UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Escola Tècnica
Superior d'Enginyeria
Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Detección de palabras cortadas al inicio y al fin de línea en textos manuscritos

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Vicente Gras Mas

Tutor: Carlos David Martínez Hinarejos

Curso 2019 - 2020

Resumen

En este trabajo se describe el proceso de extracción de muestras con unas características definidas a partir de imágenes, las cuales son páginas escaneadas de un libro manuscrito, mediante visión artificial. Tras ello se aplicarán varios algoritmos de aprendizaje automático y diferentes técnicas aplicables a las muestras para detectar cuándo una palabra al final o al principio de una línea del texto es partida o no, es decir, si continúa en la siguiente línea o es continuación de la línea previa o por el contrario está entera.

Palabras clave: Procesado de texto manuscrito, procesado de documentos, aprendizaje automático, visión artificial.

Abstract

This project describes the extraction process of samples with defined features from images, which are scanned pages of a manuscript book through artificial vision. After that, a few machine learning algorithms and different techniques will be applied on the samples to detect when a word at the end or at the beginning of a line is splitted or not, that is, if the words continues on the next line or is a continuation of the previous line or is not and the word is not splitted.

Keywords : handwritten text processing, document processing, machine learning, artificial vision.

Tabla de contenidos

1. Introducción 7

1.1. Motivación 7

1.2. Objetivos 8

1.3. Estructura 8

2. Estado del arte 11

2.1. Visión por computador 11

2.2. Técnicas de aprendizaje automático 12

2.2.1. Algoritmos supervisados 16

2.2.2. Algoritmos no supervisados 26

2.3. Proyectos similares 29

3. Análisis del problema 31

3.1. Identificación y análisis de soluciones posibles 31

3.2. Solución propuesta. 34

4. Diseño de la solución 35

4.1. Arquitectura del sistema 35

4.2. Diseño detallado 36

4.2.1. Preproceso 36

4.2.2. Proceso 37

4.2.3. Postproceso 38

4.3. Tecnología utilizada 38

4.3.1. Software 38

4.3.2. Hardware 39

5. Desarrollo de la solución propuesta 41

6. Pruebas 49

7. Conclusiones 65

7.1. Relación del trabajo desarrollado con los estudios cursados 66

8. Trabajos futuros 67

9. Referencias 69

Apéndice I 73

1. Introducción

En esta introducción situaremos este proyecto en su contexto. En primer lugar hablaremos sobre la motivación que nos ha llevado a trabajar en este proyecto; a continuación veremos diferentes objetivos buscados en el desarrollo de este TFG; por último, haremos un recorrido rápido a todo el contenido que nos vamos a encontrar a lo largo de este proyecto.

1.1 Motivación

La digitalización de imágenes y su procesado para diferentes usos en un equipo se encuentra a la orden del día. Desde sanidad [1] pasando por industria [2] hasta automovilismo [3], muchos sectores usan procesado de imagen digital. Pero no solo el procesamiento de dichas imágenes es importante, también lo es el postproceso que se hace sobre ellas para extraer información que nos dé una ventaja a la hora de cumplir nuestros objetivos. Por ejemplo, en el campo de la visión artificial automovilística, nuestro objetivo sería conseguir un buen procesado de imagen para reconocer diferentes formas (señales, peatones, otros vehículos…) para lograr que el vehículo circule con seguridad por la carretera.

En vista de toda la capacidad que tiene esta tecnología, siempre he tenido en cuenta que tener un buen conocimiento en este campo de la computación es muy importante, no solo por su impacto en el mundo laboral, sino por todos los avances que puede significar en un futuro cercano a la raza humana en cuanto a cómo percibe y actúa en su entorno.

Teniendo en cuenta que la mención de computación en el grado de ingeniería informática otorga una base de estas tecnologías, pero no una buena profundización (ya que se necesitan muchas horas dada la gran cantidad de herramientas y técnicas que existen) mi elección estaba clara, ya que este TFG me permite estudiar en profundidad técnicas y métodos para resolver un problema que implica visión artificial y aprendizaje automático para resolver un problema de reconocimiento de formas en una imagen.

1.2 Objetivos

El objetivo de este proyecto es conocer técnicas de procesamiento de imágenes y hacer un estudio de diferentes técnicas de aprendizaje automático para detectar palabras partidas (palabra que no cabe en la misma línea y se parte para ser finalizada en la siguiente línea, se aprecia un ejemplo en la figura 1) en textos antiguos. Lógicamente, esto no va a ser tarea fácil; para aproximarse a un resultado aceptable se han probado diferentes técnicas con diferentes parámetros.

¿Qué prefieres, chocolate blanco o cho-
colate con leche?

Figura 1. La palabra *chocolate* en este fragmento de texto es un ejemplo de una palabra partida.

Esto no es un trabajo definitivo, pero es una buena aproximación para seguir estudiando el problema.

1.3 Estructura

Para facilitar la lectura de este documento expondremos un esquema a continuación, explicando brevemente cada uno de los apartados. No entraremos en detalle en los diferentes subapartados, ya que explicaremos solamente una visión global de cada uno.

- Introducción: se pondrá en situación sobre el objetivo de este proyecto, la motivación encontrada para realizarlo y los objetivos que se esperan cumplir.

- Estado del arte: se explicará cuál es el momento actual de la tecnología usada y su base tecnológica.

- Análisis del problema: se llevará a cabo un análisis de nuestro problema y cómo va a ser abordado.

- Diseño de la solución: en este apartado se argumentan las decisiones tomadas acerca de cómo llevar a cabo las diferentes técnicas usadas para encontrar una solución al problema y los recursos usados para ello.

- Desarrollo de la solución propuesta: se describirán todos los problemas a los que nos hemos enfrentado y las diferentes soluciones propuestas que se han decidido usar durante el desarrollo de la solución.

- Pruebas: se presentarán diferentes pruebas con sus respectivos datos de salida para discutir cuáles son buenas soluciones, cuáles no, y si los diferentes métodos usados valen la pena o no.

- Conclusiones: se hablará sobre los diferentes problemas que se han encontrado y cómo se han afrontado, así como los errores cometidos, qué se ha aprendido y la relación con estudios cursados.

- Trabajos futuros: Se presenta una lista de futuros puntos a explotar y diversas mejoras que puede tener este trabajo.

2. Estado del arte

En este capítulo hablaremos del estado actual de las tecnologías que hemos usado, así como los conceptos principales relativos a ellas. Este capítulo está dividido en tres subapartados. El primero es visión por computador, donde se hablará sobre cómo un computador entiende una imagen y sus principales características. En el segundo subapartado se explicará la base teórica que hay en las diferentes técnicas de aprendizaje automático usadas en este proyecto. Por último, se hablará de trabajos más específicos en la temática de detección de palabras partidas.

2.1. Visión por computador

En este apartado vamos a explicar un punto clave de nuestro proyecto a partir del cual vamos a desarrollar todo, la visión por computador. Esto es necesario para entender el problema al que nos enfrentamos. Lo primero será entender cómo un computador codifica una imagen digital.

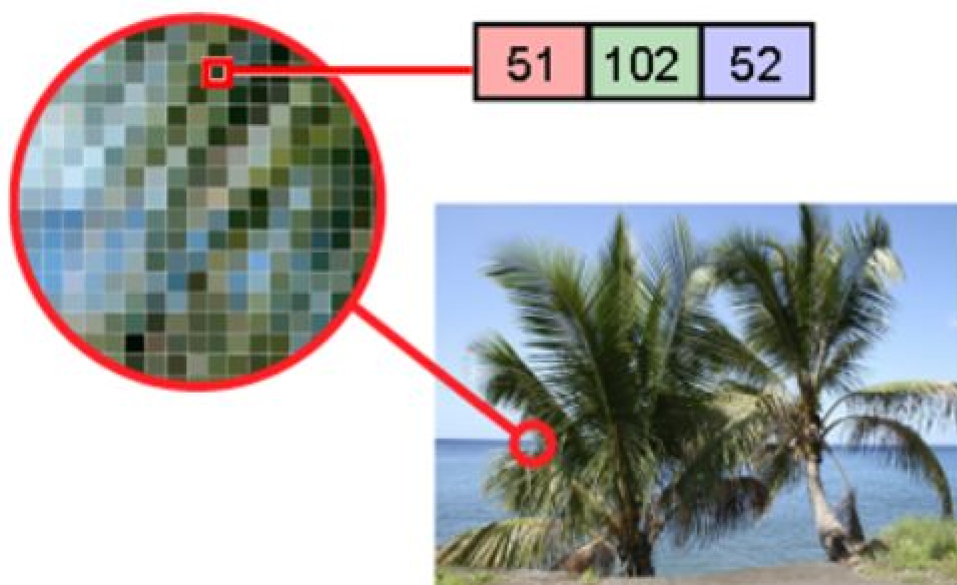


Figura 2. Imagen digital en un computador.

