ tal que:*

1. *Existe una función $f : \mathcal{X} \to \{0,1\}$ tal que $L_D(f) = 0$*
2. *Con probabilidad al menos 1/7 sobre la elección de $S \sim \mathcal{D}^m$ se tiene que:*

$$L_D(A(S)) \geq 1/8$$

Demostración. Una prueba de este teorema se encuentra en [8]. ∎

Este resultado implica que, para cada aprendiz, existe una tarea para la que fracasa al intentar aprenderla, aunque probablemente exista otro aprendiz que sea capaz de aprenderla sin problemas.

Nos enfrentamos ahora a un nuevo problema: ¿Cómo podemos elegir una clase de predictores que nos permita prevenir estos fallos? Nos enfrentamos además a la siguiente proposición:

Proposición 3.2. Sea $\mathcal{X}$ un conjunto infinito y $\mathcal{H}$ la clase de funciones que van de $\mathcal{X}$ a $\{0,1\}$. Entonces $\mathcal{H}$ no es PAC aprendible.

Por tanto, no podremos tomar clases de predictores demasiado ricas. La solución la encontramos en el llamado *bias-complexity tradeoff*.

Definición 3.8 (Bias-complexity tradeoff). *El error de un predictor $ERM_{\mathcal{H}}$ se puede descomponer en dos componentes como sigue:*

Sea h_S una hipótesis $ERM_{\mathcal{H}}$. Entonces se puede escribir:

$$L_D = \epsilon_{app} + \epsilon_{est}$$

donde $\epsilon_{app} = min_{h \in \mathcal{H}} L_D(h)$ es el error de aproximación, y $\epsilon_{est} = L_D(h_S) - \epsilon_{app}$ es el error de estimación:

- ***Error de aproximación:*** *Es el menor error que puede lograr un predictor de una clase de hipótesis. Este término mide cuál es el error que cometemos al ajustarnos a una clase de hipótesis específica. Agrandar la clase de hipótesis puede reducir el error de aproximación.*
- ***Error de estimación:*** *El error de estimación resulta de que el error empírico es sólo una estimación del error real. La calidad de esta estimación depende del tamaño del conjunto de entrenamiento, y de la complejidad de la clase de hipótesis.*

Como nuestra intención es minimizar el error total, nos enfrentamos a un equilibrio entre ambos errores. Por una parte, enriquecer la clase $\mathcal{H}$ puede decrecer el error de aproximación pero incrementar el error de estimación, ya que una clase rica puede llevar a que se produzca un sobreajuste. Por otra parte, si tomamos $\mathcal{H}$ demasiado pequeño, incrementamos el error de aproximación pero decrece el error de estimación.

3.3 Algoritmos

En esta sección se describen los algoritmos que se aplicarán más adelante a los datos de los autómatas celulares. La definición de estos algoritmos se ha obtenido de [8] y de [9].

3.3.1 Predictores lineales

Comenzaremos presentando la familia de los *predictores lineales*, que se clasifica dentro del aprendizaje supervisado. Es una de las familias más útiles, y muchos algoritmos de aprendizaje los utilizan. Los predictores lineales son intuitivos, fáciles de

interpretar, y ajustan los datos razonablemente bien en muchos problemas de aprendizaje.

Definición 3.9. *Se define como* $L_d = \{h_{w,b} : w \in \mathbb{R}^d, b \in \mathbb{R}\}$ *a la clase de funciones lineales afines* $h : \mathbb{R}^d \to \mathbb{R}$*, de modo que* $h_{w,b}(x) = \langle w, x\rangle + b = \left(\sum_{i=1}^{d} w_i x_i\right) + b$.

Lema 3.1. Cualquier función lineal afín de $\mathbb{R}^d$ puede convertirse en una función lineal homogénea de $\mathbb{R}^{d+1}$.

Demostración. Definimos $w' = \left(b, w_1, ..., w_d\right) \in \mathbb{R}^{d+1}$ y $x' = \left(1, x_1, x_2, ..., x_d\right) \in \mathbb{R}^{d+1}$. Entonces se tiene que $h_{w,b}(x) = \langle w, x\rangle + b = \langle w', x'\rangle$.

Definición 3.10. *Se define a la clase de los semiespacios como el conjunto* $HS_d = sign \circ L_d = \{x \mapsto sign\left(h_{w,b}(x)\right) : h_{w,b} \in L_d\}$*. Esta clase está diseñada para problemas de clasificación binaria.*

Algoritmo 3.1 (Mínimos cuadrados). *Mínimos cuadrados es el algoritmo que resuelve el problema ERM para la clase de hipótesis de los predictores de regresión lineales respecto del error cuadrático medio. El problema ERM, dado un conjunto de entrenamiento* S *y usando la versión homogénea de* L_d*, es encontrar (Figura 3.3):*

$$argmin_w L_S\left(h_w\right) = argmin_w \frac{1}{m}\sum_{i=1}^{m}\left(\langle w, x_i\rangle - y_i\right)^2$$

Para resolverlo se calcula el gradiente de la función objetivo y se compara con cero. Es decir, hay que resolver:

$$\frac{2}{m}\sum_{i=1}^{m}\left(\langle w, x_i\rangle - y_i\right)x_i = 0$$

Podemos reescribirlo como el problema $Aw = b$ *donde*

$$A = \left(\sum_{i=1}^{m} x_i x_i^T\right)$$

$$b = \sum_{i=1}^{m} y_i x_i$$

Si A *es invertible, entonces la solución al problema ERM es*

$$w = A^{-1}b$$

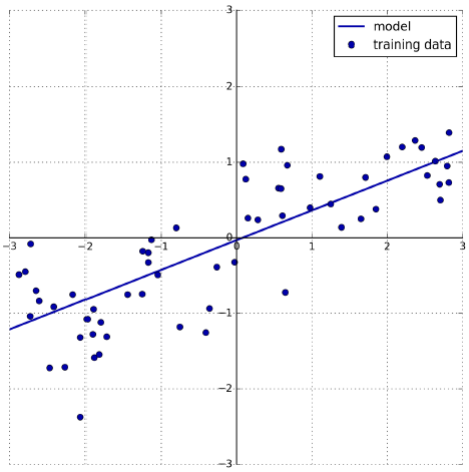


Figura 3.3: Ejemplo de regresión por mínimos cuadrados. Imagen extraída de [9]

En caso de que A no sea invertible, se pueden usar herramientas de álgebra lineal para encontrar $A = PDP^T$, con D una matriz diagonal y P una matriz ortogonal. Se define entonces $A^+ = PD^+P^T$ con :

$$D^+_{i,i} = \begin{cases} 0 & , si\ D_{i,i} = 0 \\ 1/D_{i,i} & , en\ otro\ caso \end{cases}$$

Entonces, si $\hat{w} = A^+b$, se tiene que:

$$A\hat{w} = AA^+b = VDV^TVD^+V^Tb = VDD^+V^Tb = \sum_{i\,:\,D_{i,i}\neq 0} v_i v_i^T b$$

Como $A\hat{w}$ es la proyección de b en el subespacio generado por los v_i para los cuales $D_{i,i} \neq 0$, y este subespacio es el mismo que el generado por los x_i, se tiene que $A\hat{w} = b$.

Existen variantes interesantes de este algoritmo que imponen restricciones adicionales para acotar el espacio de soluciones resultante. Por ejemplo, los dos que nombramos a continuación:

Algoritmo 3.2 (Regresión de arista). *La regresión de arista es otro modelo lineal de regresión en el que se usa también el error cuadrático medio. Sin embargo, en este caso los coeficientes (w) intentan minimizarse lo máximo posible. Esto se puede conseguir, por ejemplo, penalizando aquellas soluciones que proporcionen una norma L2 de los coeficientes demasiado elevada. La penalización se controlará con un parámetro C, que en el caso $C = 0$ resultará en el mismo método de mínimos cuadrados visto anteriormente.*

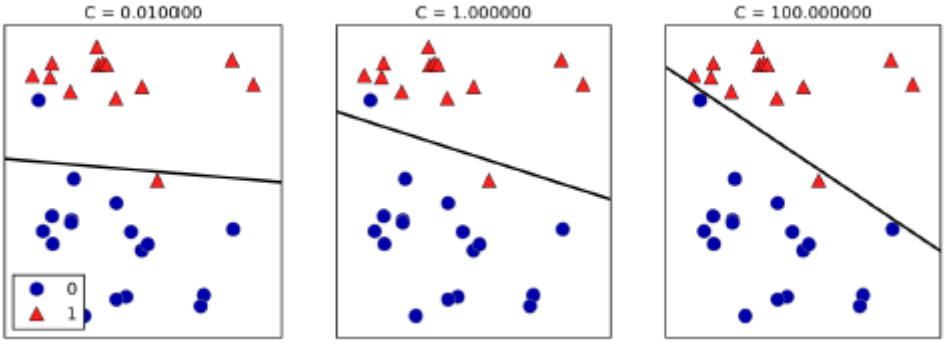


Figura 3.4: Ejemplos de regresión de arista con distintas C. Imagen extraída de [9]

Los efectos del parámetro C pueden observarse en la figura 3.4, que muestra que cuanto más grande es C, más trata de ajustarse a los datos.

Algoritmo 3.3 (Lasso). *Lasso es una alternativa a la regresión de arista. Es idéntico, salvo porque minimizará la norma L1 de los coeficientes en lugar de la norma L2.*

El hecho de usar L1 en vez de L2 hará que algunos coeficientes puedan llegar a ser exactamente 0, lo que se traducirá en que algunas de las características del modelo se ignorarán completamente. Las consecuencias son modelos más simples pero que muestran cuáles son las características más importantes en los datos de entrada.

Nota 3.1. Estos modelos pueden usarse como modelos de clasificación binaria. En este caso, la predicción se realizará usando la siguiente fórmula:

$$y = \langle w, x \rangle + b > 0$$

De esta forma, si el resultado es mayor que 0, entonces se asignará la clase 1, y si es menor que 0, se asignará la clase -1.

Los dos algoritmos lineales de clasificación más comunes son la regresión logística y las máquinas de soporte vectorial lineales. Las máquinas de soporte vectorial se explicarán en el correspondiente apartado más adelante.

Algoritmo 3.4 (Regresión logística). *La regresión logística utiliza una función sigmoidea para aprender una serie de funciones h de $\mathbb{R}^d$ en el intervalo $[0, 1]$. La clase de hipótesis asociada es la composición de dicha sigmoidea con la clase L_d. En particular, se usa la función logística, definida como:*

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

La clase de hipótesis es, por tanto:

$$H_{sig} = \phi_{sig} \circ L_d = \{x \to \phi_{sig}(\langle w, x \rangle) : w \in \mathbb{R}^d\}$$

El problema ERM asociado con la regresión logística se traduce en:

$$argmin_{w \in \mathbb{R}^d} \frac{1}{m} \sum_{i=1}^{m} log\left(1 + e^{-y_i \langle w, x_i \rangle}\right)$$

3.3.2 k-Vecinos más cercanos

Este algoritmo es, quizás, el más simple de todos los algoritmos de aprendizaje automático de clasificación que vamos a considerar, pero ofrece resultados razonablemente buenos en muchos casos en los que, incluso, algoritmos más complejos se encuentran con problemas. Para construir este modelo sólo hace falta almacenar los datos del conjunto de entrenamiento y elegir cuántos vecinos se considerarán en su vecindario.

Algoritmo 3.5. *Sea $S = \{(x_i, y_i) : i \in D\}$ un conjunto de entrenamiento, donde $S_x = \{x_i : i \in D\}$ son puntos de un espacio métrico, e y_i es la clasificación asociada a cada uno de ellos. Para clasificar un nuevo dato x se calculan los k puntos de S_x más cercanos a x, y se le asigna a x la clase más representada entre esos k puntos.*

En la figura 3.5 se muestra un ejemplo de aplicación de este algoritmo para $k = 3$. En el caso de clasificación binaria, es habitual considerar k impar para evitar empates en la asignación de clases.

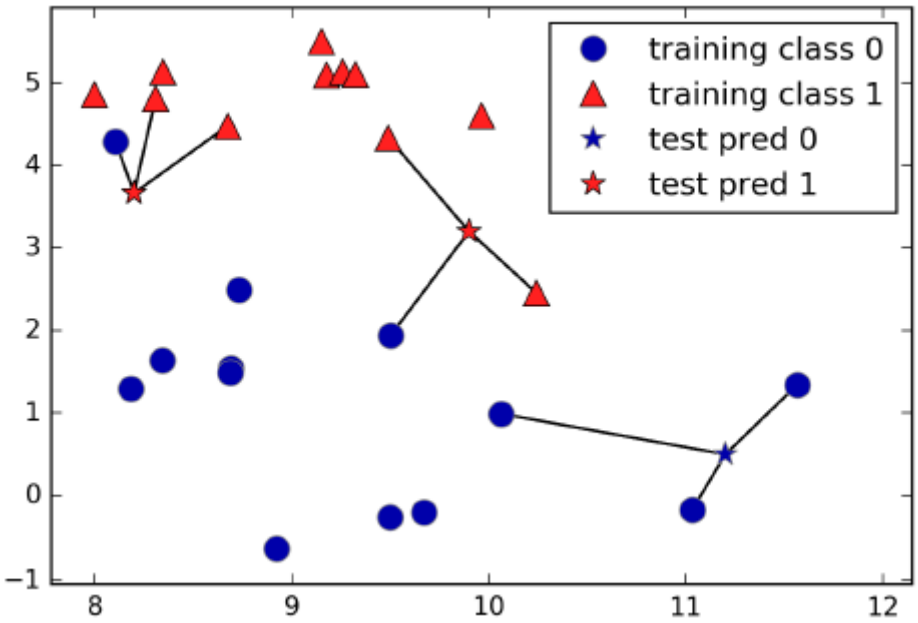


Figura 3.5: 3-Vecinos más cercanos. Imagen extraída de [9]

A pesar de su simplicidad, este algoritmo tiene un grave inconveniente, y es que requiere que el conjunto de entrenamiento esté almacenado en todo momento, y no proporciona una solución explícita (un modelo) que pueda ser reutilizado posteriormente. En este sentido, es un representante de aproximación no paramétrica del ML.

Además, en principio es necesario escanear el conjunto entero cada vez que se clasifica un nuevo dato con el objetivo de encontrar los vecinos más cercanos, aunque para mitigar este problema se puede hacer una búsqueda aproximada, limitando el radio de búsqueda.

Si consideramos el caso $k = 1$, se puede probar que cada punto $x_i \in S_x$ genera un volumen poliédrico a su alrededor (el conjunto de puntos más cercanos a él, y que se clasificarán en este caso como y_i) que se corresponde con el diagrama de Voronoi asociado a S_x.

Si aumentamos el valor de k, la frontera entre ambas zonas de clasificación se suaviza, tal y como se muestra en la Figura 3.6.

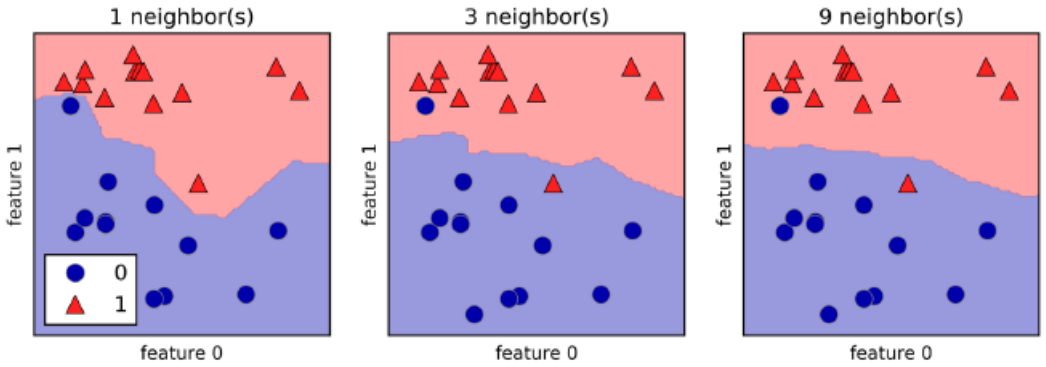


Figura 3.6: Fronteras en el problema de clasificación para distintos valores de k. Imagen extraída de [9]

3.3.3 Árboles de decisión

Un *árbol de decisión* es un predictor capaz de predecir la etiqueta asociada a un valor por medio de un proceso de decisión que se puede representar por medio de un camino en un árbol binario que conecta el nodo raíz del árbol con una de sus hojas.

Para ello, se considera en cada nodo interno del árbol (que no es hoja) habrá un predicado que evaluará el dato de entrada. Si el dato verifica el predicado, pasará a una de las ramas que salen del nodo, y si es falso pasaremos a otra. Las hojas del árbol se asocian a diferentes clasificaciones para el problema, por lo que cuando lleguemos a una de ellas se asignará al dato de entrada la clasificación dada por la hoja de llegada.

Por la naturaleza de los árboles de decisión, la construcción de estos no es única, y a veces obtendremos árboles muy distintos pero igual de válidos a la hora de cumplir su función.

En 1979, John Ross Quinlan utiliza la teoría de la información desarrollada por C. Shannon en 1948 para elaborar un método que permitía construir árboles de decisión con muy buenas características, como buen balanceado y pequeño tamaño. Este algoritmo se conoce como ID3 [13], y construye el árbol de decisión de arriba a abajo y de forma directa.

Aunque este algoritmo se suele presentar de forma general en la que los datos de entrada y la clasificación pueden formar cualquier conjunto (no necesariamente numérico), en la formalización que daremos a continuación nos restringimos a árboles

de decisión que manipulan datos de un espacio vectorial real y las clasificaciones son, asimismo, numéricas. Además, y aunque tampoco es necesario, trabajaremos con árboles binarios (cada nodo interior tiene 2 hijos).

Algoritmo 3.6 (Árbol de decisión). *Dado un conjunto de entrenamiento* $S = \{(x, y) : x \in R^d, y \in R\}$, *un árbol de decisión divide el espacio recursivamente de modo que las muestras con las mismas etiquetas se agrupan juntas y se redirigen por ramas disjuntas del árbol, por lo que en etapas sucesivas el conjunto de datos que llega a un nodo es un suconjunto del conjunto de datos que pasó por su nodo padre.*

Si los datos que llegan a un nodo interior m se representan por Q. *Para cada posible división* $\theta = (j, t_m)$, *con* $1 \leq j \leq d$ *y* $t_m \in R$, *se separan los datos en los subconjuntos:*

$$Q_{izq}(\theta) = \{(x, y) \in Q : x_j \leq t_m\}$$

$$Q_{dcha}(\theta) = Q \setminus Q_{izq}(\theta)$$

Este proceso no determina un árbol único, sino que para cada posible elección de θ *obtendremos árboles distintos. Con el fin de obtener árboles pequeños (en número de nodos interiores) que sean capaces de clasificar eficientemente datos complejos, se suele hacer uso de lo que se denominan funciones de impureza, que miden, de alguna forma, cómo de homogéneos son los datos que llegan a un nodo respecto a sus clasificaciones y ayudarán a decidir qué predicado considerar.*

Si H *es una de tales medidas de impureza, se define la impureza derivada de* H *que proporciona una división concreta,* θ, *como:*

$$G(Q, \theta) = \frac{n_{izq}}{N_m} H(Q_{izq}(\theta)) + \frac{n_{dcha}}{N_m} H(Q_{dcha}(\theta))$$

donde $N_m = |Q|$; $n_{izq} = |Q_{izq}|$; $n_{dcha} = |Q_{dcha}|$

Así pues, se elige el predicado que minimiza esta impureza derivada:

$$\theta^* = argmin_\theta G(Q, \theta)$$

Para cada uno de los dos conjuntos generados se crea un nodo al que se le pasará el contenido correspondiente, y se colocan como hijos de m en la construcción del árbol de decisión.

Comenzando por un nodo raíz que se asocia al conjunto S *completo, se repite de forma recursiva este proceso hasta que se alcanza algún requisito previamente establecido (por ejemplo, una máxima profundidad,* $G(Q, \theta) < umbral$, *o* $N_m = 1$*).*

La Figura 3.7 muestra un ejemplo de árbol de decisión de profundidad 2 y una entrada de datos cualquiera.

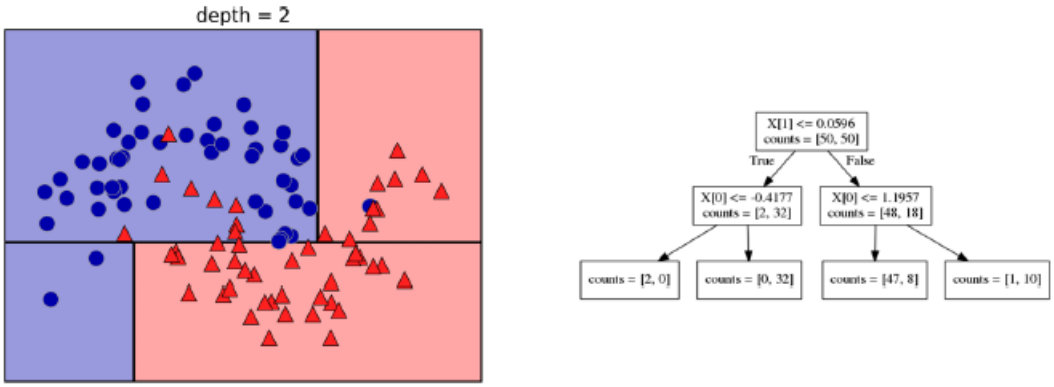


Figura 3.7: Árbol de profundidad 2. Imagen extraída de [9]

Algunos ejemplos de H son :

- Gini: $H(X_m) = \sum_k p_{mk}(1 - p_{mk})$.
- Entropía cruzada: $H(X_m) = -\sum_k p_{mk} log(p_{mk})$.
- $H(X_m) = 1 - max(p_{mk})$, donde X_m son los datos de entrenamiento en el nodo m y $p_{mk} = \frac{1}{N_m} \sum_{x_i \in \mathbb{R}_m} I(y_i = k)$.

Hay varias técnicas para reducir el riesgo de sobreajuste a los datos en los árboles de decisión, como acotar la profundidad máxima que deben alcanzar. Pero también se han creado otros algoritmos basados en árboles que persiguen el mismo objetivo a la vez que mejorar su capacidad predictiva. Los conglomerados de árboles, o *bosques aleatorios*, son una de las soluciones más extendidas debido a su facilidad de implementación y buenos resultados.

Algoritmo 3.7 (Bosques aleatorios). *Para obtener un bosque aleatorio, primero hay que decidir de cuántos árboles va a componerse. Si disponemos de n árboles, entonces se crearán n nuevos conjuntos a partir del conjunto de entrenamiento a base de extraer elementos con reemplazo de dicho conjunto. Cada nuevo conjunto debe tener el tamaño del conjunto de entrenamiento. A continuación, se entrena un árbol con cada conjunto, obteniendo así n árboles diferentes.*

Para realizar una predicción con nuestro bosque aleatorio, lo que haremos será evaluar todos los árboles sobre el dato de entrada, y por voto de la mayoría decidir la etiqueta que hay que asignarle.

La Figura 3.8 muestra un ejemplo de bosque aleatorio, donde se han tomado cinco árboles distintos entrenados sobre subconjuntos del mismo conjunto de entrenamiento.

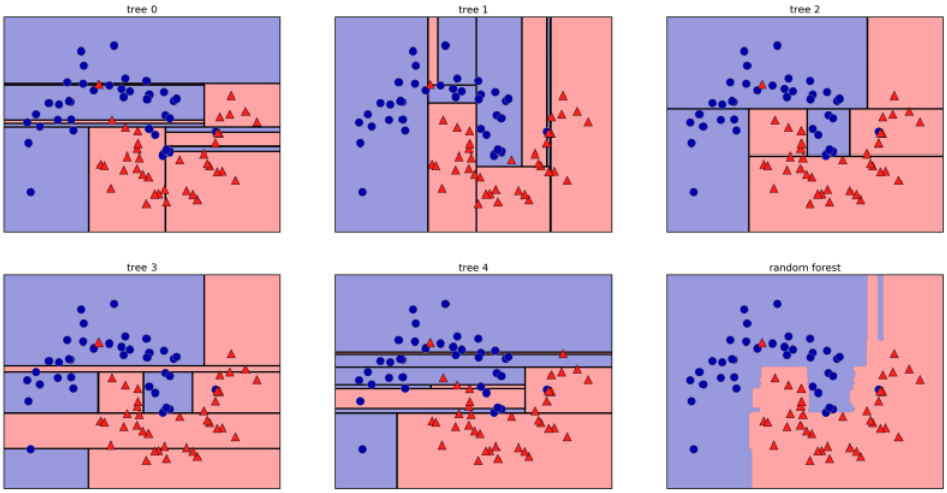


Figura 3.8: Ejemplo de Bosque aleatorio. Imagen extraída de [9]

3.3.4 Boosting

El *Boosting* es un paradigma algorítmico que intenta tratar dos problemas importantes: la complejidad computacional de aprendizaje, y el equilibrio entre Bias y Complejidad, tal y como se definió en 3.8

Un algoritmo de boosting amplifica la precisión de algoritmos de aprendizaje débiles. Intuitivamente, estos son algoritmos fáciles de computar, pero que apenas mejoran una predicción aleatoria. Boosting proporciona herramientas para usar estos algoritmos débiles y convertirlos en algoritmos mejores.

Definición 3.11. *Sea $\mathcal{X} = \mathbb{R}^d$. Al conjunto:*

$$\mathcal{H}_{DS} = \{x \mapsto sign(\theta - x_i) \cdot b : \theta \in \mathbb{R}, i \in \{1, ..., d\}, b \in \{-1, 1\}\}$$

lo llamaremos la clase de los nodos de decisión sobre $\mathbb{R}^d$.

Asumiremos que $b = 1$ para estudiar los predictores de la forma $sign(\theta - x_i)$.

Definición 3.12. *Un algoritmo de aprendizaje A se dice un γ-aprendiz-débil para una clase $\mathcal{H}$ si existe una función $m_{\mathcal{H}} : (0, 1) \mapsto \mathbb{N}$ tal que*

- *para cada* $\delta \in (0,1)$,
- *para cada distribución* $\mathcal{D}$ *sobre* $\mathcal{X}$,
- *y para cada función de etiquetado* $f : \mathcal{X} \mapsto \{\pm 1\}$

si existe un predictor que cumpla el supuesto de realización (3.3) respecto a $\mathcal{H}, \mathcal{D}$ *y* f, *entonces cuando se ejecuta el algoritmo sobre* $m \geq m_{\mathcal{H}}(\delta)$ *ejemplos independientes e idénticamente distribuidos (i.i.d.) generados por* $\mathcal{D}$ *y etiquetados por* f, *el algoritmo devuelve un predictor* h *tal que con probabilidad al menos* $1-\delta$ *cumple que* $L_{(\mathcal{D},f)}(h) \leq 1/2 - \gamma$.

Algoritmo 3.8 (AdaBoost). *Dado un conjunto de entrenamiento* $S = \{x_i, y_i\}_{i=1}^m$ *tal que para cada* $i, y_i = f(x_i)$ *con* f *una función de etiquetado, y un algoritmo de aprendizaje débil* AD, *se define el algoritmo AdaBoost como:*

Entrada: *Conjunto de entrenamiento* $S = \{x_i, y_i\}_{i=1}^m$
Aprendiz débil AD
Número de rondas T
Salida: *El predictor* $h_s(x) = sign\left(\sum_{t=1}^T w_t h_t(x)\right)$
Inicializar: $D^{(1)} = \left(\frac{1}{m}, ..., \frac{1}{m}\right)$
para $t = 1, 2..., T$ ***hacer***
Determinar un aprendiz débil $h_t = AD(D^{(t)}, S)$
Calcular $\epsilon_t = \sum_{i=1}^m D_i^{(t)} \mathbb{1}_{[y_i \neq h_t(x_i)]}$
$w_t = \frac{1}{2} log\left(\frac{1}{\epsilon_t} - 1\right)$
$D_i^{(t+1)} = \frac{D_i^{(t)} e^{(-w_t y_i h_t(x_i))}}{\sum_{j=1}^m D_j^{(t)} e^{(-w_t y_j h_t(x_j))}} \forall i = 1, ..., m$
fin para

Teorema 3.2. *Sea S un conjunto de entrenamiento. Asumimos que en cada iteración de AdaBoost, el algoritmo de aprendizaje débil devuelve un predictor para el cual* $\epsilon_t \leq 1/2 - \gamma$. *Entonces se tiene que el error de entrenamiento para el predictor que devuelve AdaBoost es como mucho*

$$L_S(h_s) = \frac{1}{m} \sum_{i=1}^m \mathbb{1}_{[h_s(x_i) \neq y_i]} \leq e^{(-2\gamma^2 T)}$$

Demostración. Una prueba de este resultado se encuentra en [8], páginas 135-137.

Observación 3.1. El teorema nos asegura que el error sobre el conjunto de entrenamiento del predictor tiende a 0 cuando T crece.

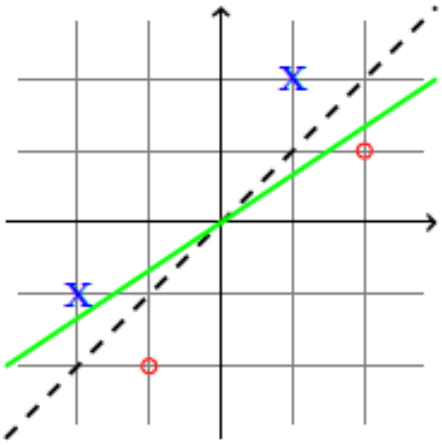


Figura 3.9: Intuitivamente, sabemos que la línea negra es mejor predictor que la línea verde. Imagen extraída de [8]

3.3.5 Máquinas de soporte vectorial

El objetivo de esta sección es hallar predictores lineales en espacios de dimensiones más grandes, y reducir el error derivado del conjunto de entrenamiento, así como la complejidad computacional.

La idea principal es encontrar los predictores que dejen mayor margen de error a las muestras del conjunto de entrenamiento. Por ejemplo, en la figura 3.9 vemos que, aunque tanto la línea negra como la línea verde separan bien las cuatro muestras para clasificarlas, intuitivamente pensaremos que es mejor la línea negra.

Formalmente, lo que se busca es resolver el siguiente problema, llamado Hard-SVM:

$$argmax_{(w,b):||w||=1} min_{i\in[m]} |\langle w, x_i\rangle + b|$$

tal que $\forall i, y_i \left(\langle w, x_i\rangle + b\right) > 0$.

Proposición 3.3. Si existe una solución del problema anterior, entonces este es equivalente a:

$$argmax_{(w,b):||w||=1} min_{i\in[m]} y_i \left(\langle w, x_i\rangle + b\right)$$

Llamaremos γ_i a $min_{i\in[m]} y_i \left(\langle w, x_i\rangle + b\right)$.

Una formulación equivalente como un problema de optimización cuadrática es la siguiente:

$$argmin_{(w,b)}||w||^2 : \forall i, y_i \left(\langle w, x_i\rangle + b\right) \geq 1$$

Nota 3.2. A veces será más conveniente trabajar con semiespacios homogéneos. Como vimos al principio de la sección de predictores lineales, podemos convertir el problema anterior en uno homogéneo equivalente:

$$min_w||w||^2 : \forall i, y_i\langle w, x_i\rangle \geq 1$$

Definición 3.13 (Soft-SVM). *A veces no se cumplirá la condición* $y_i \left(\langle w, x_i\rangle + b\right) \geq 1$ *para algún* i*, por lo que introduciremos una versión, llamada Soft-SVM, que suaviza esta restricción introduciendo nuevos parámetros:*

$$min_{w,b,\zeta}\left(\lambda||w||^2 + \frac{1}{m}\sum_{i=1}^{m}\zeta_i\right)$$

tal que $\forall i, y_i \left(\langle w, x_i\rangle + b\right) \geq 1 - \zeta_i$ *y* $\zeta_i \geq 0$.

El parámetro λ *servirá para controlar el equilibrio entre la norma de* w *y lo que se relajan las restricciones.*

En ocasiones no podremos clasificar los datos usando predictores lineales, pero existirán inmersiones que trasladen los datos a espacios de dimensión superior, y en estos sí podremos clasificarlos sin problemas. El coste computacional aumenta a cambio de obtener mayor precisión. El espacio que se obtiene de la inmersión puede ser de dimensión infinita. Un ejemplo visual de lo que se pretende conseguir se encuentra en la figura 3.10.

Definición 3.14 (El truco del núcleo). *Dada una inmersión* ψ *de un espacio* $\mathcal{X}$ *en un espacio de Hilbert, definimos la función núcleo* $K(x, x') = \langle\psi(x), \psi(x')\rangle$.

Algoritmo 3.9 (Máquinas de soporte vectorial lineales con núcleos). *El objetivo será resolver el problema*

$$min_w\left(\frac{\lambda}{2}||w||^2 + \frac{1}{m}\sum_{i=1}^{m} max\{0, 1 - y\langle w, \psi\left(x_i\right)\rangle\}\right)$$

.

Entrada: *Conjunto de entrenamiento* $S = \{x_i, y_i\}_{i=1}^m$
Número de rondas T

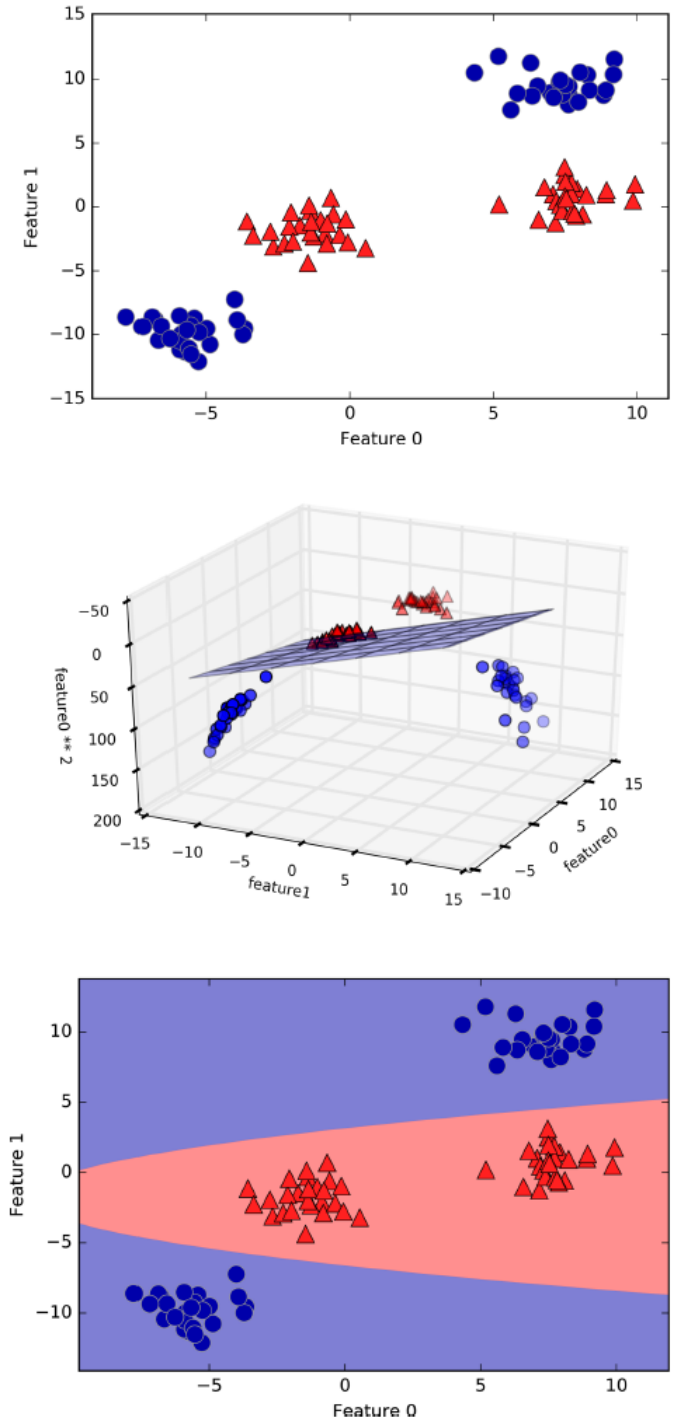


Figura 3.10: Inmersión de los datos en un espacio de dimensión superior para encontrar un mejor predictor usando SVM. Imagenes extraídas de [9]

Salida: $\bar{w} = \sum_{j=1}^{m} \bar{\alpha}_j \psi\left(x_j\right)$ *con* $\bar{\alpha} = \frac{1}{T}\sum_{t=1}^{T} \alpha^{(t)}$
Inicializar: $\beta^{(1)} = 0$
para $t = 1, 2..., T$ ***hacer***
Determinar $\alpha^{(t)} = \frac{1}{\lambda t}\beta^{(t)}$
Elegir i *uniformemente en* $\{1, ..., m\}$
Para todo $j \neq i$ *establecer* $\beta_j^{(t+1)} = \beta_j^{(t)}$
si $y_i \sum_{j=1}^{m} \alpha_j^{(t)} K\left(x_j, x_i\right) < 1$ ***entonces***
Establecer $\beta_i^{(t+1)} = \beta_i^{(t)} + y_i$
si no
Establecer $\beta_i^{(t+1)} = \beta_i^{(t)}$
fin si
fin para

Nota 3.3. Para las aproximaciones experimentales que se muestran en el capítulo siguiente usaremos el algoritmo *SVC* de la librería *sklearn*, que incluye un parámetro C en lugar del parámetro λ que encontramos en el algoritmo con las siguientes diferencias: multiplica el argumento a minimizar por $\frac{1}{\lambda}$, y luego realiza el cambio $C = \frac{1}{\lambda \cdot m}$. Las soluciones que se obtienen de la minimización siguen siendo las mismas, pero ahora la constante por la que multiplica es C.

3.3.6 Multiclases

A veces existirán etiquetas que no sean binarias, es decir, existirá un conjunto finito de posibles etiquetas con más de dos elementos. Para encontrar un predictor que permita realizar la clasificación deseada dispondremos de varios algoritmos:

A partir de este momento tomaremos $S = \{x_i, y_i\}_{i=1}^{m}$ un conjunto de entrenamiento tal que $y_i \in \mathcal{Y} = \{1, ..., k\} \forall i$ sin pérdida de generalidad.