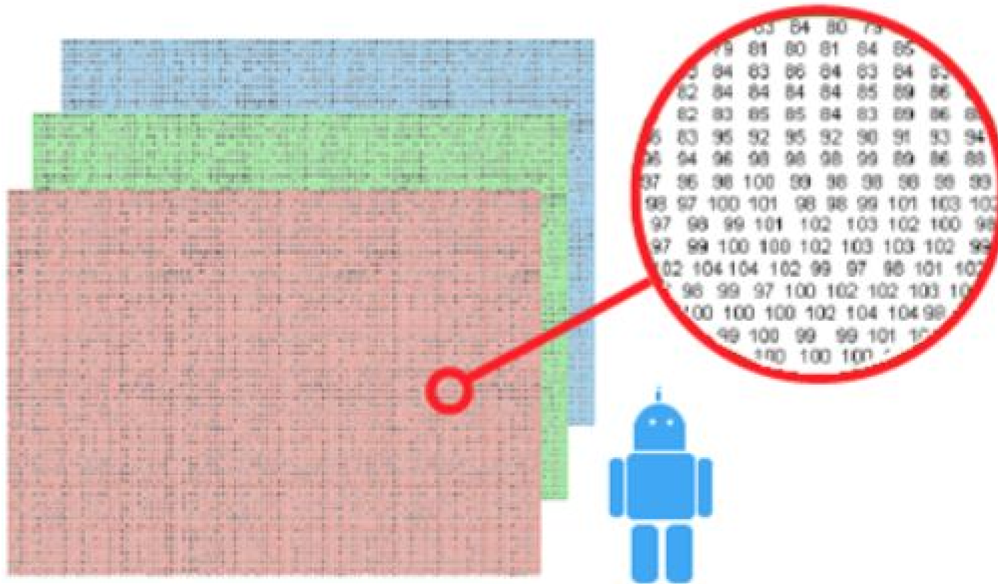


Figura 3. Matriz de números que representa una imagen.

Para una máquina, una imagen es una enorme matriz tridimensional llena de números. Concretamente, cada píxel de la imagen queda representado como una tupla de 3 valores RGB (rojo, verde y azul, esta configuración de color es la más habitual, sin embargo no es la única [4]) del píxel, como se puede ver en la figura 2. En la matriz nombrada anteriormente, el desplazamiento horizontal y vertical corresponde a la posición del píxel en la imagen, mientras que su desplazamiento en la tercera dimensión corresponde al valor de rojo, azul o verde del píxel. En la figura 3 se muestra gráficamente cómo un equipo guarda en memoria la imagen de la figura 2 [5].

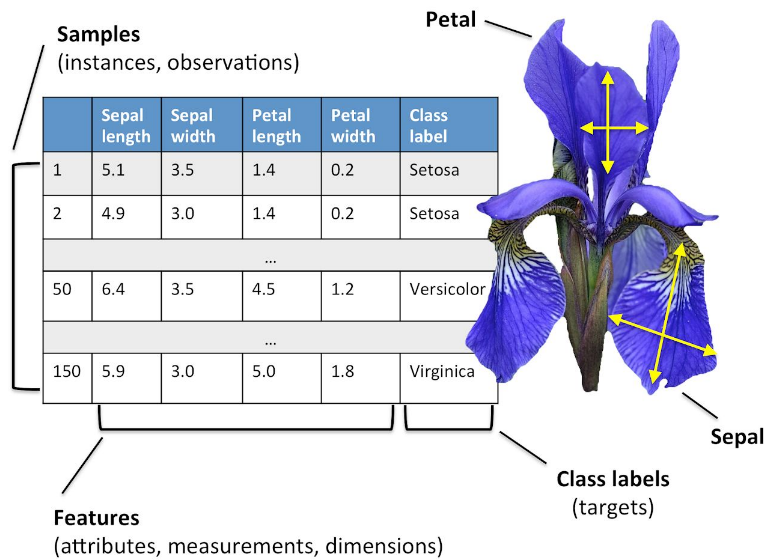
Una vez conocido esto podemos observar que es muy fácil trabajar con estas matrices en cualquier lenguaje de programación. El reto es enseñar a una máquina a reconocer en ellas objetos o formas.

2.2. Técnicas de aprendizaje automático

El aprendizaje automático es una rama de la inteligencia artificial (IA) en la cual se construye una IA que no depende de unas reglas y un programador, sino que ella misma se pone las reglas. Su objetivo es crear un modelo que nos permita

resolver una tarea dada. Una vez tenemos ese modelo, se entrena con una gran cantidad de datos de forma que una vez entrenado es capaz de, para una entrada, predecir una salida. En nuestro caso, por ejemplo, para una imagen de texto manuscrito predecir si existe la representación de una palabra partida en ella o no.

La configuración que hace que nuestro modelo tome mejores o peores decisiones a la hora de clasificar nuestros datos dependerá totalmente de qué atributos decidimos que son relevantes para definir a las muestras y qué algoritmo, o conjunto de algoritmos, usamos. Es decir, no es lo mismo hacer un modelo para la clasificación de flores silvestres usando como atributos el largo y ancho del sépalo y el pétalo de la flor [6], que un modelo para la clasificación de números manuscritos [7] usando el valor de cada píxel en la imagen como atributo (en este caso, usando una imagen de 28x28 pixeles nos daría 784 atributos para cada muestra). Estos diferentes casos se ilustran en las figuras 4 y 5.



	Sepal length	Sepal width	Petal length	Petal width	Class label
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
…					
50	6.4	3.5	4.5	1.2	Versicolor
…					
150	5.9	3.0	5.0	1.8	Virginica

Figura 4. Representación de la base de datos de flores silvestres [6].

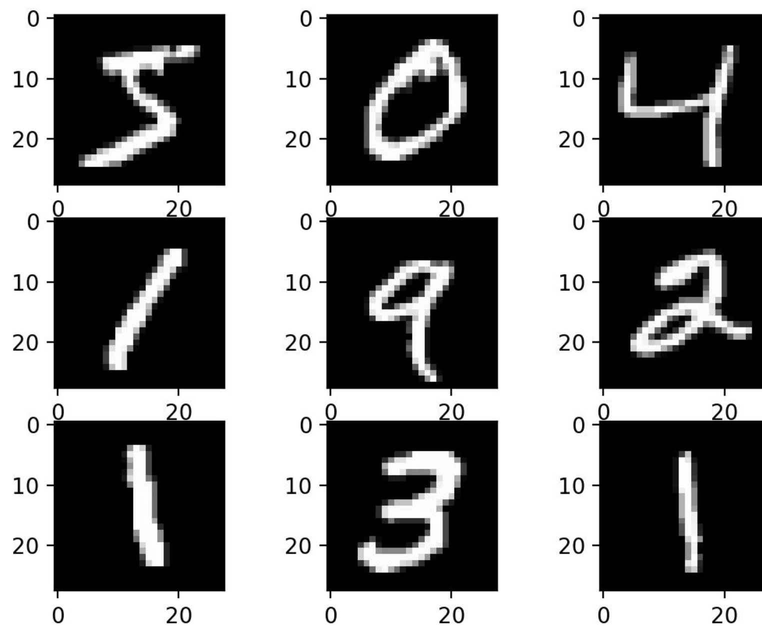


Figura 5. Representación de una datos base de datos de imágenes de números manuscritos. Se traduciría a una matriz de 28x28 donde cada píxel tendría el valor en blanco-negro (255 para el blanco, 0 para el negro) [7].