Algoritmo 3.10 (Uno contra todos). *La idea de este algoritmo será encontrar* k *predictores lineales, uno por cada categoría, y realizar la clasificación de* x *aplicando* $\underset{i \in [k]}{\operatorname{argmax}}\, h_i(x)$. *En caso de que haya empate, entonces se tomará uno fijado previamente (por ejemplo, el que tenga menor* i*).*

Entrada: Conjunto S
Algoritmo de clasificación binaria A
para todo $i \in \mathcal{Y}$ ***hacer***
Establecer $S_i = (x_1, (-1)^{\mathbb{1}_{[y_1 \neq i]}}), ..., (x_m, (-1)^{\mathbb{1}_{[y_m \neq i]}})$

Establecer $h_i = A(S_i)$
fin para
Salida: $h(x) \in \underset{i \in \mathcal{Y}}{argmax}\ h_i(x)$

Algoritmo 3.11 (Todos los pares). *La idea de este algoritmo será comparar parejas de clases. Formalmente, tomaremos para cada* $1 \leq i < j \leq k$ *una secuencia* $S_{i,j}$ *que contendrá a todos los elementos de* S *tales que su etiqueta es* i *o* j*. Cada elemento de* $S_{i,j}$ *lo etiquetaremos con +1 si su etiqueta es* i *y con -1 si su etiqueta es* j*. Luego conseguiremos un predictor binario* $h_{i,j}$ *para cada* $S_{i,j}$*, y construiremos el predictor encontrando la clase que tiene mayor cantidad de "victorias".*

Entrada: Conjunto S
Algoritmo de clasificación binaria A
para todo $i, j \in \mathcal{Y}$ *tal que* $i < j$ ***hacer***
Inicializar $S_{i,j}$ *como un conjunto vacío.*
para $t = 1, ..., m$ ***hacer***
Si $y_t = i$*, añadir* $(x_t, 1)$ *a* $S_{i,j}$*.*
Si $y_t = j$*, añadir* $(x_t, -1)$ *a* $S_{i,j}$*.*
fin para
Establecer $h_{i,j} = A(S_{i,j})$
fin para
Salida: $h(x) \in \underset{i \in \mathcal{Y}}{argmax} \sum_{j \in \mathcal{Y}} sign(j - i)\, h_{i,j}(x)$

Nota 3.4. Todos los algoritmos que usaremos, cuando sean aplicadas a clasificaciones multiclase, utilizarán el algoritmo de **Todos los pares**.

3.3.7 Redes neuronales

En este apartado trabajaremos sobre modelos que imitan el comportamiento de las redes neuronales reales. Utilizaremos grafos dirigidos que no contengan ciclos, y cuyos vértices podremos separar en capas según los vértices a los que estén conectados.

Sea $G = (V, E)$ un grafo dirigido sin ciclos y $w : E \to \mathbb{R}$ una función peso definida sobre las aristas. Cada vértice se corresponde con una neurona de nuestra red neuronal. Podemos expresar $V = \bigcup_{t=0}^{T} V_t$ como unión disjunta, de modo que cada arista de E conecta un vértice de V_{t-1} con un vértice de V_t. Las neuronas de V_0 tienen como salida los x_i a excepción de la última, que tiene salida constante 1.

Un esquema de una red neuronal con una capa oculta es el de la figura 3.11. En ella vemos que la capa de entrada tiene tres neuronas, la oculta tiene tres, y la de salida tiene dos.

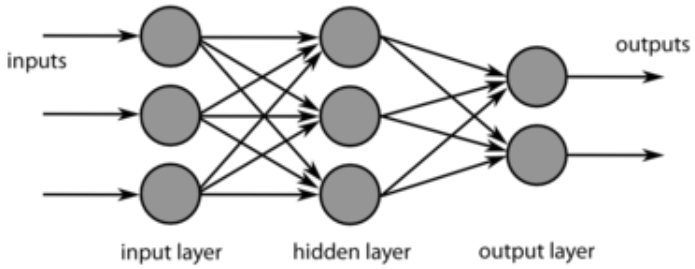


Figura 3.11: Esquema de Red Neuronal. Recuperada de es.wikipedia.org

Sea $v_{t+1,j} \in V_{t+1}$ una neurona de la capa V_{t+1} y $o_{t,i}(x)$ la salida de la neurona $v_{t,i}$. Entonces el valor que recibe como entrada dicha neurona es:

$$\sum_{r:(v_{t,r},v_{t+1,j})\in E} w((v_{t,r}, v_{t+1,j}))o_{t,r}(x)$$

Para calcular el valor que se utiliza como salida se aplica a dicho valor la función σ, que normalmente es la función signo, una función umbral o la sigmoidea, que es buena aproximación de la umbral y además es derivable (algo fundamental para los procesos de ajuste que veremos después).

A la capa V_0 se le llama *capa de entrada*, a la capa V_T *capa de salida*, y al resto *capas ocultas*.

Para modificar la función peso utilizaremos el algoritmo *Backpropagation*, que es un algoritmo optimización de la familia de Gradiente Descendiente y hará que obtengamos un buen predictor respecto a los datos de entrada.

Algoritmo 3.12 (SGD para redes neuronales). *Dada una distribución $\mathcal{D}$ sobre $\mathcal{X} \times \mathcal{Y}$:*

Parámetros:
Número de iteraciones τ
Secuencia de longitudes de paso: $\eta_1, ..., \eta_\tau$
Parámetro de regularización $\lambda > 0$
Entrada:

Grafo etiquetado (V, E)
Función de activación diferenciable $\sigma : \mathbb{R} \to \mathbb{R}$
Inicializar $w^{(1)} \in \mathbb{R}^{|E|}$ *al azar de una distribución tal que esté cerca del 0.*
para *t=1,...,*τ ***hacer***
Tomar elemento $(x, y) \sim \mathcal{D}$
Calcular el gradiente $v_i = backpropagation(x,y,w,(V, E), \sigma)$
Actualizar $w^{(i+1)} = w^{(i)} - \eta_i \left(v_i + \lambda w^{(i)}\right)$
fin para
Salida: $\bar{w}$ *es el* $w^{(i)}$ *con mejor rendimiento en un conjunto de validación*

El algoritmo de backpropagation se define como:

Entrada: *Ejemplo* (x, y)*, vector de pesos* w*, grafo etiquetado* (V, E)*, función de activación* $\sigma : \mathbb{R} \to \mathbb{R}$
Inicializar:
Denotar las capas del grafo $V_0, ..., V_T$ *con* $V_t = v_{t,1}, ..., v_{t,k_t}$
Definir $W_{t,i,j}$ *como el peso de la arista* $(v_{t,j}, v_{t+1,i})$*. Si la arista no existe se define como 0.*
Hacia delante:
Establecer $o_0 = x$
para $t = 1, ...T$ ***hacer***
para $i = 1, ..., k_t$ ***hacer***
Establecer $a_{t,i} = \sum_{j=1}^{k_t - 1} W_{t-1,i,j} o_{t-1,j}$
Establecer $o_{t,i} = \sigma(a_{t,i})$
fin para
fin para
Hacia atrás:
Establecer $\delta_T = o_T - y$
para $t = T - 1, ..., 1$ ***hacer***
para $i = 1, ..., k_t$ ***hacer***
$\delta_{t,i} = \sum_{j=1}^{k_{t+1}} W_{t,j,i} \delta_{t+1,j} \sigma'(a_{t+1,j})$
fin para
fin para
Salida: *Para cada arista* $(v_{t-1,j}, v_{t,i}) \in E$ *devolver* $\delta_{t,i} \sigma'(a_{t,i}) o_{t-1,j}$

Además de las redes neuronales descritas, también llamadas Redes Neuronales Artificiales, o *ANN* por sus siglas en inglés, existen algunas variantes, como las redes neuronales convolucionales, o *CNN*, las cuales están especializadas en el reconoci-

miento de imágenes, y que, a pesar de ser de gran interés para la evolución actual de las redes neuronales y de su uso intensivo en tareas de alta complejidad hoy en día (llegando a evolucionar en un nuevo tipo de Machine Learning conocido como *Deep Learning*) no presentamos con más detalle al no haber sido utilizadas para este trabajo. En todo caso, el lector puede encontrar más información sobre ellas en [14].

4 Resultados Experimentales

En este capítulo se analizan los resultados obtenidos tras aplicar los algoritmos de aprendizaje anteriormente definidos sobre ciertos autómatas celulares. Las especificaciones se definen en la primera sección, la implementación se trata en la segunda sección, y en la tercera se analizan los datos en sí.

Todas las imágenes usadas en este capítulo han sido creadas usando el código desarrollado. Concretamente, las gráficas se han creado usando la librería *matplotlib* de Python.

4.1 Consideraciones previas

Una vez que se han introducido los autómatas celulares y los algoritmos de Machine Learning a utilizar, hay que tomar una serie de decisiones sobre los autómatas que se van a estudiar y los algoritmos que se van a utilizar. Estos parámetros pueden ser modificados en el código.

La metodología seguida en la etapa experimental que se ha realizado para este trabajo se puede resumir en los siguientes puntos clave:

4.1.1 Autómatas Celulares

- Los autómatas celulares serán unidimensionales y tendrán frontera circular.
- De las reglas que los definen, se estudiarán las 88 que no son equivalentes entre sí y que representan las clases de equivalencia que se indicaron en el capítulo correspondiente.

– Cada autómata tendrá 100 células, se ejecutarán 131 generaciones por autómata.

– Se generarán 1.000 experimentos para cada una de las 88 reglas, cada uno de ellos partirá de una configuración inicial generada aleatoriamente.

4.1.2 Machine Learning

– Los algoritmos de aprendizaje, que se usarán para el reconocimiento de patrones sobre los autómatas, utilizarán 750 de dichos autómatas para la etapa de entrenamiento (*conjunto de entrenamiento*), y los 250 restantes para comprobar la bondad de su aprendizaje (*conjunto de test*).

– Los algoritmos usados serán los de clasificación. Algunos algoritmos de Machine Learning requieren de algunos parámetros, como la cantidad de capas ocultas y neuronas por capa en las redes neuronales, o el parámetro α que mide la penalización de la norma de los coeficientes en la regresión de arista.

  Estos parámetros se eligen por un procedimiento heurístico. En el archivo *Main.py* del código asociado a cada sección de ML tendrá un apartado dedicado a los parámetros que requiere dicha sección con el fin de que los parámetros puedan modificarse para hacer pruebas. Debido a que no intentamos realizar comparativas con otras aplicaciones de ML al aprendizaje de patrones de autómatas celulares, y este trabajo solo supone una primera aproximación a este tipo de análisis, no se ha hecho un especial hincapié en la optimización de estos hiperparámetros ni se ha seguido un protocolo exhaustivo para su búsqueda (por ejemplo, manteniendo un tercer conjunto de muestras para una validación final).

– Para cada algoritmo que requiera parámetros, se tomará el mejor parámetro para cada autómata elegido en un rango finito. Esto significa que la red neuronal que hemos entrenado sobre una regla puede tener distintas capas ocultas o neuronas por capa que la entrenada sobre otra regla.

  El motivo ha sido el de querer encontrar el algoritmo de aprendizaje que mejor funciona para cada autómata dentro de un mismo tipo de algoritmo, y compararlo luego con el resto de algoritmos de aprendizaje.

– Se han intentado predecir dos rasgos de los distintos autómatas. Por un lado, se ha intentado predecir la **mayoría**, en el sentido de predecir si hay más células vivas que muertas. En ocasiones llamaremos a este rasgo **densidad**, en el sentido

de que se predice si hay más densidad de células vivas o muertas. Por otro lado, se ha intentado predecir la **evolución temporal** de una porción del autómata: el estado de la primera célula por un lado, y el estado de las tres primeras células de cada autómata por otro.

Los datos de entrada que recibe cada algoritmo son la configuración inicial del autómata y el rasgo a estudiar de la última generación. Por ejemplo, el autómata con regla 0 puede tener como datos de entrada una cadena formada por 0s y 1s generada aleatoriamente y la mayoría 0, ya que en la regla 0 siempre mueren todas las células. En el caso de estudiar las tres primeras células, reciben como datos de entrada la cadena formada por 0s y 1s generada aleatoriamente, y la cadena 000.

Por la forma en que se ha desarrollado el código, pueden crearse con facilidad nuevas funciones que analizasen otros aspectos de los autómatas celulares usando los mismos algoritmos de ML, de modo que el código generado sirva también como una herramienta de experimentación ampliable a otro tipo de cuestiones derivadas con la evolución y complejidad de los autómatas celulares considerados.

4.2 Implementación

El código implementado, como se dijo anteriormente, está escrito en *Python* y utiliza las librerías *sklearn* y *matplotlib*. Hay varios archivos, y la estructura de estos es la que sigue:

- Un archivo *Main*, que es el que llamará a las funciones de creación de ficheros de autómatas, de creación y entrenamiento de los algoritmos (ver 4.1), y de creación de las gráficas. En este archivo se definen parámetros necesarios, como la cantidad de autómatas celulares que se van a analizar, y se realizan tareas como la creación de archivos que recojan la precisión de distintos predictores. Aunque en la figura 4.1 solo se vean predictores lineales, este archivo incluye funciones de creación de todos los algoritmos.

- Un archivo, llamado *Automatas*, que contiene las funciones necesarias para crear autómatas celulares correspondientes a las 88 reglas, ficheros que almacenen dichos autómatas y las funciones que extraen el rasgo a estudiar en cada uno de los experimentos. En la Figura 4.2 se muestran dos funciones: la función *Autómata*,

que almacena en un fichero un autómata de una regla, tamaño y con un número de generaciones introducidos como parámetros, y la función *CreaciónFicheros*, que crea para cada una de las 88 reglas un número determinado de ficheros.

– Un archivo por cada algoritmo de aprendizaje, que contiene dos tipos de funciones: las de entrenamiento de dicho algoritmo (ver 4.3) y las de creación de los ficheros que almacenen la precisión de cada algoritmo para las distintas reglas (ver 4.4).

  En la Figura 4.3 podemos ver cómo se entrenan los árboles de decisión. Primero se obtienen los datos a estudiar de entre los autómatas generados previamente. Luego, se separan en conjunto de entrenamiento y de test. Se entrenan por último árboles según distintos parámetros y se almacenan únicamente los mejores resultados. Para los otros algoritmos de aprendizaje el código es similar.

  Las dos funciones mostradas en la Figura 4.4 son prácticamente idénticas, debido a que lo que hace este código es comprobar si existe la carpeta donde almacenar los resultados, si no es así la crea, y para cada una de las 88 reglas almacena los resultados de entrenar arboles de decisión o bosques. Para los otros algoritmos de aprendizaje el código es similar.

– Un archivo, llamado *Graficas*, que contiene las funciones necesarias para crear gráficas que representen la precisión con la que los algoritmos predicen las distintas reglas (ver 4.5). En el código mostrado podemos ver la función *ComparaAutomatas*, que lo que hace es crear gráficas para las reglas y los rasgos elegidos en las que comparar la precisión de los distintos algoritmos de aprendizaje.

4.3 Aprendizaje

En esta sección vamos a tomar algunos ejemplos de cada clase de Wolfram (ver 2.3) y vamos a ver cómo se comportan los distintos algoritmos sobre ellos al estudiar diferentes rasgos. Aunque solo se muestran los resultados de 8 reglas de las 88 analizadas, el comportamiento de las reglas mostradas es representativo dentro de las clases.

Las reglas que vamos a considerar para cada una de las clases son:

– Clase I: 0 (Figura 6.1) y 168 (Figura 6.6).

– Clase II: 14 (Figura 6.2) y 108 (Figura 6.5).

```
# Parametros de entrenamiento
# N es el numero de automatas que se van a analizar. Se hara un reparto entre los de entrenamiento y
N = 1000
# Puede ser 'porcion', 'densidad' o 'celula'
etiquetas = 'celula'
# Lista con valores de posibles alpha para la regresion de arista y la Lasso
alphas = [0.01, 0.1, 1, 10, 100, 1000]
# Lista con los valores de posibles c para la regresion logistica
cs = [0.001, 0.01, 0.1, 1, 10, 100]

# Comentar esta linea si no se desea entrenar los predictores lineales por minimos cuadrados
RegresionLineal.CreacionMinCuad(N, etiquetas)
# Comentar esta linea si no se desea entrenar los predictores lineales por regresion de arista
RegresionLineal.CreacionRegRidge(N, etiquetas, alphas)
# Comentar esta linea si no se desea entrenar los predictores lineales por Lasso
RegresionLineal.CreacionRegLasso(N, etiquetas, alphas)
# Comentar esta linea si no se desea entrenar los predictores lineales por regresion logistica
RegresionLineal.CreacionRegLog(N, etiquetas, cs)
```

Figura 4.1: Muestra del código del archivo *Main*.

– Clase III: 30 (Figura 6.3) y 126 (Figura 6.5).

– Clase IV: 54 (Figura 6.4) y 110 (Figura 6.5).

En el apéndice (6.1) podemos encontrar imágenes de ejecuciones particulares para el resto de reglas. Hemos tomado estas ocho reglas por representar adecuadamente el comportamiento de las clases de Wolfram, pero con el código disponible se pueden repetir con facilidad los mismos análisis realizados sobre el resto de reglas.

Además del código, en la versión digital se adjuntan las gráficas con las comparaciones de cada subsección para las 88 reglas, por si se quisiera profundizar en su análisis.

4.3.1 Aprendizaje global: predicción de la mayoría

Una vez que se ha diseñado la batería de programas que nos permiten automatizar los procesos experimentales de cálculo de dinámicas y entrenamiento de algoritmos, nuestro primer experimento será el de ver hasta qué punto, para las distintas reglas consideradas y con las ejecuciones aleatorias almacenadas, los diversos algoritmos de aprendizaje son capaces de predecir el valor de **la mayoría** tras 130 pasos evolutivos. El valor de 130 no tiene ninguna característica esencial, pero sí es lo suficientemente grande como para que se supere el horizonte de predicción que de forma natural se podría observar si los algoritmos de aprendizaje memorizaran la evolución punto a

```
def Automata(regla,dirFichero):
    # Creacion de las celulas iniciales aleatorias.
    universe = ''.join(random.choice('01') for i in range(numeroCelulas))
    # Interpretacion de la regla introducida
    numero = BinByte(regla)
    neighbours2newstate = {
        '000': numero[7],
        '001': numero[6],
        '010': numero[5],
        '011': numero[4],
        '100': numero[3],
        '101': numero[2],
        '110': numero[1],
        '111': numero[0],
    }
    f = open(dirFichero, 'w')
    # Para cada iteracion se escribe en el fichero una linea de celulas, que evolucionan desde
    for i in range(generaciones):
        f.write(universe + '\n')
        universe = universe[numeroCelulas-1] + universe + universe[0]
        universe = ''.join(neighbours2newstate[universe[i:i+3]] for i in range(numeroCelulas))
    f.close()


# Funcion para crear ficheros de las 88 reglas que no son equivalentes a ninguna otra.
def CreacionFicheros():
    for i in clasesAutomatas:
        regla = i
        if not os.path.isdir('c:/Automatas/automata' + str(i)):
            os.mkdir('c:/Automatas/automata' + str(i))
        for j in range(numeroFicheros):
            dirFichero = 'c:/Automatas/automata' + str(i) + '/ejemplo' + str(j) + '.txt'
            Automata(regla, dirFichero)
```

Figura 4.2: Muestra del código del archivo *Automatas*.

punto.

Las figuras 4.6a a 4.9b muestran la precisión obtenida por los algoritmos cuando han tratado de predecir si había más densidad de células vivas o muertas:

Como era de esperar, todos los algoritmos predicen con probabilidad 1 el **problema de la mayoría** para los autómatas de la clase I de Wolfram (en los ejemplos mostrados, representadas por las reglas 0 (Fig. 4.6a) y 168 (Fig. 4.6b)). Esto se debe a que los elementos de esta clase evolucionan a un estado estable y homogéneo, por lo que la mayoría será siempre la misma.