La elección de estos atributos y/o la naturaleza de ellos puede hacer que los datos de la misma base de datos sean más fácilmente diferenciables de otros. En el caso del ejemplo anterior de la base de datos de flores silvestres, las clases se diferencian más entre sí que los números manuscritos, como se puede ver en las ilustraciones de las figuras 6 y 7.

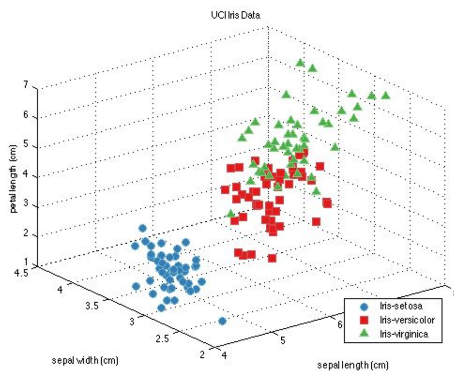


Figura 6. Representación gráfica para flores silvestres.

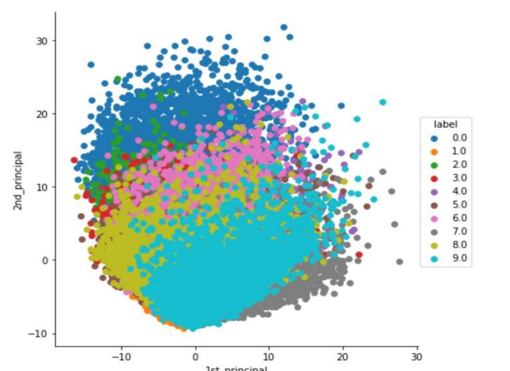


Figura 7. Representación gráfica para números manuscritos.

Una vez elegimos una parametrización de los datos, estos pueden ser separables linealmente o no. Si podemos poner una frontera completamente lineal entre dos clases de una base de datos, como es el caso de la clase iris-setosa frente a iris-versicolor y/o iris-virginica de la figura 6, esto será una frontera de separación lineal. En cambio, si esto no es posible, estaremos frente a dos grupos de datos no separables linealmente, lo cual significa que no existe una separación trivial y tendremos que decidir fronteras de decisión a trozos, con curvas o decidir no separar todos los datos y perder un poco de acierto de clasificación, como es el caso de la figura 7.

Es por esto por lo que la elección del tipo de modelo y del algoritmo para fabricar nuestro modelo final también afectará a la clasificación de los datos, ya que existen tipos de modelos y de algoritmos que funcionan mejor con datos separables linealmente y no tan bien con no separables linealmente y viceversa.

Por ello, lo mejor es hacer un estudio de diferentes modelos, algoritmos y técnicas para, con los datos que tenemos, intentar conseguir el mejor modelo posible que clasifique nuestros datos, ya que en nuestro caso no conocemos la naturaleza de los datos y a priori no sabemos cuál será la mejor elección.

A raíz de una investigación por la red [8][9], hemos visto la gran cantidad de diferentes algoritmos de aprendizaje automático que existen. A fin de acotar la extensión de este trabajo a unos límites razonables se ha decidido probar cuatro de los algoritmos más vistos hoy en dia y que más parámetros se les puede cambiar para poder probar diferentes formas de generar un modelo con el mismo algoritmo.

A grandes rasgos, existen dos grandes grupos de algoritmos de aprendizaje automático, los cuales son los de aprendizaje supervisado y los de aprendizaje no supervisado. La diferencia entre estos dos grupos es que, en la fase de entrenamiento de nuestro modelo para la clasificación, se necesitan muestras etiquetadas para el modelo de aprendizaje supervisado y para el aprendizaje no supervisado no es necesario. Las muestras que se usan como entrenamiento para nuestro modelo serán similares a las que luego intentaremos clasificar con el modelo algorítmico ya finalizado. La gran diferencia es que las muestras para el aprendizaje supervisado llevarán una etiqueta marcando la clase de la que forman parte, y para los no supervisados no llevarán ningún tipo de etiqueta.

Por ejemplo, supongamos que queremos clasificar imágenes donde aparezcan coches de un corpus de imágenes de diferentes vehículos. Se podría crear una estructura de datos donde asociamos a cada imagen un 0 para una imagen donde no aparezca un coche y un 1 para donde aparezca; de este modo, si nuestro modelo en fase de entrenamiento usa una imagen que tiene un 1 asociado, sabe que es un coche y lo tendrá en cuenta a la de buscar rasgos comunes; en cambio, si aparece un 0 sabe que no es un coche, y que las características que aparecen en esta imagen pueden ser un claro indicio de ello.

Si usamos algoritmos no supervisados no existirá este apoyo y, por tanto, tendrá que ser el propio algoritmo el que encuentre patrones en las imágenes sin ninguna clase de ayuda y las etiquete.

Los algoritmos seleccionados a estudiar son: algoritmo k-vecinos cercanos, máquinas de vectores soporte, árboles de decisión y k-medias. Los tres primeros mencionados forman parte del grupo de algoritmos supervisados, y el último es un algoritmo no supervisado. De este modo, podremos hacernos una idea de qué estrategia será mejor abordar.

2.2.1. Algoritmos supervisados

k-vecinos más cercanos

El algoritmo por k-vecinos más cercanos [10] clasifica cada dato nuevo en el grupo que corresponda según tenga k vecinos más cerca de un grupo u otro. Para ello se basa en el concepto de distancia, que determina la cercanía del dato a clasificar respecto a cada prototipo. Una vez calculadas las distancias a los prototipos, se toman aquellos k prototipos a menor distancia del dato a clasificar y se clasifica como el grupo más abundante entre ese conjunto de k prototipos.

Es un algoritmo con fuertes ventajas y desventajas. Una de las ventajas que nos ha hecho plantearnos su uso es que funciona muy bien con muestras con muchas características; teniendo en cuenta que tenemos que buscar formas en una imagen, las características serán muchas y, por tanto, hará conveniente este algoritmo para este problema.

Para presentar su implementación, a continuación mostramos matemáticamente la notación utilizada.

		X_1	X_j	X_n	C
(x_1,c_1)	1	x_{11}	x_{1j}	x_{1n}	c_1
(x_i,c_i)	i	x_{i1}	x_{ij}	x_{in}	c_i
(x_N,c_N)	N	x_{N1}	x_{Nj}	x_{Nn}	c_N
x	N + 1	$x_{N+1,1}$	$x_{N+1,j}$	$x_{N+1,n}$	?

Figura 8. Notación para el algoritmo k-vecinos más cercanos.

En la figura 8 se muestra cómo con un grupo de datos D (conjunto de prototipos) formado de N datos, los cuales son una dupla de x_i (muestra) etiquetada en c_i (grupo), cada x_i muestra tiene n características. Una vez tenemos esto, para clasificar un dato x_{N+1}, con sus $x_{N+1,1}$ … $x_{N+1,n}$ características, podremos predecir la pertenencia a un grupo C.

En la figura 9 se muestra un pseudocódigo para el clasificador. Tal y como se observa, se calculan las distancias de todos los casos ya etiquetados (prototipos) al nuevo caso x que se pretende clasificar. Una vez ordenadas las distancias, se seleccionarán los K casos más cercanos, y entre ellos se le asignará a x la clase más frecuente de entre estos.