Por otra parte, aunque los autómatas 14 y 108 pertenecen a la clase II de Wolfram, los algoritmos de ML utilizados son capaces de predecir con probabilidad 1 la regla 108 (Fig. 4.7b), pero no son capaces de predecir la regla 14 con probabilidad mayor a

```
def Arbol(regla,N,etiquetas,prof):
    # Obtencion de los datos sobre los que se trabaja
    x,y = A.Datos(regla,N,etiquetas)
    # Separación de los datos entre datos de entrenamiento y de test. El parametro "random_state=42" u
    # para validar multiples ejecuciones del codigo. La proporcion del conjunto de entrenamiento sera
    x_train, x_test, y_train, y_test = train_test_split(x,y,random_state=42)
    # Variables auxiliares
    best_depth = 1
    best_acc = 0
    best_acc_train = 0
    # Entrenamiento de los arboles con distintas profundidades. Unicamente se almacena el que predice
    for i in range(1, prof+1):
        tree = DecisionTreeClassifier(max_depth=i, random_state=0)
        tree.fit(x_train, y_train)
        acc = tree.score(x_test, y_test)
        if acc > best_acc :
            best_acc = acc
            best_depth = i
            best_acc_train = tree.score(x_train,y_train)

    return best_depth, best_acc_train, best_acc
```

Figura 4.3: Muestra del código del archivo *ArbolDecision* (I).

0.6. Esto se debe a que, aunque en ambas reglas la mayoría se estabiliza, la regla 14 (Fig. 4.7a) depende de la configuración inicial y en la regla 108 siempre hay más células muertas que vivas (ver 6.5). Aunque esto tiene un sentido lógico (de los grupos de más de 3 células vivas adyacentes sólo sobreviven las de los extremos, por lo que habrá muchas más vivas que muertas), esto es algo que han aprendido los algoritmos por su cuenta, y es donde reside su potencial: encuentran patrones que quizás las personas no detectan.

Por otra parte, también extraemos una primera conclusión, y es que el comportamiento de los diversos algoritmos no va a ser uniforme dentro de cada clase de complejidad de autómatas, por lo que haría falta hilar más fino para poder clasificar la complejidad del autómata por medio de la dificultad que encuentran los algoritmos de aprendizaje para su predicción.

Como era de esperar, la precisión de los algoritmos para las reglas 30 (Fig. 4.8a) y 126 (Fig. 4.8b) está en torno al 0.5. Esto se debe a que ambos autómatas pertenecen a la clase III, y por tanto siguen un comportamiento caótico, pero aún así algunos algoritmos están más cerca de predecir la mayoría con probabilidad 0.6 que con 0.5, lo que significa que ya son mejores que el azar y están aprendiendo patrones parciales del comportamiento caótico de los autómatas.

Finalmente, para las regla 54 (Fig. 4.9a) y 110 (Fig. 4.9b) se obtiene una sorpresa. Aunque los resultados para la regla 54 son los esperados de un autómata complejo (los algoritmos aprenden un poco, pues apenas superan una precisión de 0.6), para

```
def CreacionArboles(N,etiquetas,prof):
    if not os.path.isdir('c:/Automatas/Resultados'):
        os.mkdir('c:/Automatas/Resultados')
    dirFichero = 'c:/Automatas/Resultados/arboles_' + etiquetas + '.txt'
    f = open(dirFichero,'w')
    for regla in A.clasesAutomatas:
        depth, acc_train, acc_test = Arbol(regla, N, etiquetas, prof)
        f.write(str(regla) + " " + str(depth) + " " + str(acc_train) + " " + str(acc_test) + "\n")
    f.close()

def CreacionBosque(N,etiquetas,nArboles,prof):
    if not os.path.isdir('c:/Automatas/Resultados'):
        os.mkdir('c:/Automatas/Resultados')
    dirFichero = 'c:/Automatas/Resultados/bosque_' + etiquetas + '.txt'
    f = open(dirFichero, 'w')
    for regla in A.clasesAutomatas:
        nEstimators, acc_train, acc_test = Bosque(regla, N, etiquetas, nArboles, prof)
        f.write(str(regla) + " " + str(nEstimators) + " " + str(acc_train) + " " + str(acc_test) + "\n")
    f.close()
```

Figura 4.4: Muestra del código del archivo *ArbolDecision* (II).

la regla 110 casi todos los algoritmos predicen la mayoría con probabilidad casi 1, a pesar de pertenecer a la clase IV, y por tanto tener un comportamiento complejo. Esto significa que los algoritmos están aprendiendo algún patrón subyacente que no se ve a simple vista.

Es aquí donde destaca la utilidad de la aproximación realizada ya que, a pesar de ser un comportamiento complejo (no caótico), los algoritmos de ML proporcionan aproximaciones suficientemente buenas a la aproximación de la complejidad. Sin embargo, la utilidad de este aprendizaje podría estar limitado por un hecho muy simple, y es que si la mayoría del autómata varía poco durante su evolución (hay que tener en cuenta que se basa en una medida agregada), entonces no hay extracción de patrones, sino aproximación de un valor prácticamente constante, por lo que el error cometido será bajo.

4.3.2 Aprendizaje local: predicción de una célula

Abordamos ahora un problema diferente, que es el de intentar predecir el comportamiento de una sección local de la evolución (a diferencia del experimento anterior, en el que se intentaba predecir una característica global del sistema). En concreto, en este segundo experimento nos planteamos el estudio de hasta qué punto los algoritmos de aprendizaje son capaces de predecir el valor de una de las células del sistema (una en concreto prefijado). Aunque el comportamiento de los autómatas es determinista, y las reglas de evolución de los autómatas considerados son locales (solo dependen de

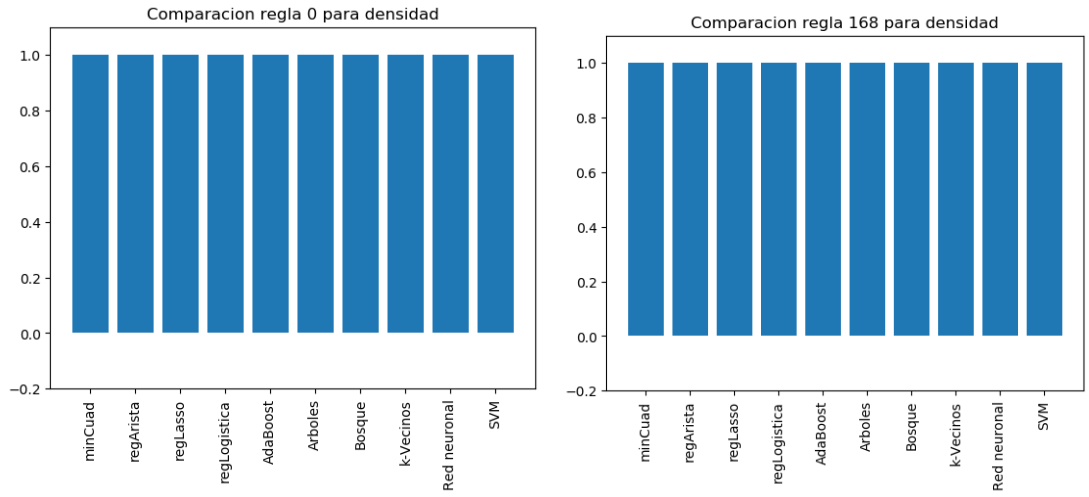
```
def ComparaAutomatas(regla, etiquetas):
    # Se crea una lista con todos los ficheros donde estan almacenados los datos de los algoritmos de l
    # aprendido respecto al rasgo dado
    ficheros = ['c:/Automatas/Resultados/minCuad_' + etiquetas + '.txt',
                'c:/Automatas/Resultados/regArista_' + etiquetas + '.txt',
                'c:/Automatas/Resultados/regLasso_' + etiquetas + '.txt',
                'c:/Automatas/Resultados/regLogistica_' + etiquetas + '.txt',
                'c:/Automatas/Resultados/AdaBoost_' + etiquetas + '.txt',
                'c:/Automatas/Resultados/arboles_' + etiquetas + '.txt',
                'c:/Automatas/Resultados/bosque_' + etiquetas + '.txt',
                'c:/Automatas/Resultados/k_Vecinos_' + etiquetas + '.txt',
                'c:/Automatas/Resultados/red_neuronal_' + etiquetas + '.txt',
                'c:/Automatas/Resultados/SVC_' + etiquetas + '.txt']
    r = str(regla)
    res = []
    # Se crea la grafica
    plt.figure(r)
    # Por cada algoritmo, se busca la precision en el automata dado
    for fic in ficheros:
        f = open(fic, 'r')
        lineas = [line.rstrip('\n') for line in f]
        for linea in lineas:
            linea = linea.split()
            if linea[0] == r:
                res.append(float(linea[len(linea)-1]))
    # Se dibujan las barras que representan a cada algoritmo
    plt.bar([i for i in range(len(ficheros))], res)
    # Se escribe el titulo
    plt.title('Comparacion regla ' + r + ' para ' + etiquetas)
    # Se etiquetan las barras con los algoritmos correspondientes
    indice = np.arange(len(ficheros))
    datos = tuple(['minCuad', 'regArista', 'regLasso', 'regLogistica', 'AdaBoost', 'Arboles', 'Bosque'
                   'Red neuronal', 'SVM'])
    plt.xticks(indice, datos, rotation='vertical')
    # Se ajusta el limite del eje y
    plt.ylim(-0.2, 1.1)
    # Si no existe la carpeta, se crea. Luego, se guarda la imagen correspondiente.
    if not os.path.isdir('c:/Automatas/Resultados/Graficas'):
        os.mkdir('c:/Automatas/Resultados/Graficas')
    plt.savefig('c:/Automatas/Resultados/Graficas/' + etiquetas + '_regla_' + r + '.png', bbox_inches=
    # Se elimina la figura para evitar problemas
    plt.close(r)
```

Figura 4.5: Muestra del código del archivo *Graficas*.

los vecinos), al haber dejado pasar muchos pasos de evolución el valor de cualquiera de las células realmente depende de un radio mucho de mayor de valores de células adyacentes, por lo que el algoritmo no puede simplemente memorizar o aprender una regla local para ser capaz de predecir con ciertas garantías el valor de una célula concreta conociendo únicamente la configuración inicial y el número de pasos.

Como estamos tratando el problema de predicción para una célula que puede tomar dos valores, cualquier predicción que supere con un cierto margen el valor de 0.5 estará mostrando un comportamiento de aprendizaje predictivo de patrones.

Las figuras (Figura 4.10a a 4.13b) muestran la precisión obtenida por los algoritmos



(a) Aprendizaje de la regla 0.

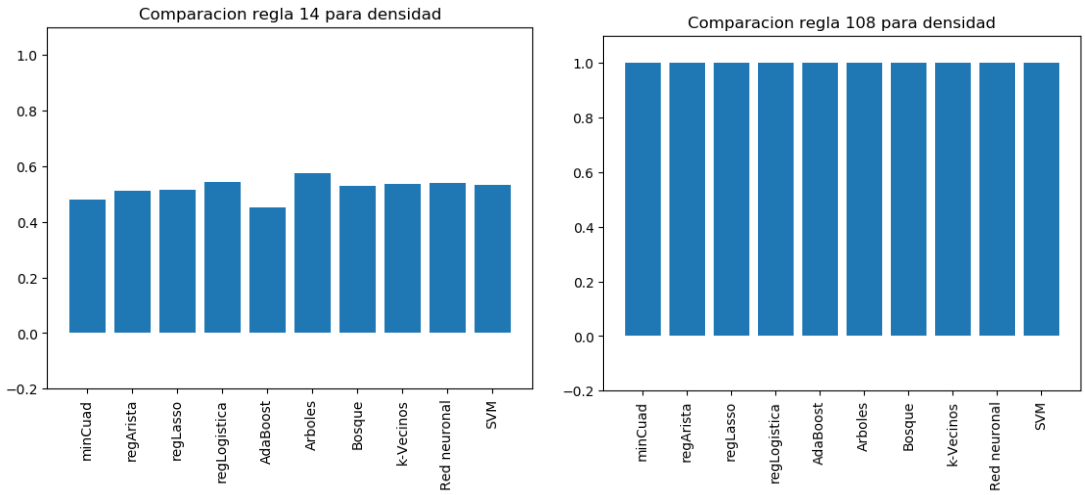
(b) Aprendizaje de la regla 168.

Figura 4.6

cuando han tratado de predecir la primera célula de cada autómata:

Como es normal, los algoritmos de aprendizaje utilizados predicen con probabilidad 1 la primera célula del autómata para las reglas 0 (Fig. 4.10a) y 168 (Fig. 4.10b) ya que, como hemos comentado, pertenecen a la clase I de Wolfram y por tanto llegan a estados homogéneos.

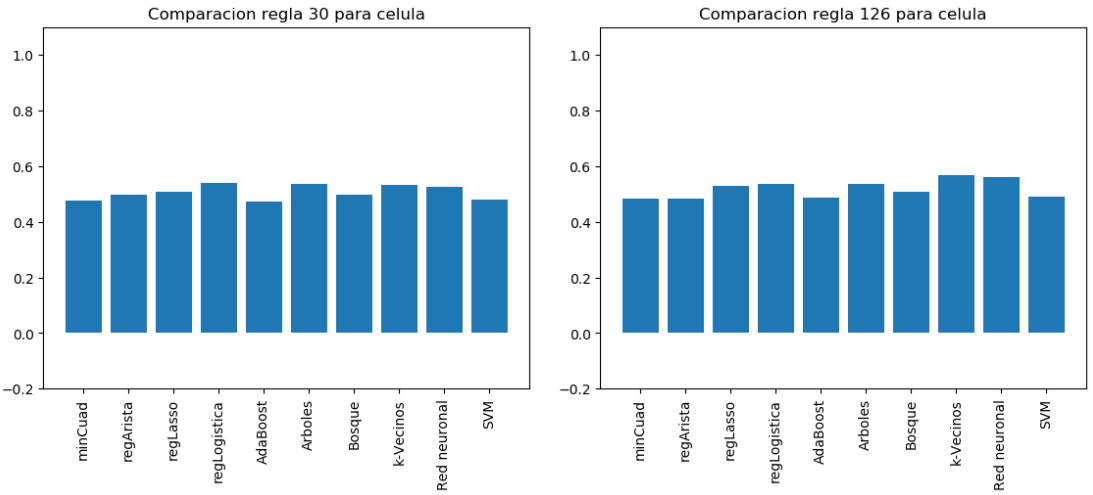
Considerando los autómatas de la clase II de Wolfram, tenemos que para la regla 14 (Fig. 4.11a) los algoritmos apenas mejoran el azar. Sin embargo, para la regla 108 (Fig. 4.11b) la predicción de la primera célula tiene una precisión de más de 0.8, la cual en sí misma es bastante buena, pero cuando observamos el aspecto (ver 6.5) del autómata, resulta sorprendente que no sean capaces de predecir con precisión casi 1 la primera célula de un autómata que se estabiliza siempre.



(a) Aprendizaje de la regla 14.

(b) Aprendizaje de la regla 108.

Figura 4.7

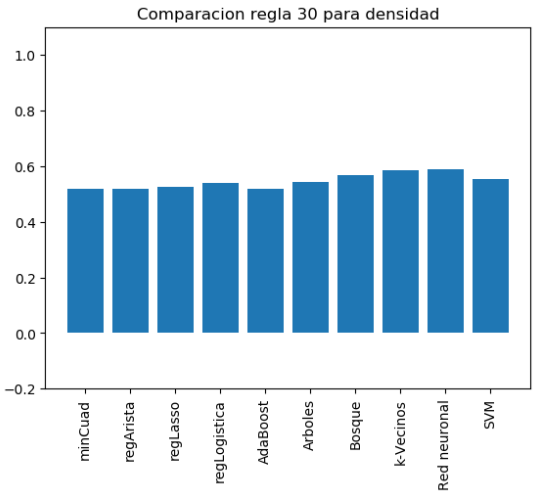


(a) Aprendizaje de la regla 30.

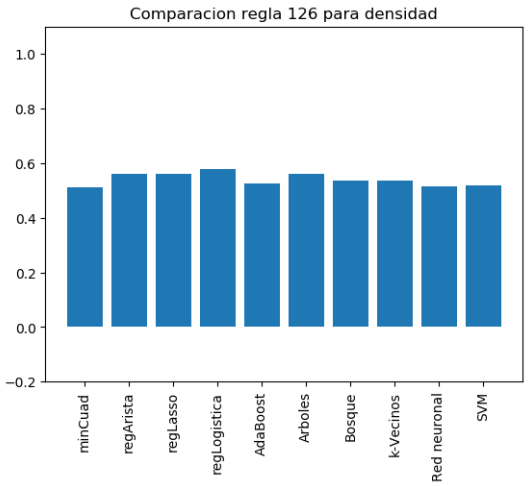
(b) Aprendizaje de la regla 126.

Figura 4.12

De nuevo, la precisión de los algoritmos sobre las reglas 30 (Fig. 4.12a) y 126 (Fig. 4.12b) está en torno a 0.5. es decir, similar al azar. Era lo esperado por ser caóticos (pertenecen a la clase III de Wolfram).

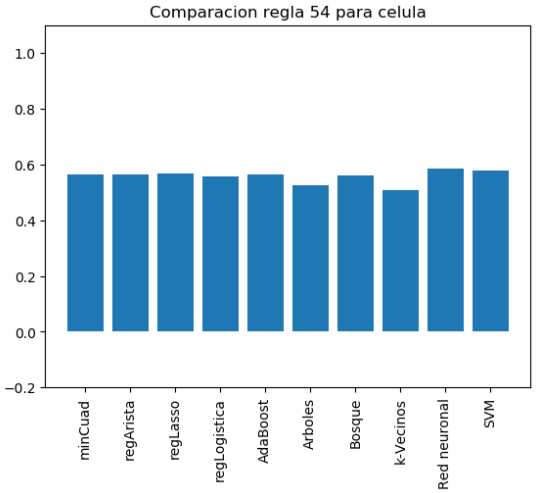


(a) Aprendizaje de la regla 30.

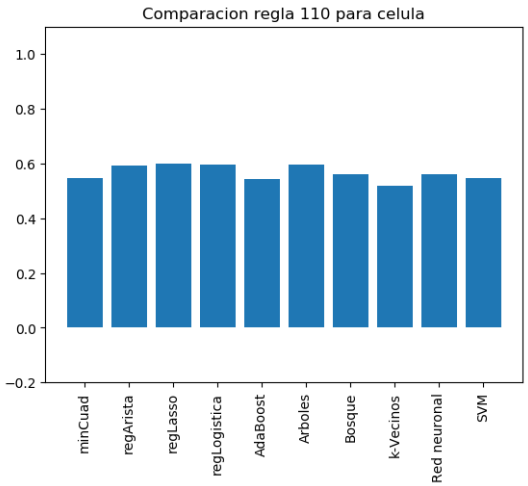


(b) Aprendizaje de la regla 126.

Figura 4.8



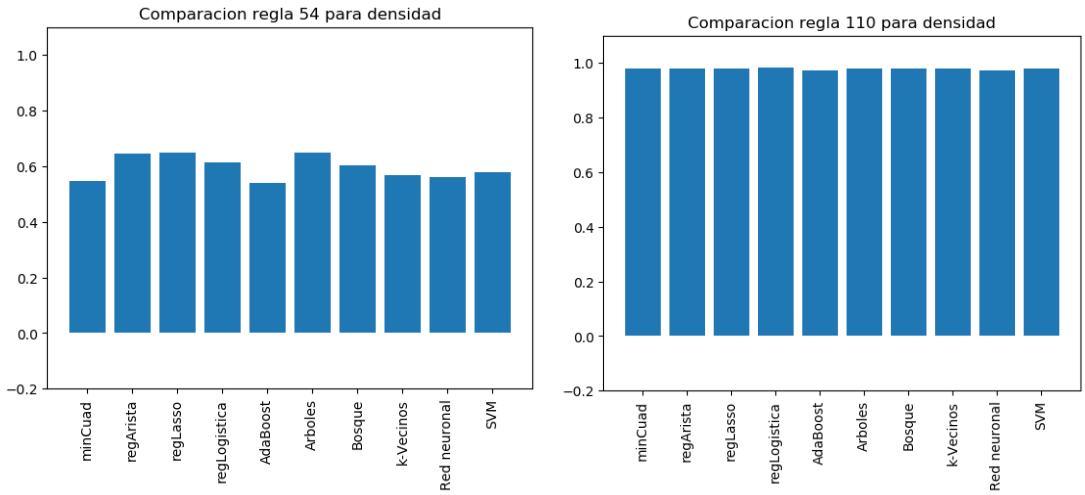
(a) Aprendizaje de la regla 54.



(b) Aprendizaje de la regla 54.

Figura 4.13

Respecto a las reglas 54 (Fig. 4.13a) y 110 (Fig. 4.13b), tenemos que las precisiones obtenidas son similares a las de los autómatas de la clase III. Esto concuerda con lo que sabemos de los autómatas complejos: estudiando individualmente cada célula no se localizan patrones, pero estudiando determinados rasgos más generales pueden surgir



(a) Aprendizaje de la regla 54.

(b) Aprendizaje de la regla 110.

Figura 4.9

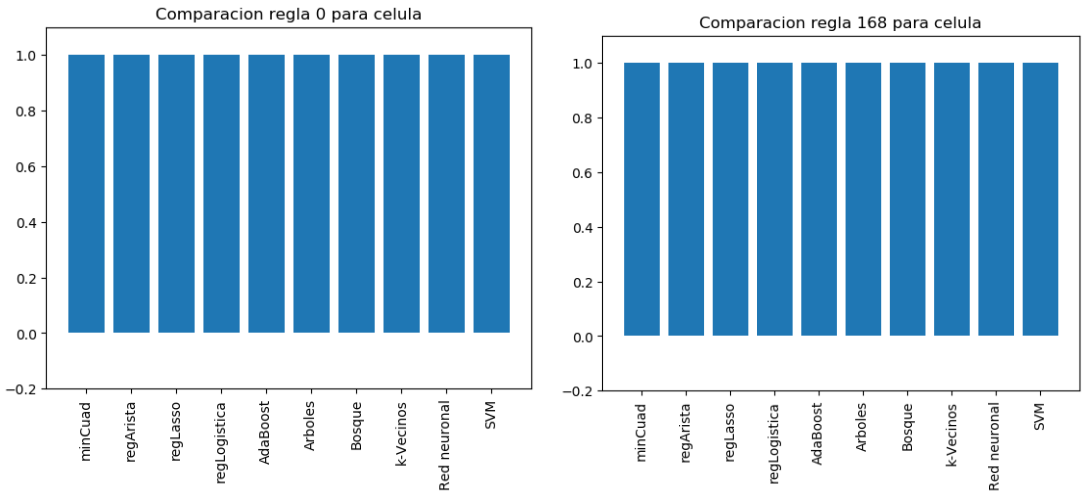
propiedades interesantes.

4.3.3 Aprendizaje local: predicción de tres células

Vamos a poner condiciones un poco más fuertes para los algoritmos de aprendizaje y estudiar cómo se comportan con un problema de aprendizaje local pero que involucra a un segmento, y no a una sola célula. En este caso, analizaremos la capacidad de predicción de 3 células consecutivas del autómata tras la evolución de 130 pasos. En este caso, convirtiendo el problema en un problema de predicción binario (si es capaz de predecir correctamente las 3 células o no) vemos que una decisión al azar conllevaría una probabilidad de acierto de 1/8 (0.125), por lo que cualquier valor que supere con cierto margen esa cantidad mostraría un aprendizaje significativo.

Las figuras (Figura 4.14a a 4.17b) muestran la precisión obtenida por los algoritmos cuando han tratado de predecir las tres primeras células de cada autómata:

De nuevo, como se podía intuir, todos los algoritmos predicen con probabilidad 1 las tres primeras células para los autómatas de la clase I de Wolfram: las reglas 0 (Fig. 4.14a) y 168 (Fig. 4.14b). La razón es la misma que antes: al evolucionar siempre al mismo estado homogéneo, los algoritmos aprenden que siempre va a dar el mismo resultado, sin importar la entrada.



(a) Aprendizaje de la regla 0.

(b) Aprendizaje de la regla 168.

Figura 4.10

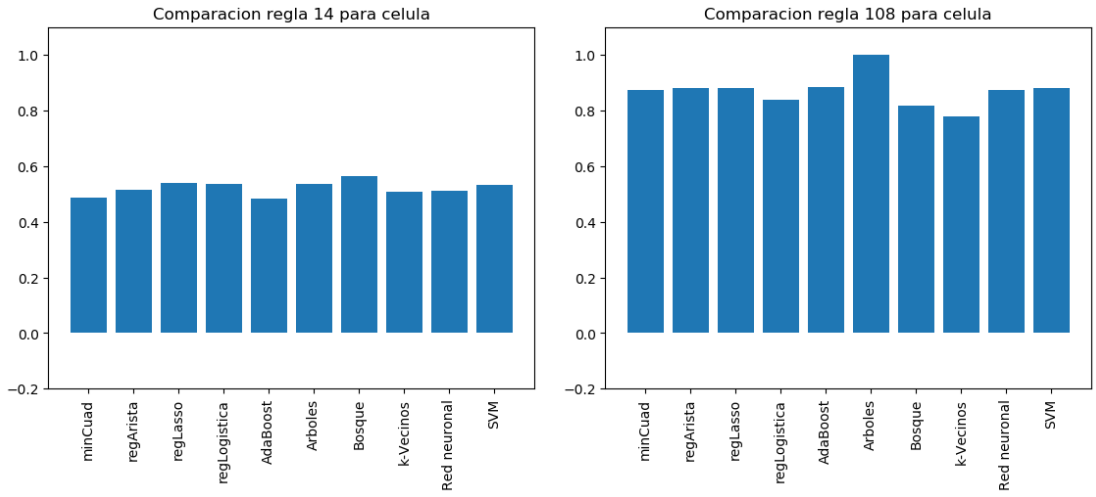
Para las reglas 14 (Fig. 4.15a) y 108 (Fig. 4.15b), ambas de la clase II de Wolfram, las cosas cambian bastante.

Aunque antes los algoritmos predecían con probabilidad 1 la mayoría para el autómata 108, ahora la mayoría de algoritmos de aprendizaje predicen las tres primeras casillas con probabilidad en torno a 0.6 (salvo por los árboles de decisión, que alcanzan un pico cercano al 0.9). En comparación al azar, con precisión 0.125, está bastante bien, ya que aumenta la precisión en 0.4 en la mayoría de los casos, y los árboles de decisión aprenden bastante bien el funcionamiento de este autómata.

Para la regla 14 los resultados son mucho peores, aunque su predicción mejora el azar: la precisión de los algoritmos está en torno a 0.2. Lo que supone una mejora porcentual considerable.

Esto nos indica que, aunque los autómatas de la clase II puedan ser similares en algunos aspectos, en ocasiones encontraremos algunos que sean fáciles de aprender por parte de los algoritmos, y otros que se resistirán a estos.

Pasando a los autómatas de la clase III, para la regla 30 (Fig. 4.16a), como era de esperar, la precisión es parecida a la del azar, tal y como pasaba en el caso del **problema de la mayoría**. Nuestro objetivo no está en la posible predicción de la evolución del autómata en el caso caótico, y no porque sea imposible, sino porque los algoritmos que hemos usado de ML son muy básicos y no son capaces de aprender sin memoria



(a) Aprendizaje de la regla 14.

(b) Aprendizaje de la regla 108.

Figura 4.11

a partir de una muestra temporal.

Para la regla 126 (Fig. 4.16b) podemos observar cierta mejoría: la mayoría de algoritmos tienen una precisión de en torno a 0.2. Aunque es algo mejor que el azar, tampoco nos da resultados significativos.

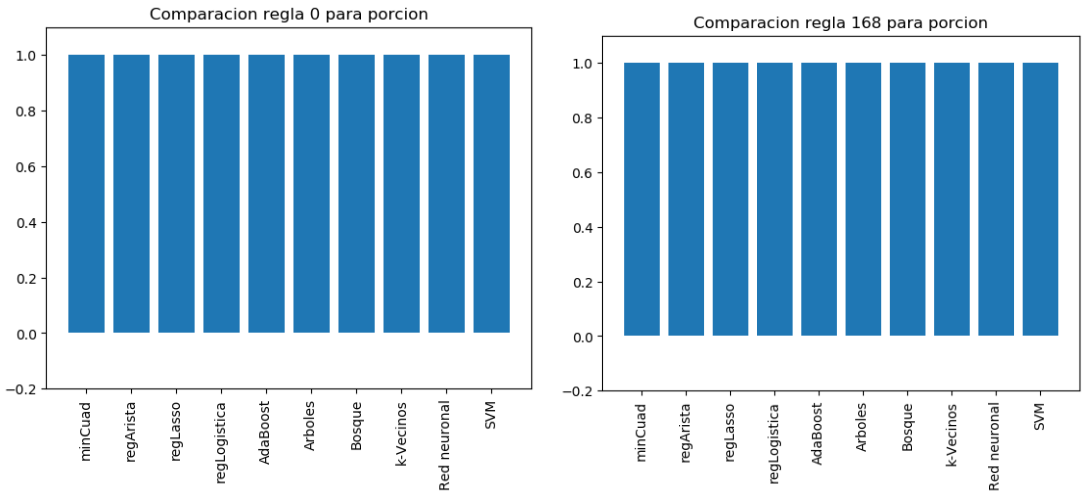
Esto nos permite deducir algo que esperábamos, y es que, debido al caos subyacente a los autómatas de la clase III, las predicciones de los algoritmos de aprendizaje básicos serán bastante pobres cuando nos enfrentemos a dinámicas caóticas.

Por último, tanto a la regla 54 (Fig. 4.17a) como a la regla 110 (Fig. 4.17b) les pasa algo similar a los de la clase III de Wolfram: el caos subyacente hace que sea difícil para los algoritmos de aprendizaje predecir las tres primeras células del autómata tras un número fijo de iteraciones, y los resultados no son mucho mejores que los del azar.

Reservamos para el último capítulo de la memoria el conjunto de conclusiones que podemos extraer de los experimentos realizados.

4.3.4 Otros experimentos

Aunque en las secciones anteriores nos hemos centrado en los métodos de clasificación por la naturaleza de los rasgos estudiados, se podrían realizar otras aproximacio-



(a) Aprendizaje de la regla 0.

(b) Aprendizaje de la regla 168.

Figura 4.14

nes que utilizasen métodos de regresión, como puede ser la predicción de la **densidad** final de los autómatas (al final de los pasos dados). Los experimentos adicionales que indicamos en este apartado se muestran solo a modo de ejemplo, como muestra del tipo de análisis de la complejidad de los autómatas que se puede llevar a cabo, así como de la capacidad predictiva de los algoritmos usados.

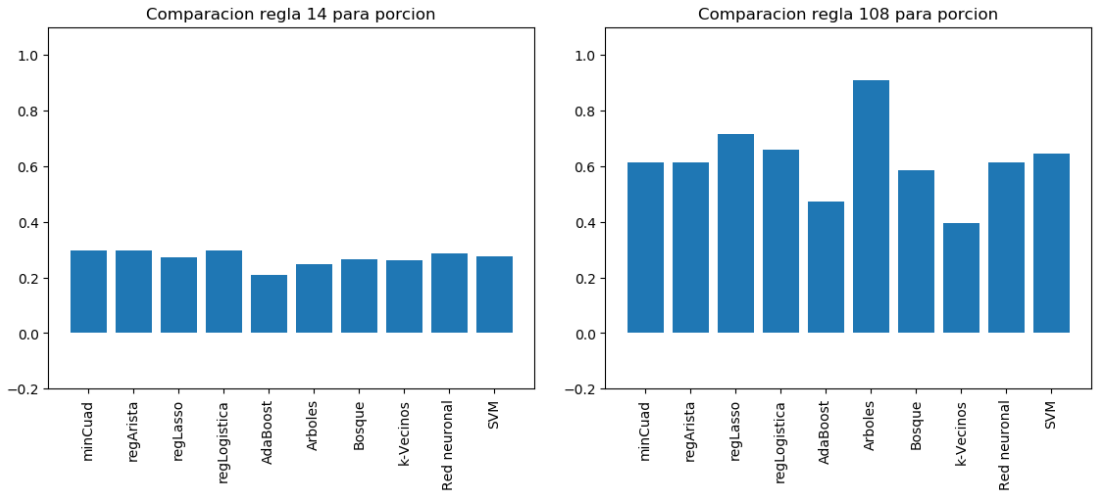
La densidad de un autómata en una configuración determinada se define como:

$$Densidad = \frac{\text{nº de células vivas}}{\text{nº de células}}$$

Utilizando, por ejemplo, el algoritmo de regresión por mínimos cuadrados se consiguen las precisiones que se ven en la figura 4.18 para los 88 autómatas señalados a lo largo de esta memoria.

La implementación del algoritmo de regresión por mínimos cuadrados que hemos usado mide el coeficiente R^2 de la aproximación y compara los resultados del algoritmo con el azar. Si el algoritmo es mucho peor, se obtienen valores negativos. Se debe tener en cuenta que existen algoritmos de regresión (por ejemplo, las redes neuronales) que pueden obtener resultados más interesantes que los arrojados por nuestra prueba.

Como era de esperar, los resultados de predicción usando esta regresión sobre autómatas mínimamente complejos son muy pobres.



(a) Aprendizaje de la regla 14.

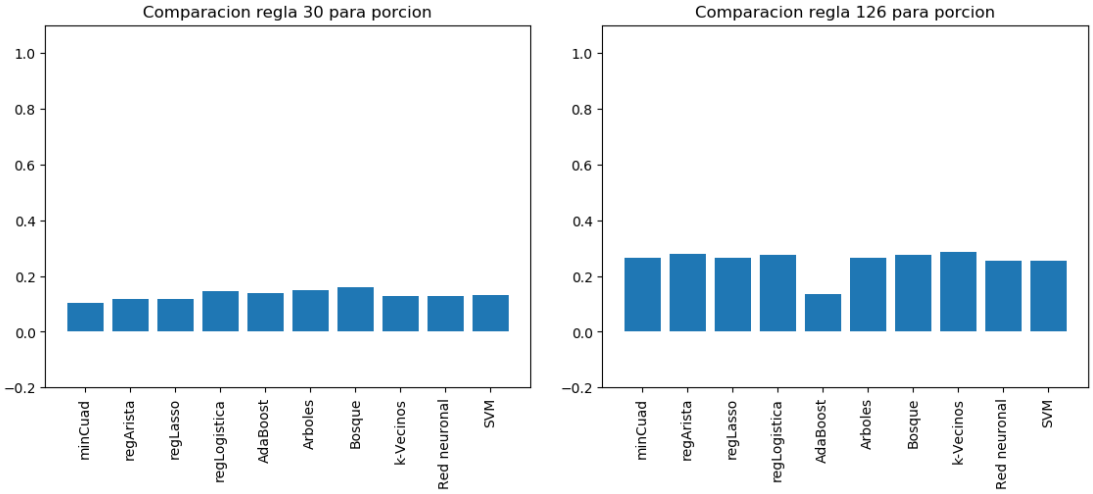
(b) Aprendizaje de la regla 108.

Figura 4.15

Por otra parte, también es interesante analizar qué ocurre cuando se consideran métricas como la distancia entre las soluciones obtenidas en la predicción de tres células en comparación a la configuración real arrojada por la simulación del autómata. No es lo mismo predecir una de las tres células erróneamente, que predecirlas todas mal.

En las figuras 4.19, 4.20, 4.21 y 4.22 se muestran mediciones de la distancia de Levenshtein media entre las predicciones obtenidas y las soluciones reales para los 8 autómatas estudiados (esencialmente, se mira el número de errores que se pueden cometer, que por las características del experimento diseñado, puede varias entre 0 y 3). Obsérvese que, en el caso de estar trabajando con cadenas binarias, la distancia de Levenshtein coincide con la distancia de Manhattan.

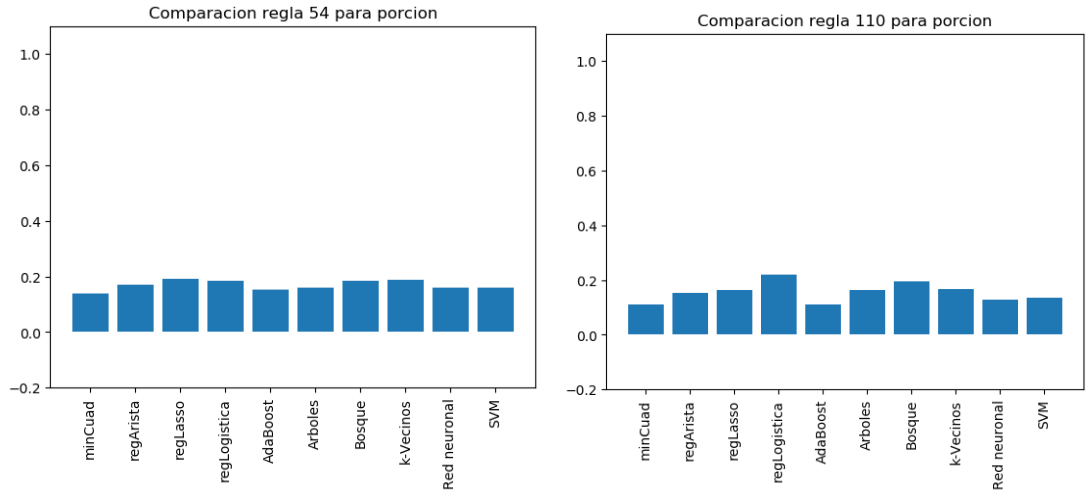
Teniendo en cuenta que la distancia máxima es 3, en las gráficas indicadas podemos observar resultados similares a los obtenidos con los experimentos anteriores: para los autómatas de la clase I de Wolfram se obtiene una distancia media 0; los de la clase II están uno en torno a 1.5 y el otro muy por debajo (en torno a 0.5); y los de la clase III y IV están en torno a 1.5 debido al comportamiento caótico o complejo. Se podría estudiar intentar optimizar los algoritmos para que penalizaran además la distancia Manhattan, obteniendo así mejores resultados.



(a) Aprendizaje de la regla 30.

(b) Aprendizaje de la regla 126.

Figura 4.16



(a) Aprendizaje de la regla 54.

(b) Aprendizaje de la regla 110.