COMIENZO

Entrada: $D = \{(x_1,c_1), \ldots , (x_N,c_N)\}$

$x = (x_1, \ldots , x_n)$ nuevo caso a clasificar

PARA todo objeto ya clasificado (x_i , c_i)

calcular $d_i = d(x_i , x)$

Ordenar $d_i (i = 1, \ldots , N)$ en orden ascendente

Quedarnos con los K casos D^K_x ya clasificados más cercanos a x

Asignar a x la clase más frecuente en D^K_x

FIN

Figura 9. Pseudocódigo algoritmo k-vecinos cercanos.

En la figura 10 vamos a encontrarnos un ejemplo de la aplicación de este algoritmo gráficamente. En este ejemplo tenemos un grupo de datos D que contiene 21 elementos; todos los elementos tendrán dos características (X_1 y X_2 en dicha figura); existen dos clases a clasificar y el número de vecinos a considerar es de tres.

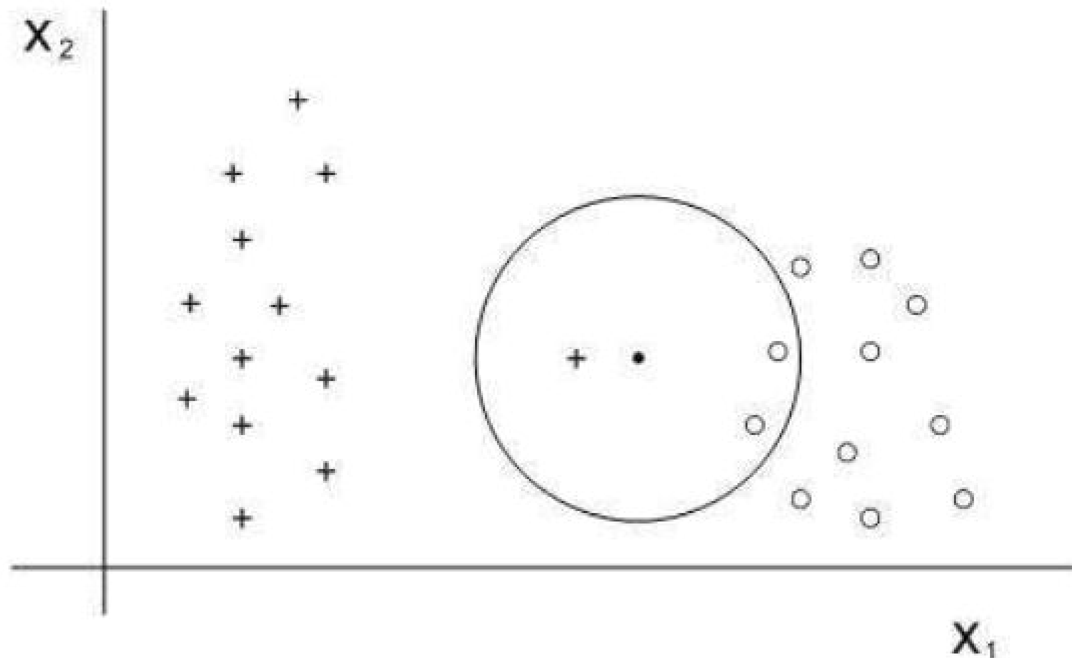


Figura 10. Ejemplo de aplicación del algoritmo k-vecinos cercanos.

Como se puede ver, existen dos grupos a clasificar, las cruces y los círculos; en este caso, al ser dos de los tres vecinos más cercanos de la clase círculo, nuestra muestra a clasificar (punto negro) se clasificará como círculo.

Existen varias versiones del algoritmo k-vecinos más cercanos. Vamos a comentar las que se tendrán en cuenta para el estudio de este algoritmo.

k-vecinos más cercanos con distancia media

En esta variación se realizará la distancia media de nuestra muestra a las clases de los k-vecinos más cercanos; es decir, si nuestro número de vecinos más cercanos a considerar es 5 y tenemos 3 vecinos clasificados como círculo y 2 vecinos clasificados como cruz, se realizará la media de las distancias de cada uno a la

muestra a clasificar entre los vecinos del mismo grupo, y la menor resultante será la que nos dará la etiqueta de nuestra muestra.

k-vecinos más cercanos con pesado de casos seleccionados

Se le asignará un peso a cada muestra dentro de los k-vecinos más cercanos para que no todos los vecinos se contabilicen de igual forma. Por ejemplo, se puede usar como pesado seleccionando de manera inversamente proporcional a la distancia del vecino a la muestra a clasificar.

Máquinas de vectores soporte

Las máquinas de vector soporte (conocidas mundialmente por las siglas SVM) [11] construyen un modelo separando las clases a los espacios más amplios posibles mediante un hiperplano. Las muestras más cercanas a la frontera de decisión forman los vectores soporte, que son los que han definido esta frontera de decisión; por eso a veces también se conoce a esta técnica como clasificador de margen máximo. Vamos a ver un ejemplo con dos clases para clasificar en un ejemplo con dos características por muestra.

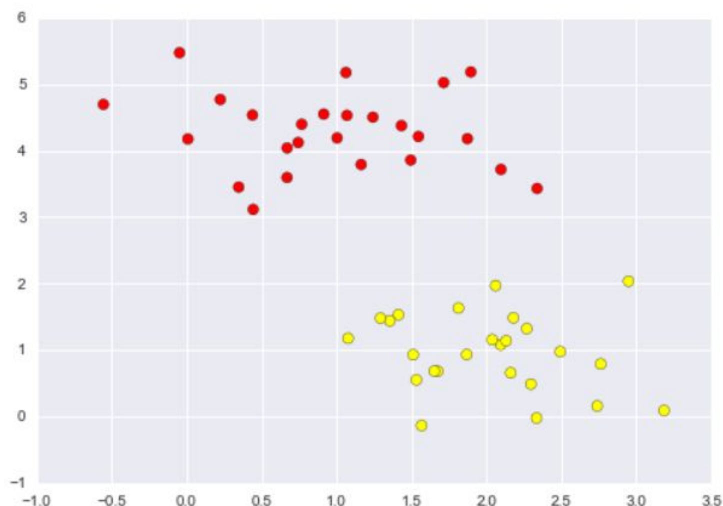


Figura 11. Dos grupos de datos a clasificar (amarillos y rojos).

En el ejemplo a clasificar mostrado en la figura 11, como estamos clasificando en dos dimensiones (dos características relevantes), el hiperplano que se

está buscando es una recta que los separe. En este caso se puede ver intuitivamente dónde trazar la frontera de decisión para clasificar un nuevo punto como rojo o como amarillo. No obstante, hay infinitas maneras de poner dicha frontera de decisión, como se puede apreciar en la figura 12. Este método resuelve el problema de encontrar la frontera de decisión óptima.

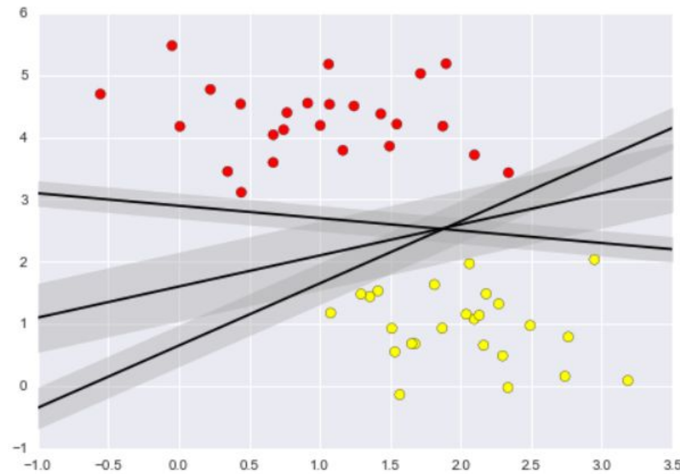


Figura 12. Posibles fronteras con sus márgenes.

El algoritmo buscará la maximización de márgenes entre la frontera de decisión y los grupos. Al hacer esto, habrá algunos puntos que se posicionarán sobre los márgenes; éstos serán los que nos marcan las fronteras de decisión y son los que forman los vectores soporte.

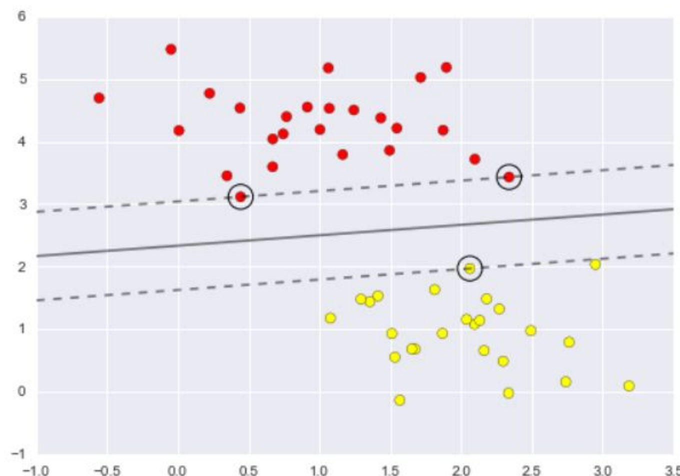


Figura 13. Frontera óptima con márgenes.

Una vez nuestro algoritmo converge obtenemos la frontera de decisión óptima, la cual tiene los márgenes más amplios posibles en cada clase. En la figura 13

aparecen los puntos que forman los vectores soporte (dentro de un círculo negro); estos puntos son los más cercanos a la frontera de decisión y, por tanto, la definen.

Este ejemplo es un ejemplo muy sencillo para entender cómo funciona este algoritmo. No obstante, no suele ser aplicable a la vida real, ya que lo común es que las muestras tengan más de dos características y no suelen ser separables linealmente, como puede ser el caso de los datos presentados en la figura 14.

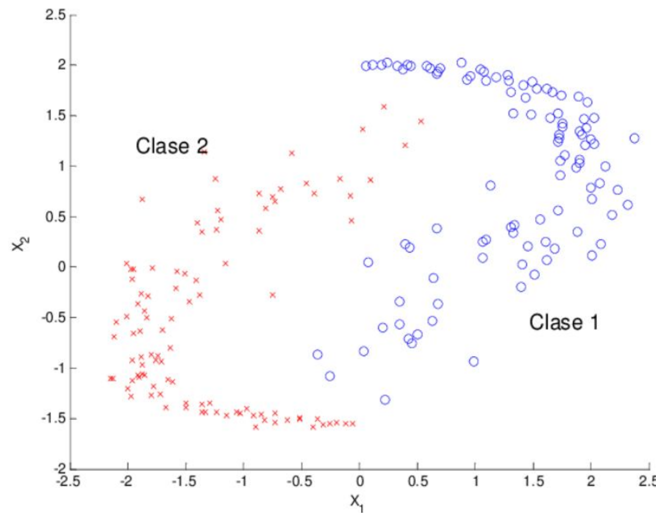


Figura 14. Ejemplo de datos no separables linealmente.

Para ayudar a nuestro clasificador cuando nos encontramos con un problema de no separación lineal y/o con muchas dimensiones, se puede usar una técnica que suele ir de la mano con la aplicación SVM, y es la aplicación de un kernel [12].

Un kernel no es más que una función matemática aplicada a cada muestra de nuestra base de datos con la idea de cambiarla de dimensión y/o posición en nuestro espacio.

Existen muchísimos tipos de funciones kernel; no obstante, como el algoritmo SVM no es el único que se va a usar en este proyecto, hemos decidido no ahondar y usar los más comunes. A continuación se presentan los tipos de kernel que hemos decidido usar: función polinomial, función de base radial Gaussiana y función sigmoid.

Polinomial

Se formula como:

$$K(x_i, x_j) = (x_i \cdot x_j)^n$$

Donde x_i y x_j son los valores de las características de la muestra y la n es el grado polinomial a seleccionar. Tiene efectos similares a los presentados en la figura 15.

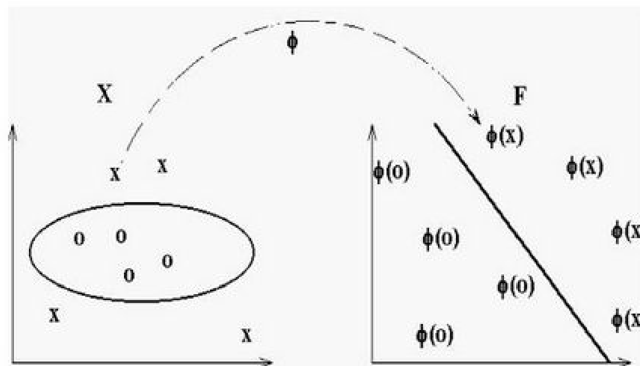


Figura 15. Conversión kernel polinomial ϕ.

Base radial Gaussiana

Este kernel responde a la fórmula:

$$K(x_i, x_j)=\exp(-(x_i-x_j)^2/2(\text{sigma})^2)$$

Donde x_i y x_j son los valores de las características de la muestra y sigma es un valor a elegir. La figura 16 muestra un ejemplo de su comportamiento.

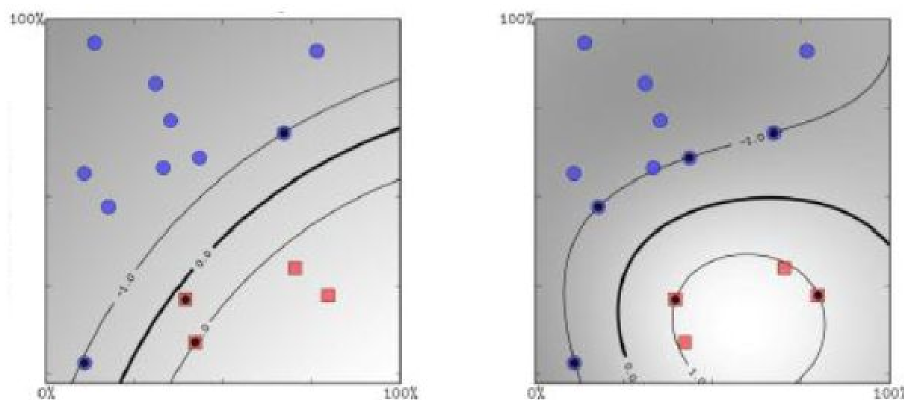


Figura 16. Conversión Gaussiana con valor de sigma 20 (izquierda) y 1 (derecha).

Sigmoid

Este kernel es equivalente a una función de activación para neuronas artificiales dentro de una red neuronal y se define por:

$$K(x_i, x_j)=\tanh(x_i \cdot x_j - \theta)$$

donde x_i y x_j son los valores de las características de la muestra y θ es un parámetro a seleccionar.

Árboles de decisión

En los árboles de decisión [13], a partir de un conjunto de datos se fabrica un diagrama que sirve para representar y categorizar una serie de condiciones que ocurren de forma sucesiva, resolviendo un problema.

Los árboles están formados por nodos, los cuales se definen como el momento en el que se ha de tomar una decisión entre varias posibles, flechas, que representan las uniones entre diferentes decisiones (uniones entre nodos), y etiquetas, las cuales dan nombre a las acciones y a las decisiones.

El clasificador comienza desde el nodo inicial, al cual no llega ninguna flecha, y a partir de éste se empieza a tomar decisiones hasta que se llega a un nodo hoja, también conocido como nodo final. Aquí se tomará la decisión de etiquetado para la muestra a clasificar. Como para construir el árbol tenemos muestras etiquetadas, nuestro algoritmo decidirá qué características son más determinantes que otras, haciendo que decidan en un nodo más alto o más bajo en el árbol. La posición de un nodo en la toma de decisión influye mucho, ya que si está muy alto su decisión hace que una gran cantidad del árbol pase a ser inaccesible, mientras que si está más bajo la cantidad del árbol que pasa a ser inaccesible es menor.