Figura 4.17

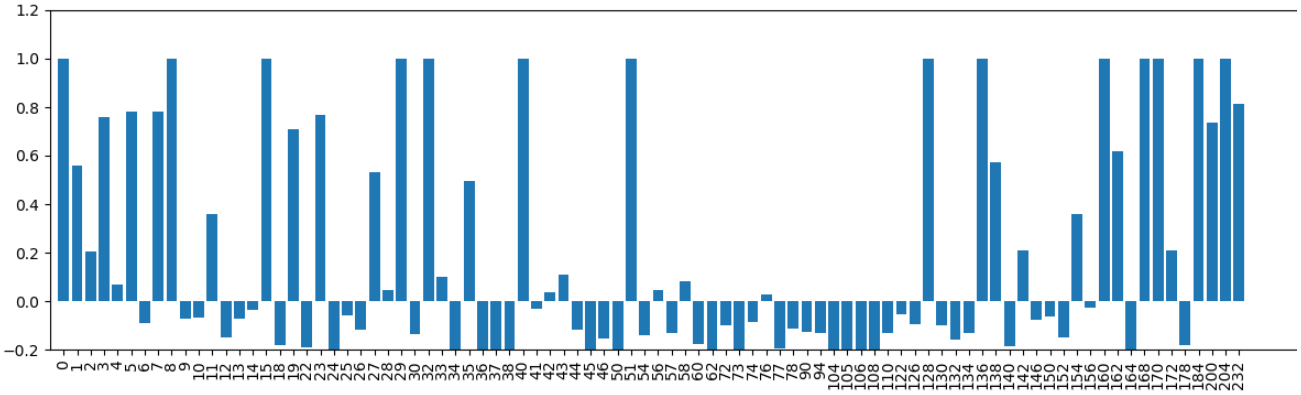
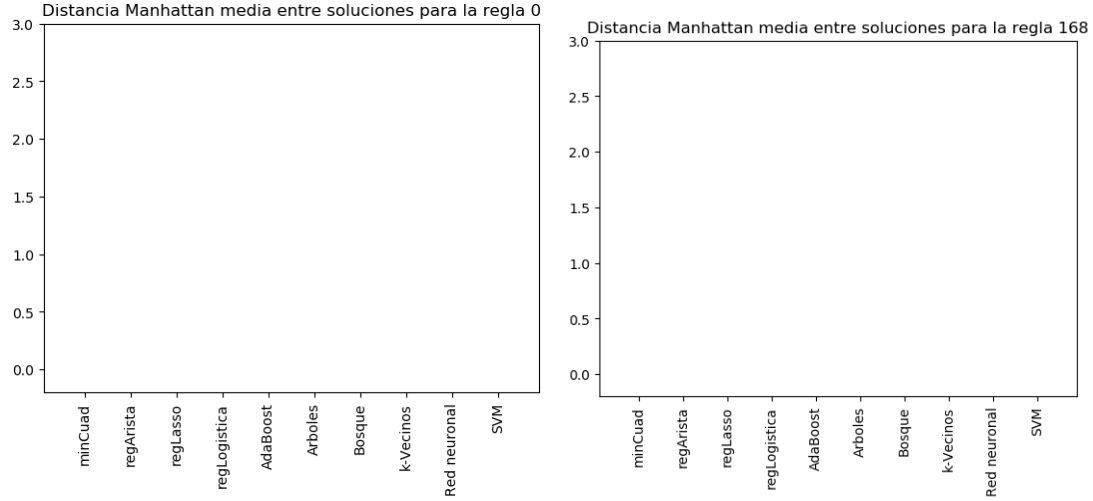
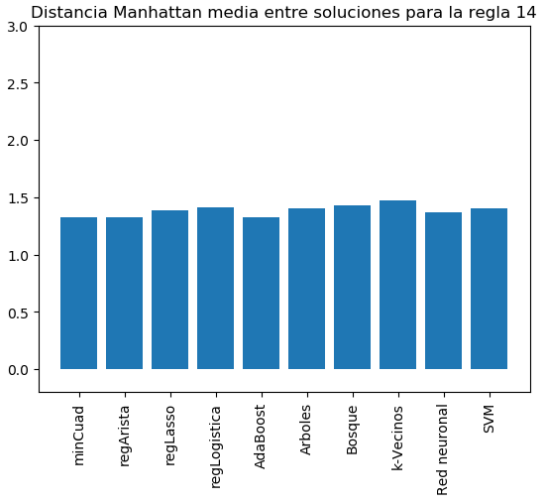


Figura 4.18: Precisión del algoritmo de regresión por mínimos cuadrados

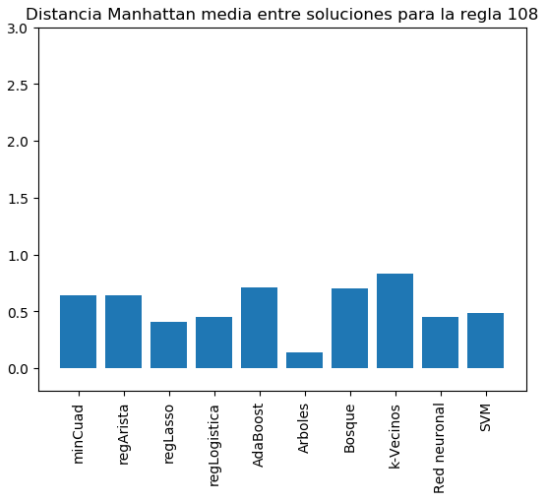


(a) Distancia de Levenshtein para la regla 0. (b) Distancia de Levenshtein para la regla 168.

Figura 4.19

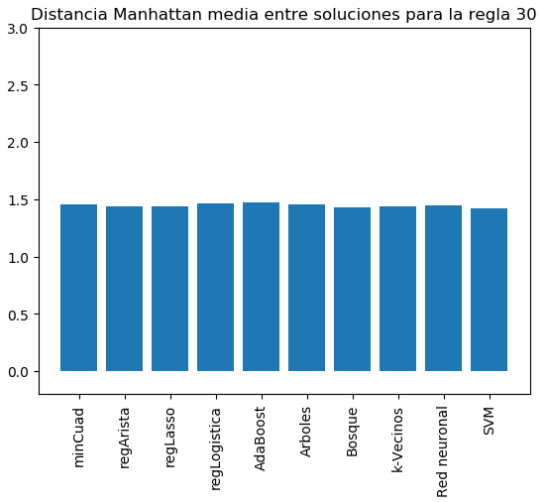


(a) Distancia de Levenshtein para la regla 14.

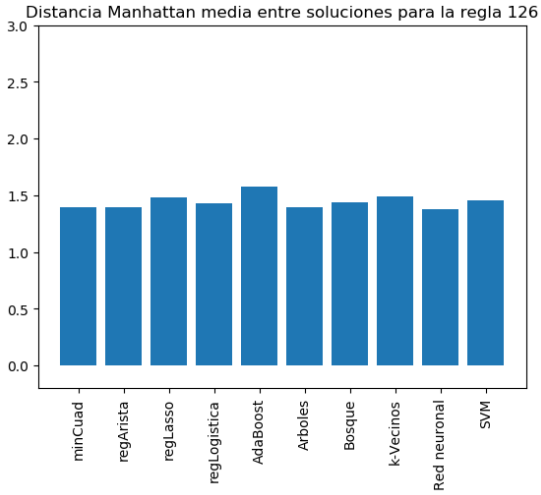


(b) Distancia de Levenshtein para la regla 108.

Figura 4.20

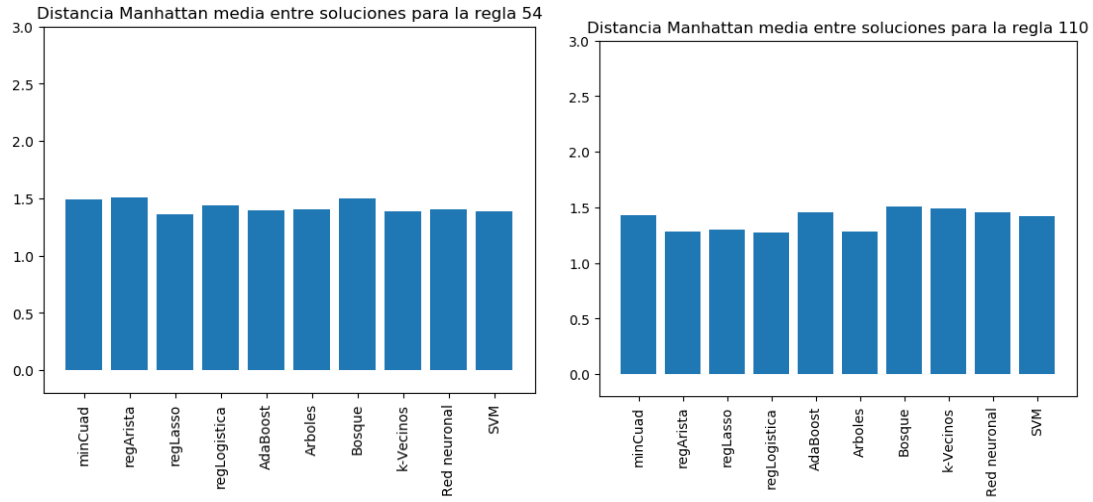


(a) Distancia de Levenshtein para la regla 30.



(b) Distancia de Levenshtein para la regla 126.

Figura 4.21



(a) Distancia de Levenshtein para la regla 54. (b) Distancia de Levenshtein para la regla 110.

Figura 4.22

5 Conclusiones y Trabajo Futuro

5.1 Conclusiones

En el caso de los autómatas celulares, hemos podido extraer las siguientes conclusiones:

- Los autómatas de la clase I de Wolfram, como era de esperar, son fácilmente aprendibles por los algoritmos de ML.
- Ciertos rasgos de algunos autómatas de la clase II de Wolfram pueden ser aprendidos hasta cierto punto y dependiendo del autómata, aunque algunos algoritmos pueden obtener resultados muy buenos (ver Figura 4.15b).
- Los autómatas de la clase III de Wolfram son difícilmente aprendibles por parte de los algoritmos, debido al caos que subyace en ellos. Es precisamente en este tipo de dinámicas caóticas donde debemos esperar el rendimiento más pobre en la búsqueda de patrones, ya que este comportamiento es el más cercano al aleatorio que se puede conseguir por medio de un sistema determinista.
- Los autómatas de la clase IV de Wolfram tienen rasgos que no son aprendibles por los algoritmos de ML, y otros que son fácilmente aprendibles (ver Figura 4.17b y Figura 4.9b). Esta clase es la más interesantes desde el punto de vista del aprendizaje, ya que, pese a presentar un grado de complejidad muy elevado, su comportamiento no es caótico y podemos observar patrones visuales que indican que hay reglas subyacentes no explicitadas todavía.

Sin duda, los algoritmos de ML utilizados para realizar este estudio han sido claramente insuficientes. Por supuesto, los objetivos originales del trabajo no cubrían la posible aplicación de algoritmos más complejos (usando redes neuronales recurrentes o con memoria), y se centraban más en justificar y motivar el uso de ML para el aná-

lisis de la complejidad, en este caso representada por la dinámica determinista, pero compleja, de pequeños autómatas.

En este sentido, y a pesar de la pobreza de los resultados obtenidos desde un punto de vista experimental, se consideran cumplidos los objetivos planteados y, como debe ser en todo trabajo de iniciación a la investigación, son más interesantes las puertas que abre que las que ha conseguido cerrar.

Un apunte extra a considerar es que si comparamos los resultados obtenidos de entrenar algunos algoritmos sobre todas las reglas, con los obtenidos de entrenar otros algoritmos sobre las mismas reglas, a veces unos obtienen resultados mejores que los otros y otras veces es al revés. Por ejemplo, se muestra a continuación la comparación entre los resultados obtenidos por los árboles de decisión y los obtenidos por las redes neuronales (Fig. 5.1). En un momento de auge de las redes neuronales, y a pesar de ser modelos mucho más flexibles para muchísimos de los problemas planteados en Inteligencia Artificial, parece mostrarse aquí una versión local del **Teorema No-Free-Lunch**, en el que no hay un algoritmo de aprendizaje que supere a los demás en todos los casos posibles.

5.2 Trabajo Futuro

Por una parte, aunque el problema de clasificar cada autómata en una clase es indecidible (ver 2.4), observando las precisiones obtenidas por los distintos algoritmos se puede estimar una primera clasificación basada únicamente en los resultados experimentales y el rendimiento dado por los diversos algoritmos:

Los autómatas que permiten precisiones similares a las del azar pertenecen a la clase III y los que obtienen precisión 1 pertenecen a la clase I. Algunos autómatas de la clase II y los de la clase IV necesitarían de más trabajo para ser distinguibles, quizás de mayor potencia computacional, mayor entrenamiento, y un cambio en el tipo de algoritmos utilizados.

En este sentido, es interesante poder asociar la complejidad del autómata a la del algoritmo más sencillo que es capaz de predecir su dinámica. Algo que se relaciona con las técnicas que estudian la complejidad de un sistema desde la complejidad descriptiva de los sistemas que lo simulan.

Por otra parte, el enfoque dado a este trabajo se puede extender ahora en direc-

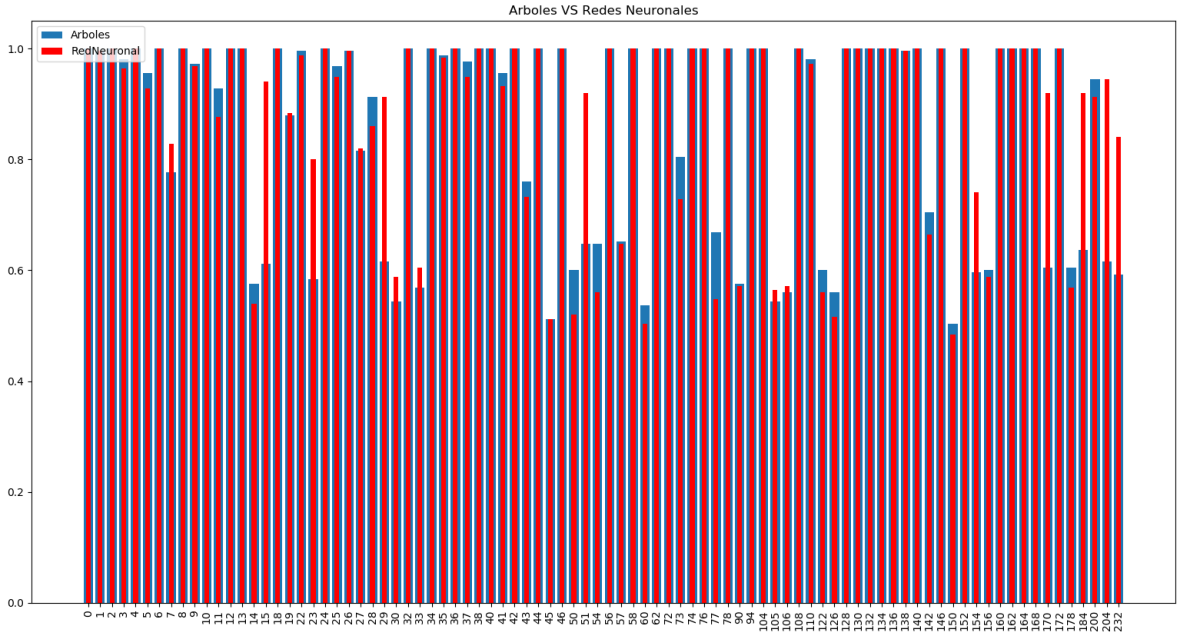


Figura 5.1: Comparación entre la precisión alcanzada por los árboles de decisión y la conseguida por las redes neuronales

ciones claramente diferenciadas, pero que convendría abordar de manera unificada y sistemática:

- Aumentando la dimensión y/o el tamaño de los autómatas. Hay sistemas bidimensionales cuya complejidad evolutiva ha sido tratada desde varios ángulos, y que sería interesante abordar ahora con las herramientas computacionales de búsqueda de patrones que se están desarrollando para otras áreas (como las nombradas Redes Convolucionales). Entre estos autómatas de dimensión superior destaca el conocido Juego de la Vida, cuya dinámica compleja se ha demostrado ilimitada y con una potencia de cálculo similar a las máquina de Turing.

- Aumentando la cantidad de ejemplos de entrenamiento y test que se les da a los algoritmos de aprendizaje. Es sabido que el tamaño y calidad de los conjunto de entrenamiento son esenciales para obtener resultados predictivos interesantes en todos los modelos de ML. Cuando tratamos con dinámicas que

se saben complejas (incluso caóticas), este factor se hace más acuciante, ya que el espacio de fase en el que se mueven las órbitas de las configuraciones de los sistemas crece exponencialmente y responde a factores mucho más desconocidos.

- Utilizando distintos algoritmos de aprendizaje a los aquí usados, o distintos parámetros para los mismos algoritmos. Como hemos comentado en párrafos anteriores, el uso de modelos tan simples es un limitante a la capacidad de reconocimiento de patrones complejos. Además, hemos de añadir que ninguno de los modelos utilizados está diseñado específicamente para el reconocimiento de patrones temporales (característica fundamental de los autómatas estudiados), por lo que la introducción de modelos más complejos, mejor adaptados a los estudios de dinámicas, y con mayor capacidad de ajuste, puede ser un factor determinante para poder aumentar la predicción y mejorar la clasificación resultante.

- Estudiando otros aspectos de los autómatas celulares que no sean la densidad o un trozo de estos. De nuevo, el proceso de simplificación necesario para los objetivos y características de un Trabajo Fin de Grado, nos han llevado a plantear un conjunto muy limitado de posibles experimentos a realizar. La elección ha sido clara: tomar una métrica local de la dinámica (el estado concreto de 3 de las células del autómata) y una métrica global (la densidad del autómata), pero sería deseable realizar un conjunto más amplio de experimentos sobre otro tipo de métricas con el fin de eavluar también hasta qué punto reflejan la dinámica compleja del proceso subyadente.

Extender esta forma de trabajo podría dar lugar a la creación de un algoritmo que nos permita clasificar los autómatas en clases (ya sean las de Wolfram, las de Culik-Yu o las de Li-Packard) con una probabilidad alta a pesar de ser un problema indecidible, o a descubrir rasgos sobre los autómatas que nos permitan avanzar en otros campos más allá de la matemática, como pueden ser la física, la biología o la ingeniería.

Como uno de los resultados fundamentales de este trabajo creemos que queda demostrado que no es necesario acudir a sofisticados generadores de complejidad para poner a prueba y ver los límites de nuestras herramientas matemático-computacionales de reconocimiento de patrones, es suficiente acudir a pequeños *inventos* matemáticos para encontrar un campo de juego lo suficientemente complejo como para disponer de retos que justifiquen la profundización y mejora de técnicas de aproximación y predicción.

Por último, la posibilidad de crear una base de datos pública a la que contribuir añadiendo los algoritmos de ML que mejor hayan aprendido un determinado rasgo de un determinado autómata puede ser una buena idea a largo plazo. Por una parte puede ayudar a descubrir propiedades de los autómatas, y por otra puede plantear nuevos desafíos a la hora de intentar predecir rasgos de autómatas caóticos.

6 Apéndice

En este capítulo se incluyen muestras de las 88 clases de equivalencia de autómatas celulares unidimensionales. Todos parten de una configuración inicial aleatoria, y permiten hacerse una idea con un simple vistazo de la complejidad de cada autómata.

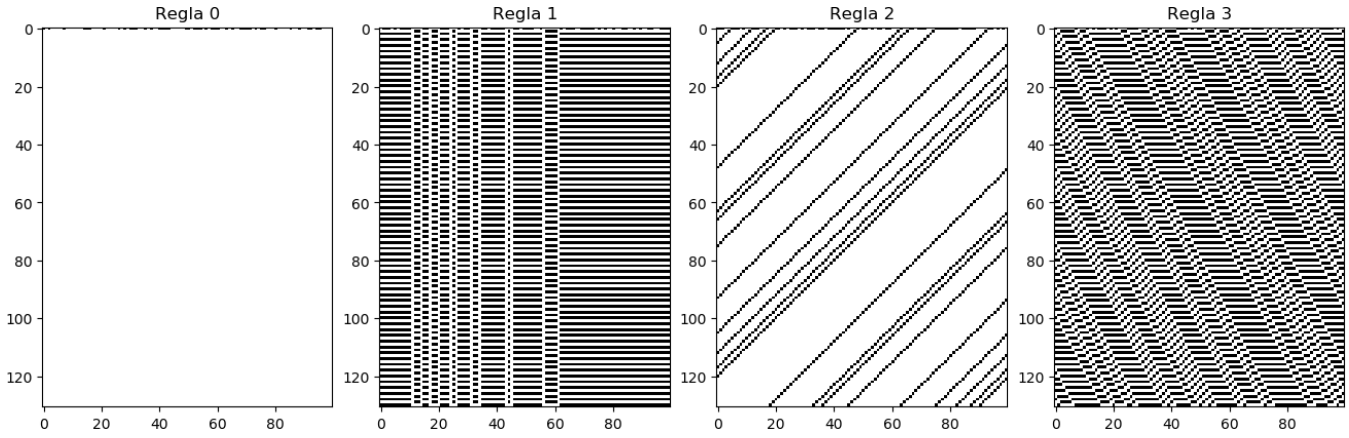


Figura 6.1

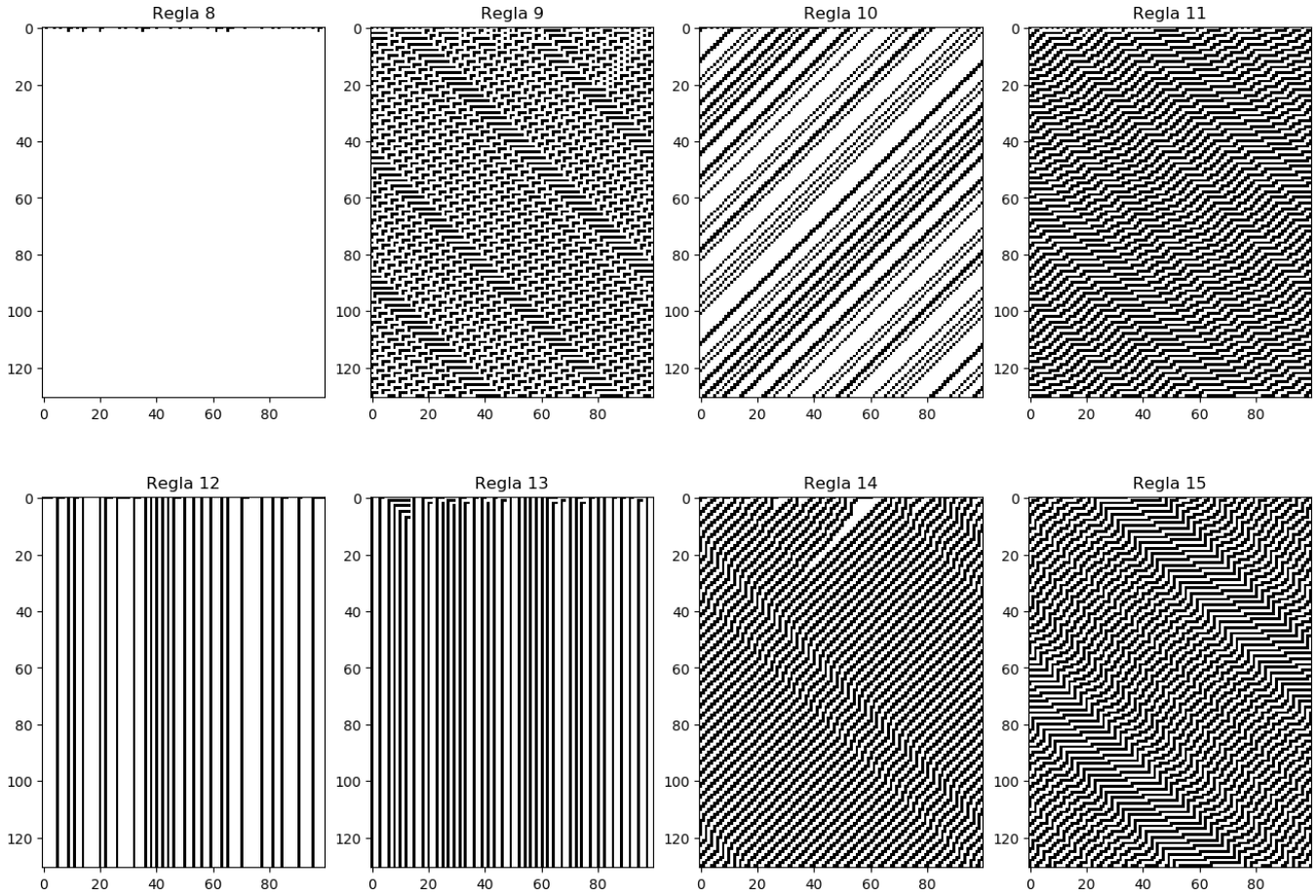
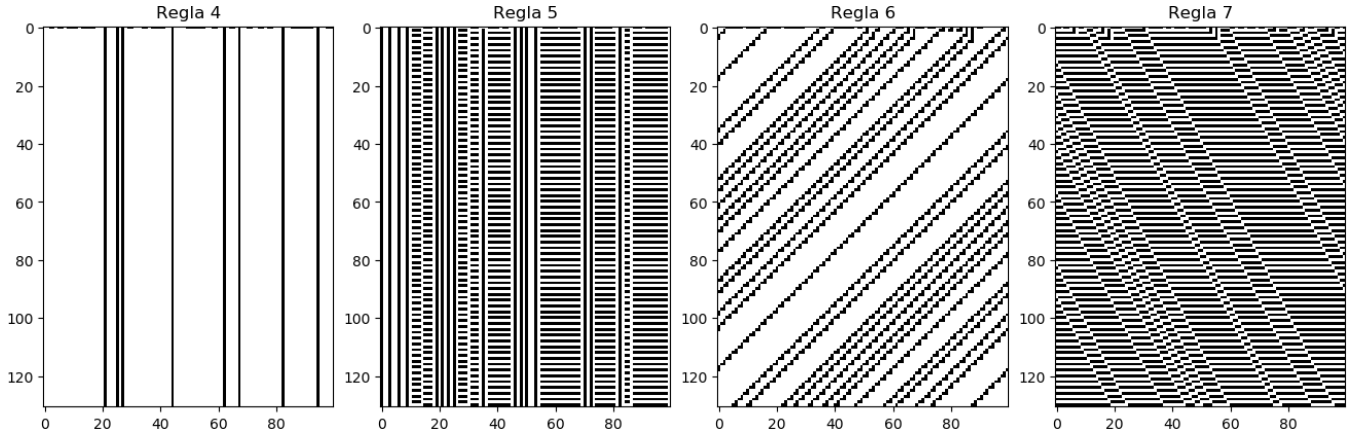

Regla 8
Regla 9
Regla 10
Regla 11
Regla 12
Regla 13
Regla 14
Regla 15

Figura 6.2


Regla 4
Regla 5
Regla 6
Regla 7

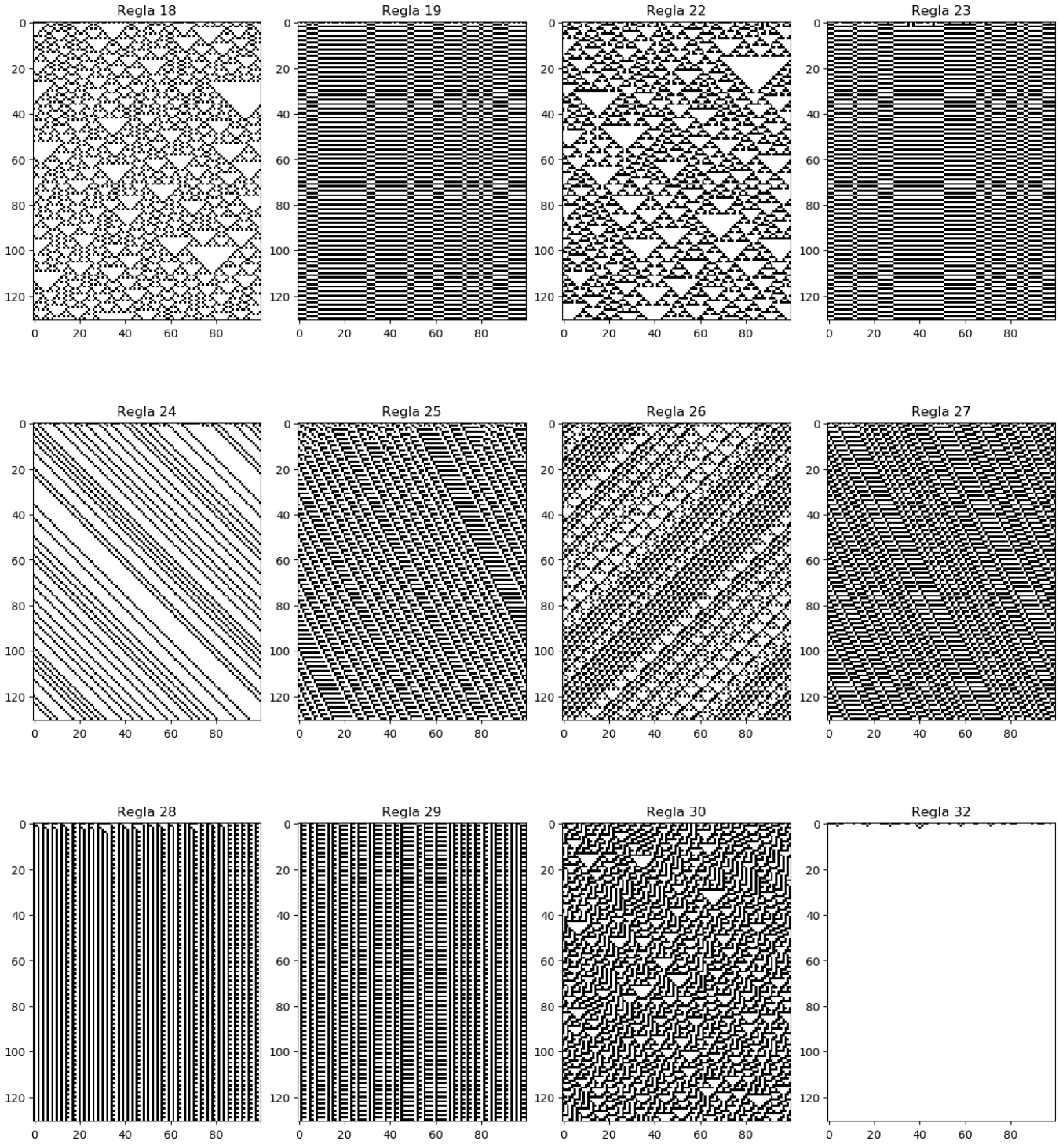
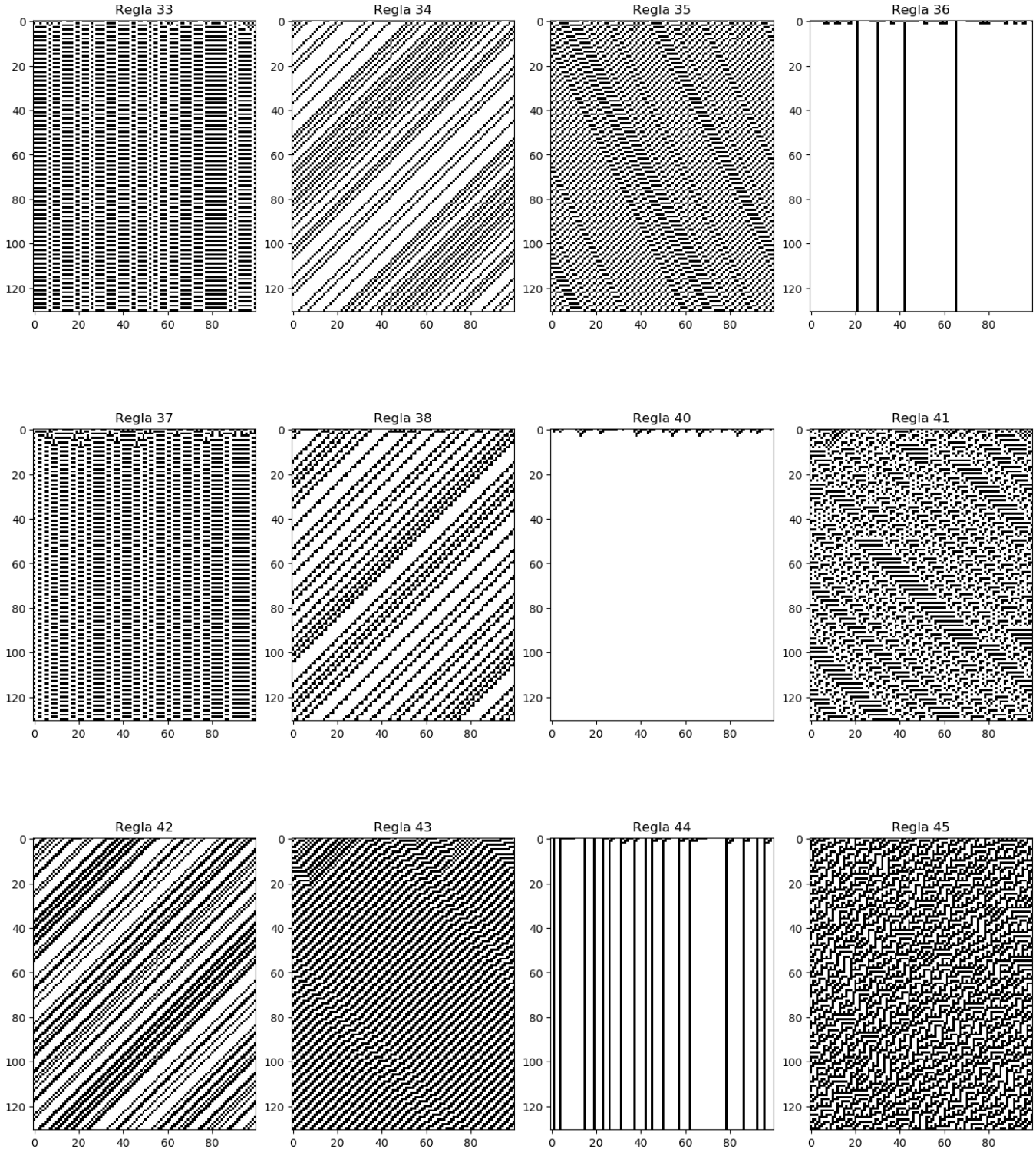

Regla 18
Regla 19
Regla 22
Regla 23
Regla 24
Regla 25
Regla 26
Regla 27
Regla 28
Regla 29
Regla 30
Regla 32

Figura 6.3


Regla 33
Regla 34
Regla 35
Regla 36
Regla 37
Regla 38
Regla 40
Regla 41
Regla 42
Regla 43
Regla 44
Regla 45

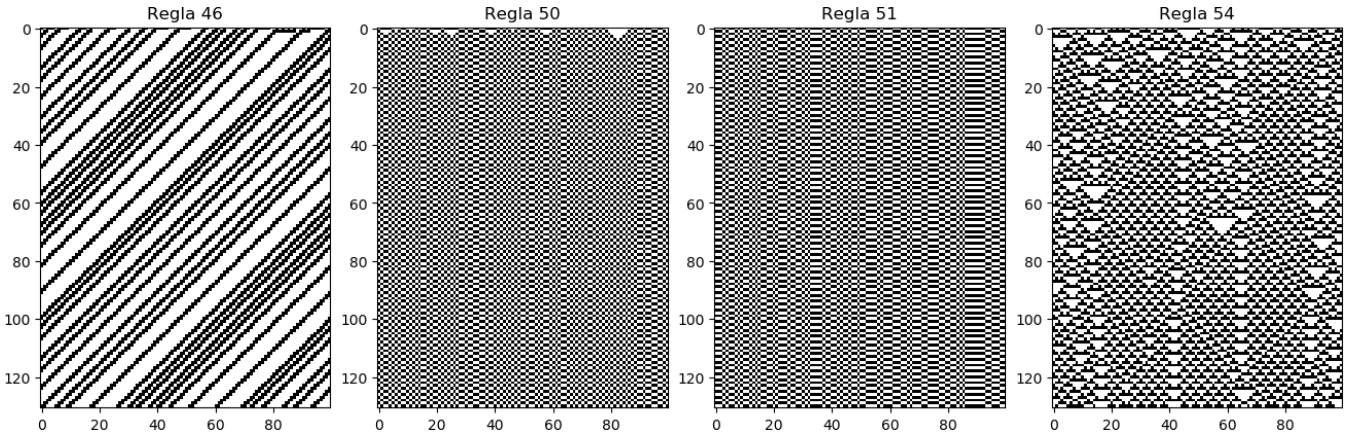
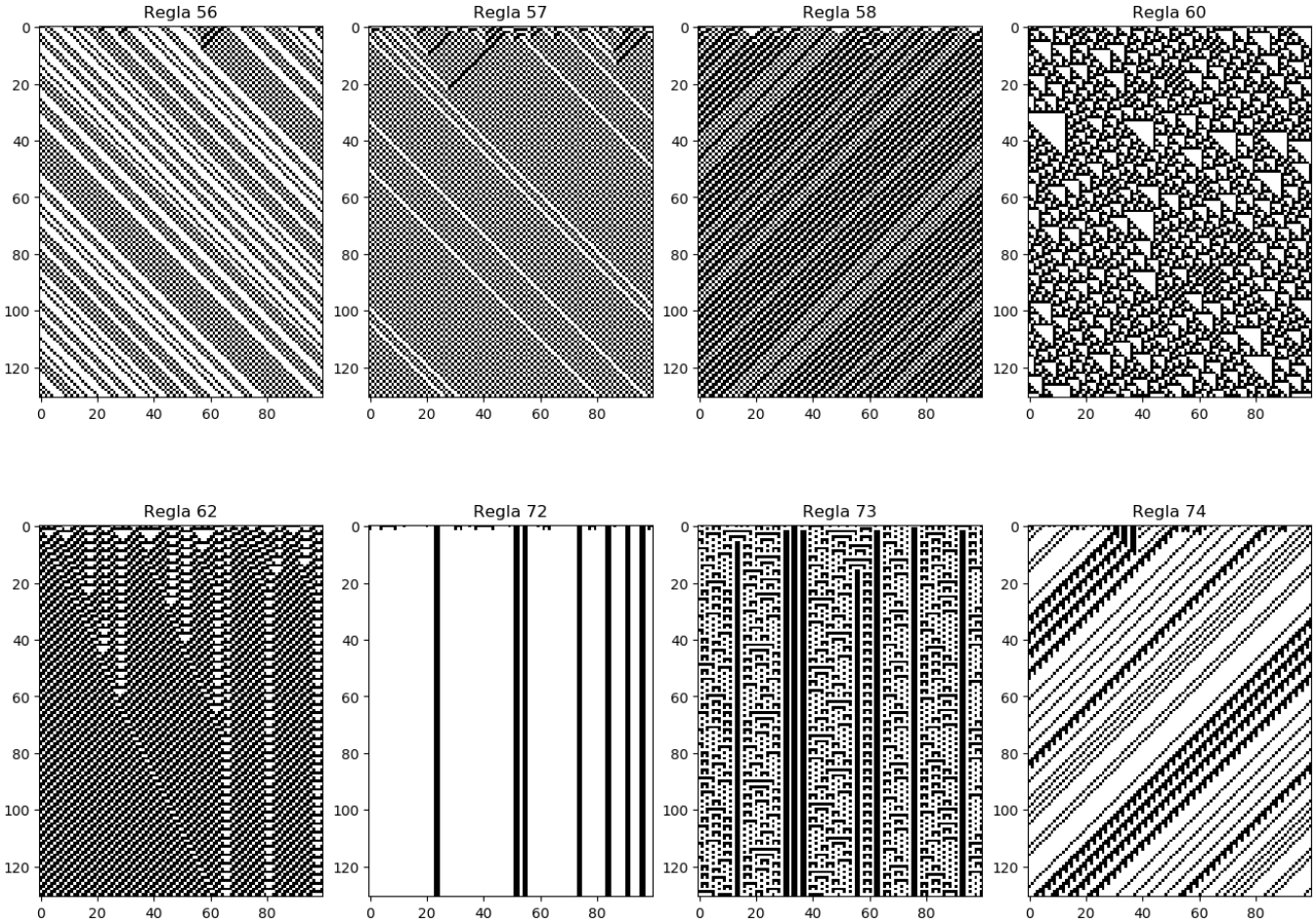