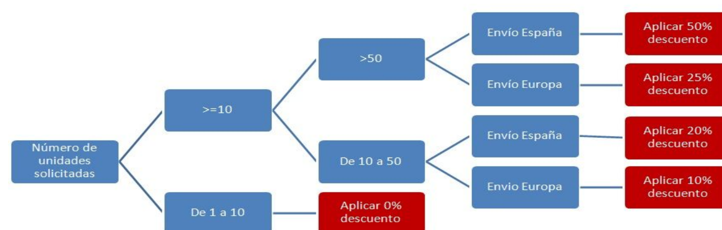


Figura 17. Ejemplo de árbol de decisión.

En el ejemplo mostrado en la figura 17, el nodo que contiene la decisión de número de unidades solicitadas es mucho más decisorio que la cantidad de unidades solicitadas, ya que si dicho número es menor que 10 casi todo el árbol se convierte en inaccesible; en cambio, si es de 10 a 50, por ejemplo, sólo 3 nodos quedarán inaccesibles.

Al ser éste un algoritmo supervisado, usaremos una parte de nuestros datos para crear el árbol de decisión. Una vez este árbol está creado, cada muestra a clasificar la colocamos en el nodo inicial y la dejamos caer por él hasta que llega a un nodo hoja; cuando ha llegado a este nodo, esa será su etiqueta de clasificación. Por ejemplo, supongamos que queremos saber si administrar un fármaco X a un paciente; para ello tenemos una base de datos de pacientes a los cuales se les ha administrado anteriormente dicho fármaco y sus características, las cuales son las presentadas en la figura 18.

Característica	Presión arterial	Azúcar en sangre	Índice de colesterol	Alergia a antibióticos	Otras alergias
Valor	Alta, Media, Baja	Alto, Bajo	Alto, Bajo	Si, No	Si, No

Figura 18. Ejemplo características de la base de datos de pacientes.

Dichos valores de las características son meramente orientativos, ya que en una aplicación real deberían de ser codificados para optimizar su funcionalidad (por ejemplo, alto, medio y bajo se podría representar como 2 para alto, 1 para medio y 0 para bajo), pero esto ya forma parte de la implementación.

En este ejemplo existirán dos grupos de clasificación, sí y no, el cual tomaría la decisión de administrar el fármaco a un paciente o no. Un ejemplo de árbol generado al crear el modelo con nuestra base de datos es el mostrado en la figura 19.

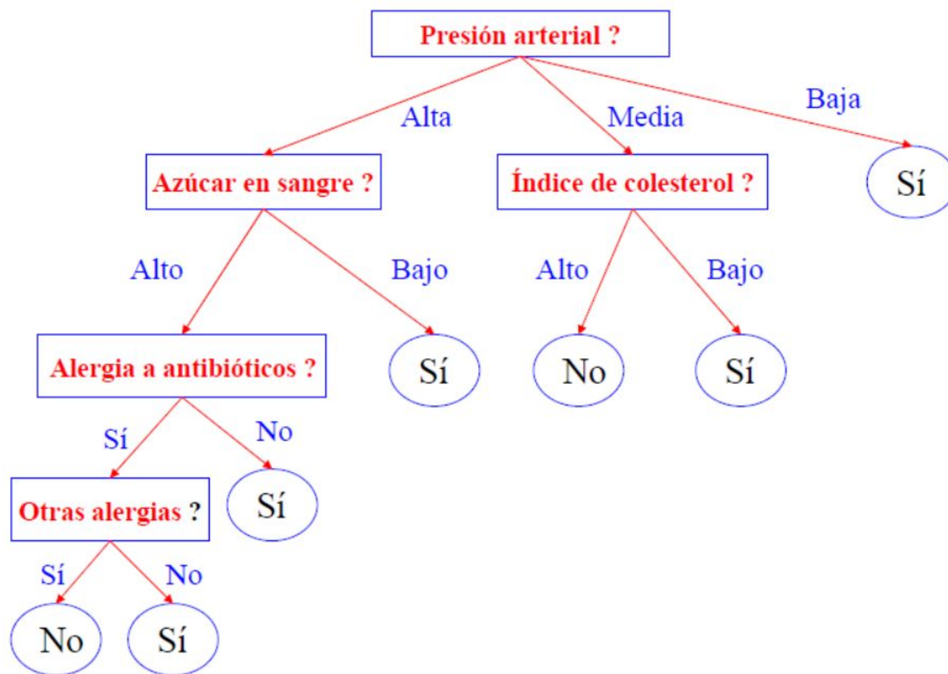


Figura 19. Ejemplo de generación de árbol de decisión con una base de datos estructurada como en la figura 18.

En este ejemplo, al tener pocas características, el árbol de decisión tiene pocos niveles hasta llegar a un nodo hoja, y es poco ancho ya que cada característica tiene pocos valores a tomar. Sin embargo, esto puede crecer muy rápidamente si nuestras muestras tienen muchas características y/o muchos valores a tomar por característica.

Al ser este árbol de la figura 19 bastante corto, podemos ver un ejemplo de cómo una vez creado el árbol podemos clasificar nuevas muestras teniendo sus características. Por ejemplo, vamos a clasificar las dos muestras presentadas por la figura 20.

Pacientes	Presión arterial	Azúcar en sangre	Índice de colesterol	Alergia a antibióticos	Otras alergias
1	Baja	Alta	Alta	No	Sí
2	Alto	Alto	Alto	Sí	Sí

Figura 20. Posibles muestras de la base de datos de pacientes, los cuales están por decidir si suministrarle el fármaco o no.

Para empezar, colocamos ambos pacientes en el nodo principal, y ese nodo decide una acción o otra dependiendo de la decisión. En este caso, el primer

nodo mira la característica de presión arterial; el paciente 1 la tiene baja, por lo que cae hacia el nodo más hacia la derecha por la flecha con la etiqueta 'baja' y llega a un nodo hoja, el cual es un 'Sí' al tomar el fármaco, por lo que al paciente 1 se le suministrará dicho fármaco. El paciente 2 caerá por la flecha con la etiqueta 'alta', entrando en otro nodo con variable decisión; en este nodo se mirará la característica azúcar en sangre; al tener este paciente el azúcar en sangre alta, caerá por la flecha con la etiqueta correspondiente y llega a un tercer nodo que mira la característica alergia a antibióticos; al tener éste alergia, llega a un cuarto nivel donde se mira la característica otras alergias, y al tener en ésta un sí, llegamos a un nodo hoja, el cual clasifica el paciente en no administrar fármaco X. Como se puede ver, en este árbol la característica más importante es la presión arterial, ya que hace que la clasificación de un nuevo paciente pueda acabar muy rápido (caso paciente 1) o llegar hasta el final del árbol (caso paciente 2).

2.2.2. Algoritmos no supervisados

k-medias

El algoritmo k-medias [14] es uno de los algoritmos de aprendizaje no supervisado más simples que resuelven el problema de la clusterización, esto es, formar grupos de muestras sólo sabiendo sus características, de forma que al colocar una nueva muestra en el espacio sepamos cuál es su grupo de pertenencia y clasificarla.

El algoritmo primero sitúa las muestras en el espacio; lo siguiente que hace es colocar K puntos (tantos como agrupamientos a priori le hayamos dicho que agrupe) en el espacio; estos puntos representarán los centroides. Tras ello, a cada muestra en el espacio se le asigna el centroide más cercano; tras haber asignado todas las muestras se recalculan las posiciones de los K centroides. Se repiten estos dos últimos pasos hasta que los centroides se mantengan estables. En el pseudocódigo de la figura 21 se puede ver esta explicación claramente. En la figura 22 se hace una representación gráfica de un ejemplo.

COMIENZO

Entrada: $D = \{x_1, \ldots, x_n\}$, $K = \{c_1, \ldots, c_k\}$ donde k es la cantidad de centroides iniciales

MIENTRAS no convergencia

PARA todo objeto x_i

se le asigna el centroide más cercano c_i

Recalculamos nuevos centroides

FIN

Figura 21. Pseudocódigo algoritmo k-medias.

En el ejemplo gráfico de la figura 23 partimos de la distribución de datos de la figura 22, y decidimos darle como argumento al algoritmo k-medias un tres para que busque tres agrupamientos de datos.

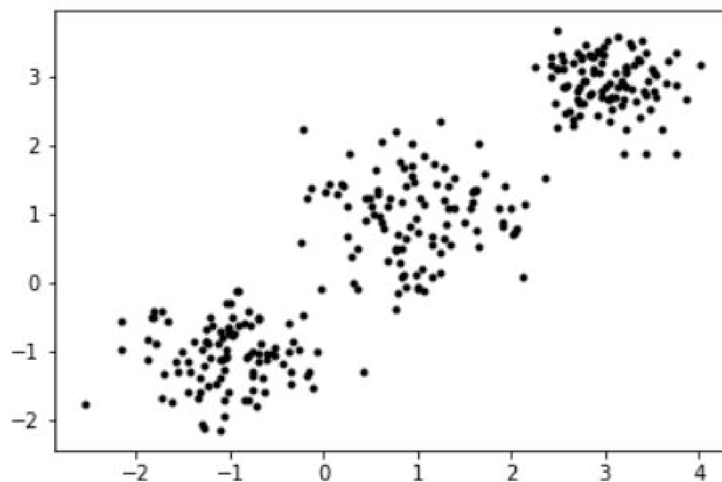


Figura 22. Conjunto de datos ejemplo para el algoritmo k-medias

Lo primero que haremos será colocar k centroides (3) en nuestro espacio. Una vez están colocados, empezaremos a asignar a las muestras los centroides más cercanos y continuaremos una y otra vez haciendo lo mismo hasta que los centroides prácticamente no cambien de posición. En la figura 23 podemos ver cómo partiendo de la figura 22 y habiendo añadido en ella los 3 centroides, acabamos con una agrupación de *clusters* donde se diferencia a la perfección qué punto forma parte de qué agrupación.

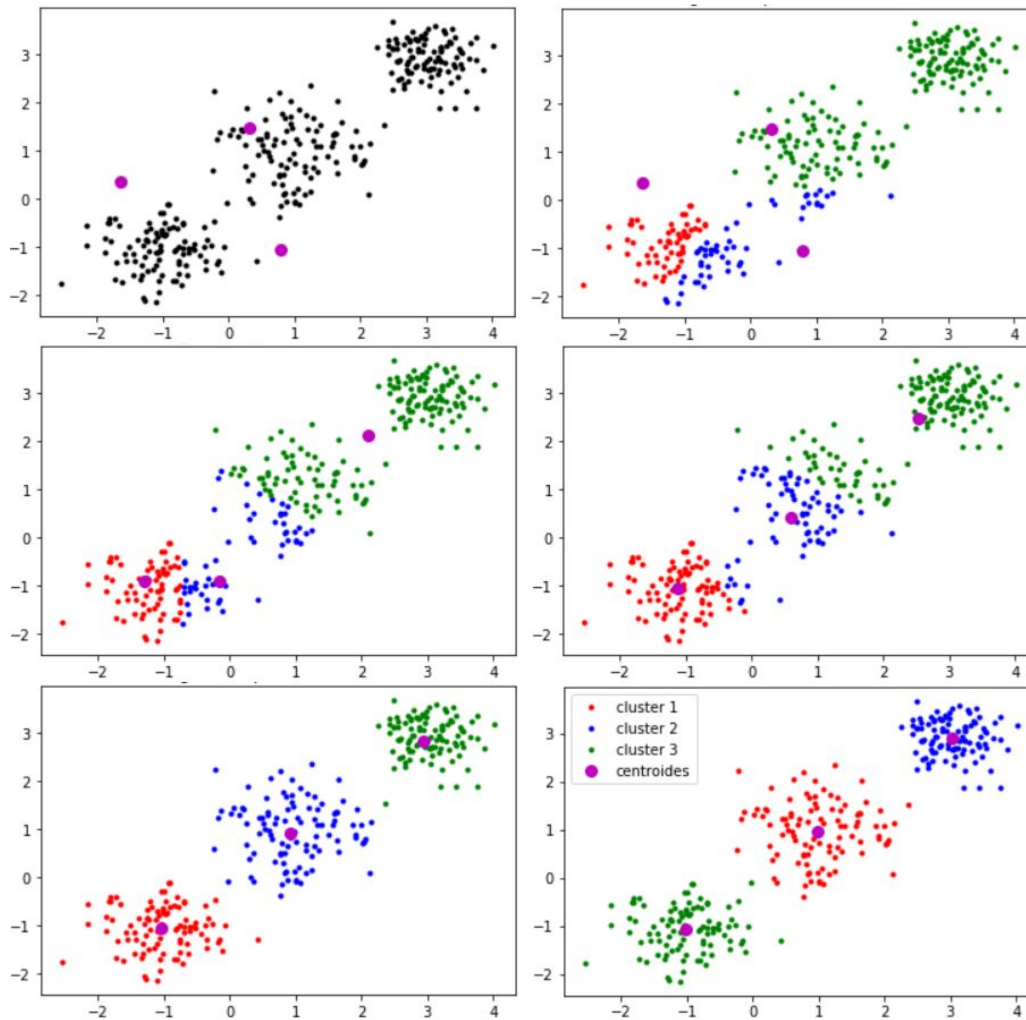


Figura 23. Representación de las iteraciones que realiza el algoritmo con tres centroides para hallar los agrupamientos de datos (de izquierda a derecha de arriba a abajo).

Este algoritmo tiene desventajas que iremos detallando en la implementación del problema. Una de ellas es el no saber qué cantidad de agrupaciones nos vamos a encontrar; teniendo en cuenta que es un parámetro necesario para el lanzamiento del algoritmo, haremos diversas pruebas. Otro problema al que nos enfrentamos es que si nuestras muestras están muy homogéneas en el espacio, el algoritmo nos devolverá tantas agrupaciones como hayamos indicado, ya que este problema es un problema de optimización. Por tanto, aunque el algoritmo haya convergido satisfactoriamente, ello no implica conseguir un buen resultado.

2.3. Proyectos similares

En este apartado se tratará de situar este trabajo entre diversos trabajos realizados en los últimos años. Buscar palabras partidas en imágenes, como hemos visto, implica usar visión artificial para encontrar las formas en el texto y aprendizaje automático para tratar de saber cuando una palabra está partida en el texto y cuando no.

Una de las líneas de investigación que existe actualmente es la localización de palabras en un texto escaneado [15], [16]. Este trabajo implica el reconocimiento de formas para detectar palabras y así poder localizar palabras clave en textos. En estas tareas podemos encontrar ideas para usar diferentes técnicas para el reconocimiento de las formas que nosotros buscamos. Esta línea de estudio se dedica a encontrar palabras y/o transcribirlas a digital, ya que una imagen digital no es lo mismo que un texto digital (imaginemos el primero como un archivo de imagen y el segundo como un pdf; evidentemente, no tienen las mismas herramientas).

Nuestro trabajo no busca palabras como tal, ya que lo que nosotros buscamos es el indicio de si una palabra está partida o no para su posterior clasificación. No obstante, estos trabajos nos darán una línea de investigación interesante.

3. Análisis del problema

Una vez visto el estado del arte, ya hemos visto qué técnicas pueden ser usadas en la resolución de nuestro problema. Dichas técnicas necesitan muestras etiquetadas y no etiquetadas para poder trabajar; sin embargo, nosotros no tenemos una base de datos con muestras normalizadas y etiquetadas, únicamente disponemos de las páginas manuscritas escaneadas a color del libro 'Noticia histórica de las fiestas con que Valencia celebró el siglo sexto de la venida a esta capital de la milagrosa imagen del salvador por D.Vicente Boix, cronista de la misma ciudad, año 1853', el cual está formado por 53 páginas. Para ver una agrupación de diversas páginas ejemplo se puede consultar al apéndice I.

Así que de momento tenemos una base de datos por pulir para poder trabajar con ella, además de la decisión de qué técnicas algorítmicas vamos a usar para resolver el problema. En los siguientes subpuntos entraremos en profundidad sobre cómo plantear la resolución del problema y qué posibles dificultades nos podemos encontrar.