Regla 46
Regla 50
Regla 51
Regla 54

Figura 6.4


Regla 56
Regla 57
Regla 58
Regla 60
Regla 62
Regla 72
Regla 73
Regla 74

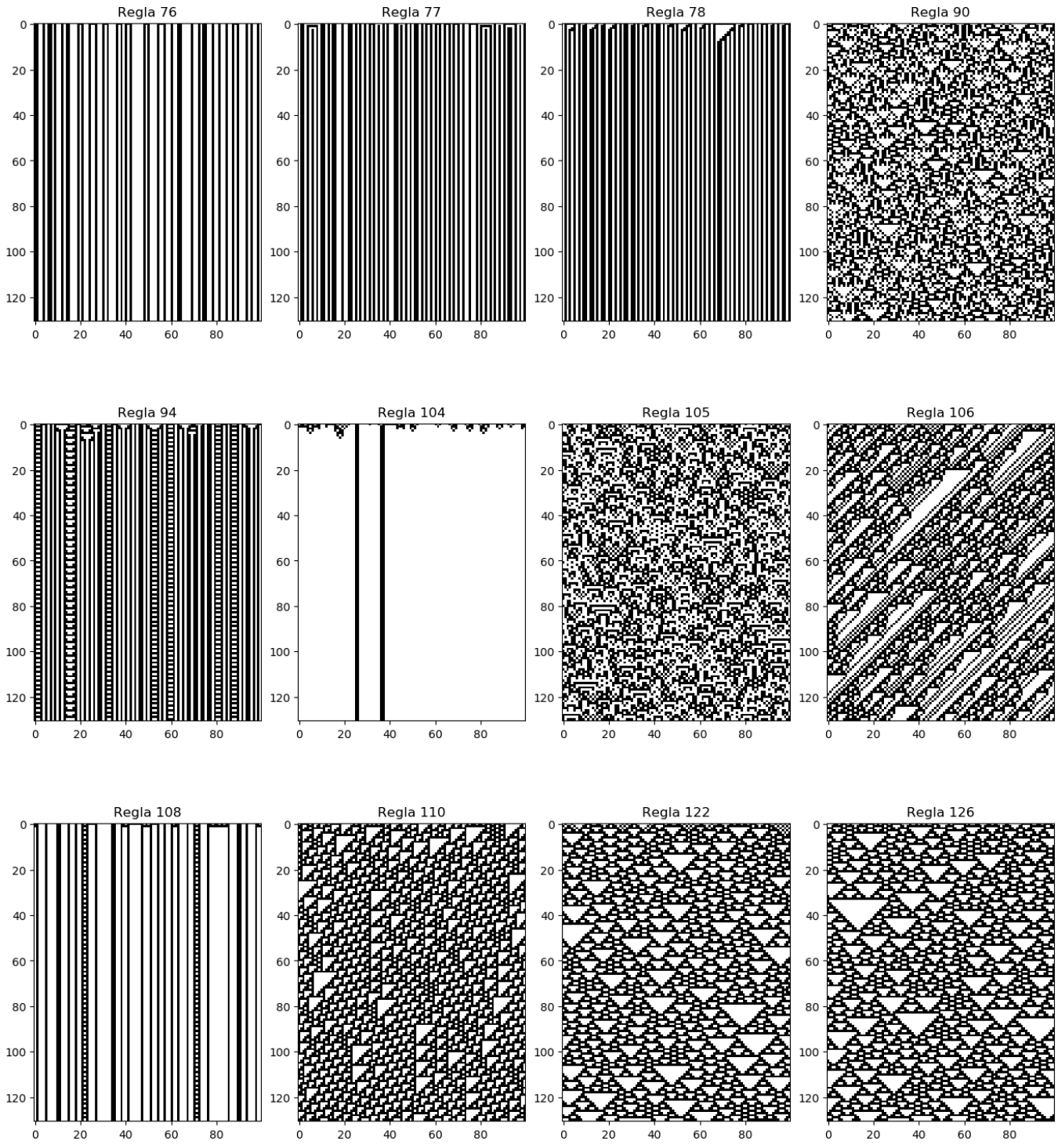
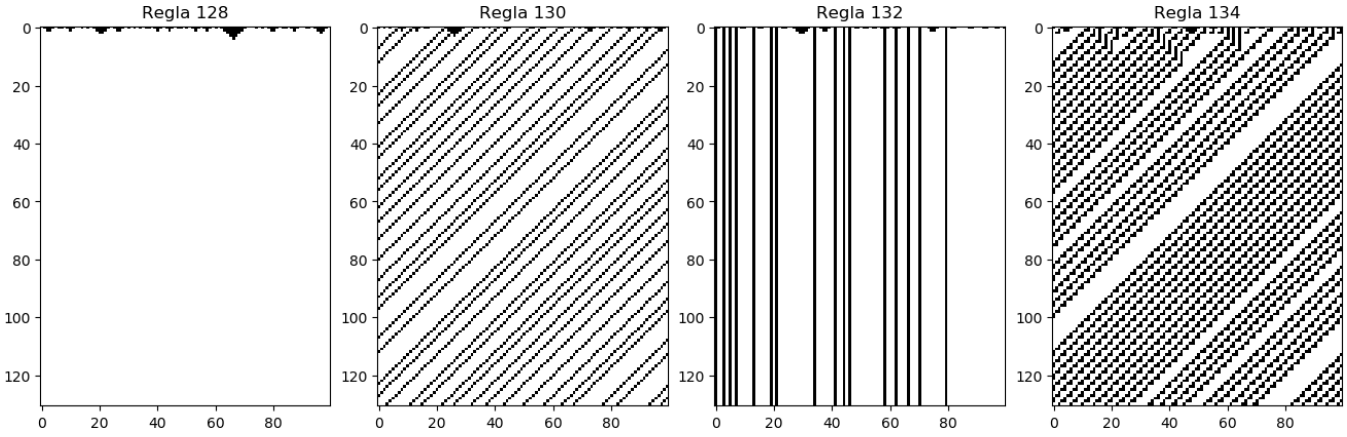
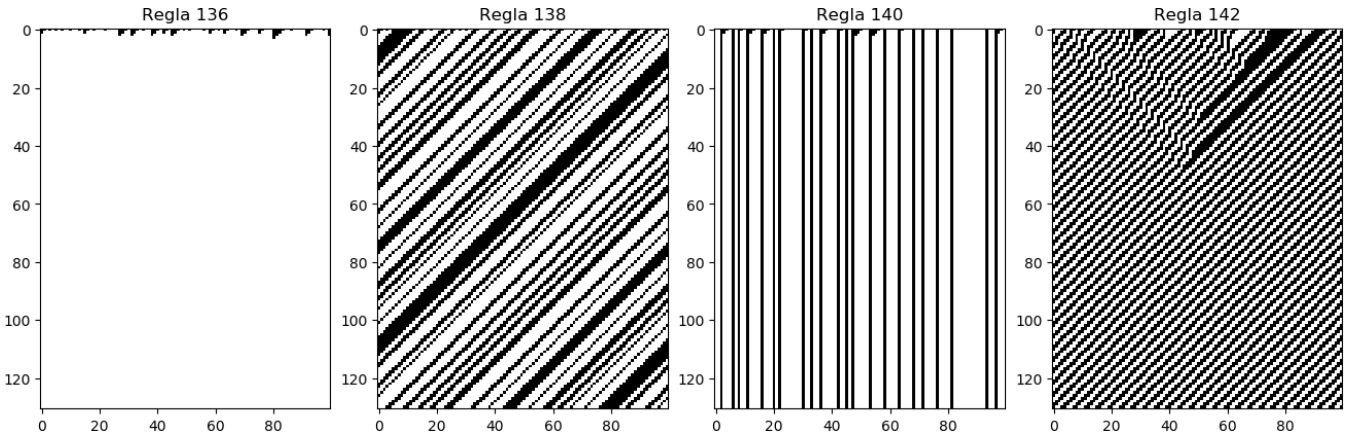
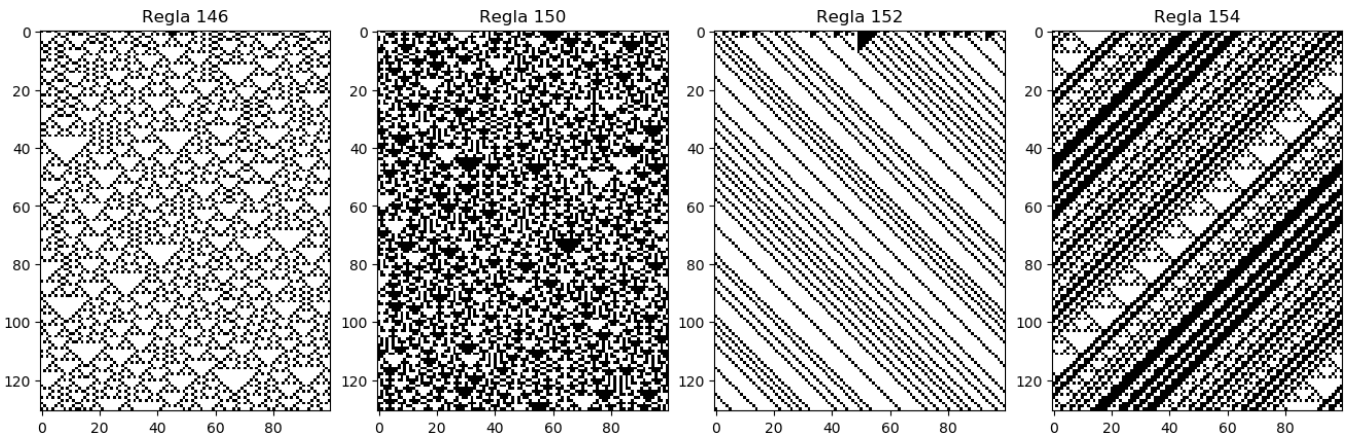

Regla 76
Regla 77
Regla 78
Regla 90
Regla 94
Regla 104
Regla 105
Regla 106
Regla 108
Regla 110
Regla 122
Regla 126

Figura 6.5


Regla 128
Regla 130
Regla 132
Regla 134


Regla 136
Regla 138
Regla 140
Regla 142


Regla 146
Regla 150
Regla 152
Regla 154

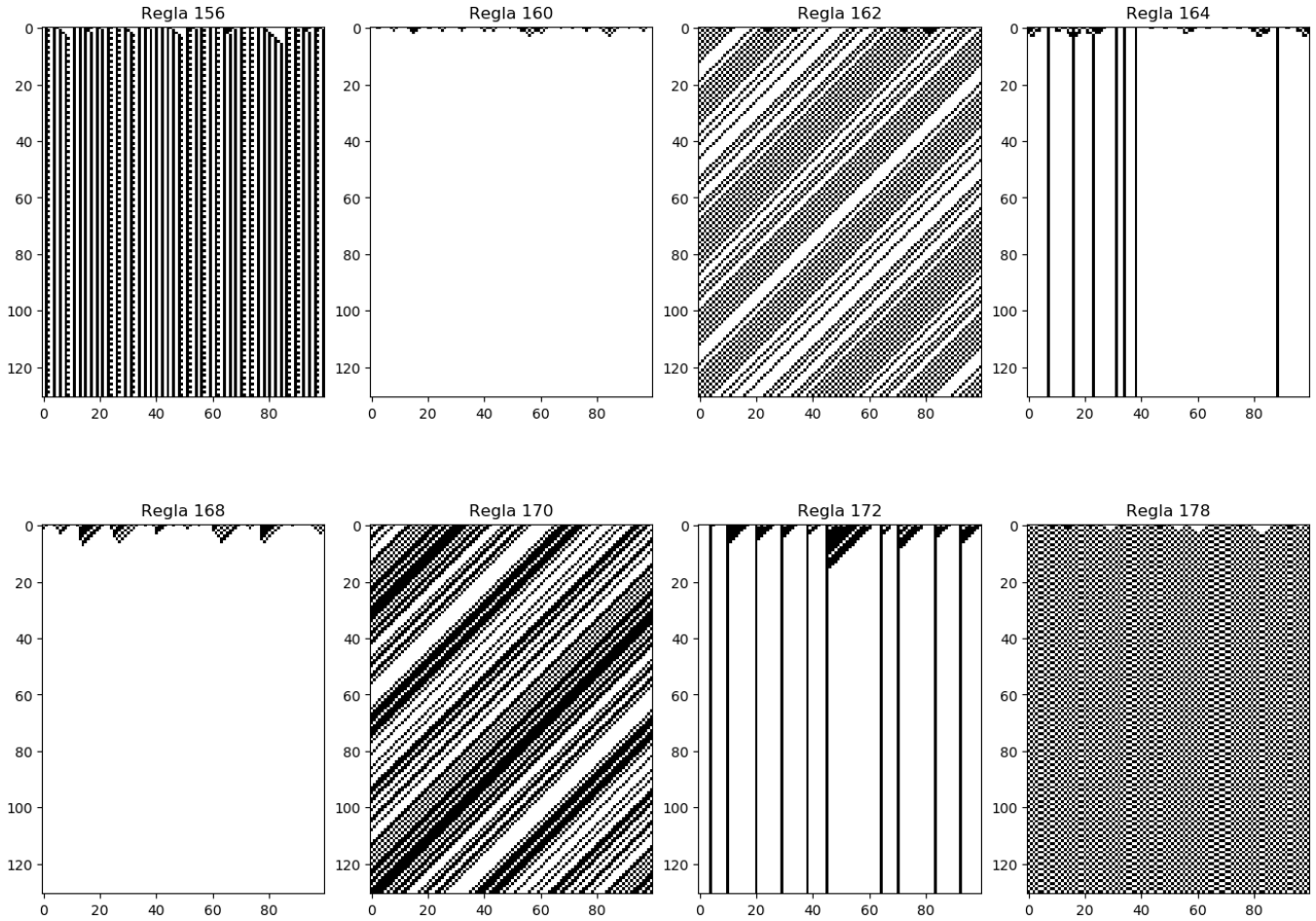
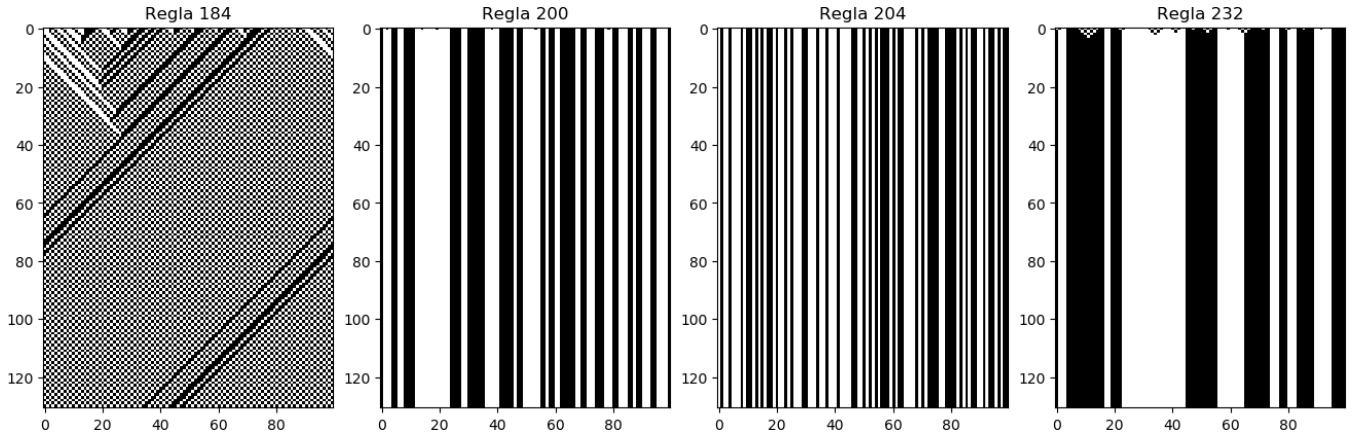

Regla 156
Regla 160
Regla 162
Regla 164
Regla 168
Regla 170
Regla 172
Regla 178

Figura 6.6


Regla 184
Regla 200
Regla 204
Regla 232

Bibliografía

[1] Stephen Wolfram, *A New Kind of Science*, Wolfram Media, 2002.

[2] Karel Culik y Sheng Yu,«Undecidability of CA classification schemes» *Complex Systems*, **2**, pp. 177–190, 1988.

[3] Wentian Li,«Phenomenology of nonlocal cellular automata» *Journal of Statistical Physics*, **68**, pp. 829–882, 1992.

[4] R. Barlovic, L. Santen, A. Schadschneider y M. Schreckenberg «Metastable states in cellular automata for traffic flow» *The European Physical Journal B - Condensed Matter and Complex Systems*, **5**, pp. 793–800, 1998.

[5] Wentian Li y Norman Packard «The Structure of the Elementary Cellular Automata Rule Space» *Complex Systems*, **4**, pp. 281–297, 1990.

[6] Stephen Wolfram «Universality and complexity in cellular automata» *Physica*, **D10**, págs. 1–35, 1984.

[7] Jürgen Albert y Karel Culik «A Simple Universal Cellular Automaton and its One-Way and Totalistic Version» *Complex Systems*, **1**, pp. 1–16, 1987.

[8] Shai Shalev-Schwartz y Shi Ben-David, *Understanding Machine Learning: From Theory to Algorithms*, Cambridge University Press, 2014.

[9] Andreas C. Müller y Sarah Guido, *Introduction to Machine Learning with Python*, O'Reilly Media, 2017.

[10] Arthur L. Samuel, «Some Studies in Machine Learning Using the Game of Checkers» *IBM Journal of Research and Development*, **3**, pp. 210–229, 1959

[11] Entrevista de Jeremy Bernstein a Marvin Minsky, The New Yorker, 14 de diciembre, 1981

[12] Ryan Poplin, Avinash V. Varadarajan, Katy Blumer, Yun Liu, Michael V. McConnell, Greg S. Corrado, Lily Peng y Dale R. Webster, «Prediction of Cardiovascular Risk Factors from Retinal Fundus Photographs via Deep Learning» *Nature Biomedical Engineering*, **2**, pp. 158–164, 2018

[13] John Ross Quinlan «Induction of Decision Trees» *Machine Learning*, **1**, pp. 81–106, 1986

[14] Keiron O'Shea y Ryan Nash, «An Introduction to Convolutional Neural Networks», *arXiv:1511.08458*, Noviembre de 2015