3.1. Identificación y análisis de soluciones posibles

Como el procesado de los datos se va a realizar con diferentes pruebas de los cuatro algoritmos que se han decidido usar y, tal como se ha indicado anteriormente, no conocemos la naturaleza de los datos, no podemos saber cuál va a funcionar mejor o peor. Por ello en este apartado analizaremos los preprocesos y postprocesos que se pueden hacer a nuestros datos de entrada.

Al ser este problema un problema de reconocimiento de patrones en imágenes, nos va a ser muchísimo más fácil abordar este problema en un menor tamaño, es decir, un buen preproceso podría ser convertir la imagen en trozos más pequeños para intentar localizar las palabras partidas mejor. Por ejemplo, si partimos cada imagen en 4, cuando encontremos una palabra partida no sabremos exactamente dónde se encuentra; en cambio si hacemos trozos más pequeños tendríamos una precisión mayor (tal y como se muestra en la figura 24), aunque

también hay que tener en cuenta con este preproceso que, cuanto más pequeñas sean las muestras, más tiempo de procesamiento nos llevará construirlas y entrenar los modelos.

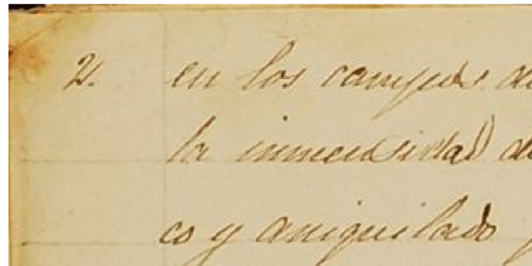


Figura 24. Posibles trozos de la imagen 5 de la base de datos, arriba en partes grandes, abajo, misma parte de la imagen en partes mucho más pequeñas.

Esta partición puede ser una ayuda a la hora de encontrar patrones, ya que los patrones no están a nivel de hoja, sino a nivel de palabra, y unas imágenes más pequeñas recogen mejor la palabra que una imagen más global del texto. Teniendo en cuenta que lo que buscamos son palabras y ésta es una unidad que puede tener un tamaño muy variable, parece razonable el hacer que estas ventanas se solapen. Este preproceso puede ser una idea para hacer las cosas más fáciles. No obstante, aún se pueden hacer muchas más cosas. Algo necesario será normalizar todas las muestras a un tamaño estándar, ya que todas las muestras tienen que tener el mismo número de características; por ello, podemos hacer que las imágenes de las páginas se partan en cantidades de píxeles fijos, y las que se quedan más pequeñas rellenarlas con vacío hasta hacer el tamaño fijado.

Otra cosa que podemos tener en cuenta en el preproceso es que, al querer reconocer formas, lo único que nos interesa es la forma del texto en la imagen. Es decir, su color es una dimensión que no nos interesa, por lo que a nuestro preproceso

podemos añadirle la conversión de cada muestra a una escala de grises, para más facilidad en su proceso.

Como a priori no conocemos la posición de las palabras, ya que el tamaño de una palabra manuscrita es muy variable, habría que tener en cuenta un método muy usado en este tipo de problemas: el solape de las ventanas en menos de un 50%. La figura 25 muestra un ejemplo de extracción de ventanas con un solape del 20%.

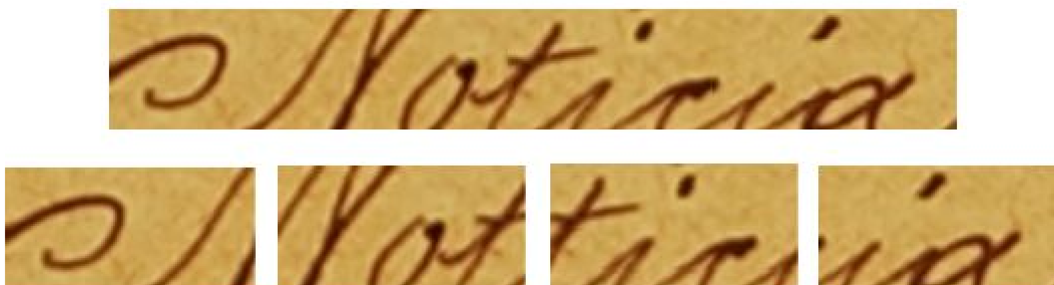


Figura 25. Imagen separada con solapamiento del 20%.

Otro detalle para el preproceso antes de ponernos a diseñar es tener en cuenta las características que vamos a usar para el entrenamiento. Como en todos los problemas de reconocimiento de formas en imágenes, nuestro aliado será la imagen digital, la cual es una matriz donde en cada celda tenemos el valor del color del píxel; en este caso, según el posible preproceso que estamos detallando, en escala de grises. Un detalle muy importante a tener en cuenta en nuestro problema es que estamos buscando algo que ya conocemos a priori dónde más o menos va a aparecer, y su posición es al principio y/o al final de una línea de un texto. Por tanto, ya que vamos a hacer un preproceso donde se tomarán imágenes más pequeñas de una imagen mayor, podemos usar como característica el desplazamiento de ésta de izquierda a derecha y de arriba a abajo (es decir, su posición horizontal y vertical en la página). La ventaja que puede suponer es que nuestros modelos aprenderán que hay unas características que le están marcando exactamente donde suelen estar las palabras partidas. Si añadimos esta información para el reconocimiento de formas podemos tener un preproceso que facilite el trabajo a nuestro modelo de clasificación.

Teniendo en cuenta todos los preprocesos que se pueden aplicar a nuestros datos para una posible solución, podemos usar dicha información para aplicar un buen postpoceso con la información que se nos da de vuelta una vez entrenado y testado el modelo. Un postprocesado a tener en cuenta puede ser

cambiar etiquetas de muestras donde, según nuestro clasificador, no aparece una palabra partida, pero sí existe una palabra partida muy cerca. Por ejemplo, en la figura 25, en las cuatro ventanas que forman la imagen, imaginemos que antes del postproceso la tercera imagen por la izquierda está clasificada como palabra partida y la cuarta no. Si estos fragmentos se localizan al final de línea, esto nos indica que algo está mal clasificado, ya que o ambas son palabras partidas, o ninguna de ellas lo es. Por tanto en nuestro postproceso podemos cambiar la etiqueta de una de las dos y volver a testar. Si incrementamos la tasa de acierto significa que este cambio ha sido correcto. Si se incrementara la tasa de acierto en el global del conjunto de prueba, este postproceso se incorporaría de manera definitiva al sistema de reconocimiento.

3.2. Solución propuesta

En vista de lo expuesto en la identificación y el análisis del problema nuestra solución se dividirá en tres subproblemas.

En el primer subproblema haremos el preproceso de nuestra base de datos. Lo primero que haremos será etiquetar las palabras partidas que aparecen en el texto; una vez tenemos esto, generamos las ventanas solapadas de las imágenes en escala de grises, las cuales serán guardadas en una estructura de datos como matrices y con su etiqueta de clase.

El segundo subproblema será el conjunto de algoritmos a estudiar, los cuales recibirán como entrada la estructura de datos diseñada en el primer subproblema y devolverán las tasas de acierto (precisión de la resolución del problema) y el nuevo etiquetado.

Por último, en el tercer subproblema postprocesaremos la información con la técnica expuesta en el apartado 3.1, es decir, reetiquetando las muestras en función de la etiqueta de sus vecinas y calculando las nuevas medidas cuantitativas de calidad del clasificador y el nuevo etiquetado.

4. Diseño de la solución

En este punto vamos a dejar claro cómo pretendemos desarrollar la solución y qué vamos a usar para ello. No obstante, todo lo plasmado en este punto es una directriz para procesar y encontrar la solución a este problema, ya que mientras se implementa pueden surgir problemas que nos hagan cambiar alguna idea para aproximarnos más a nuestro objetivo.

4.1. Arquitectura del sistema

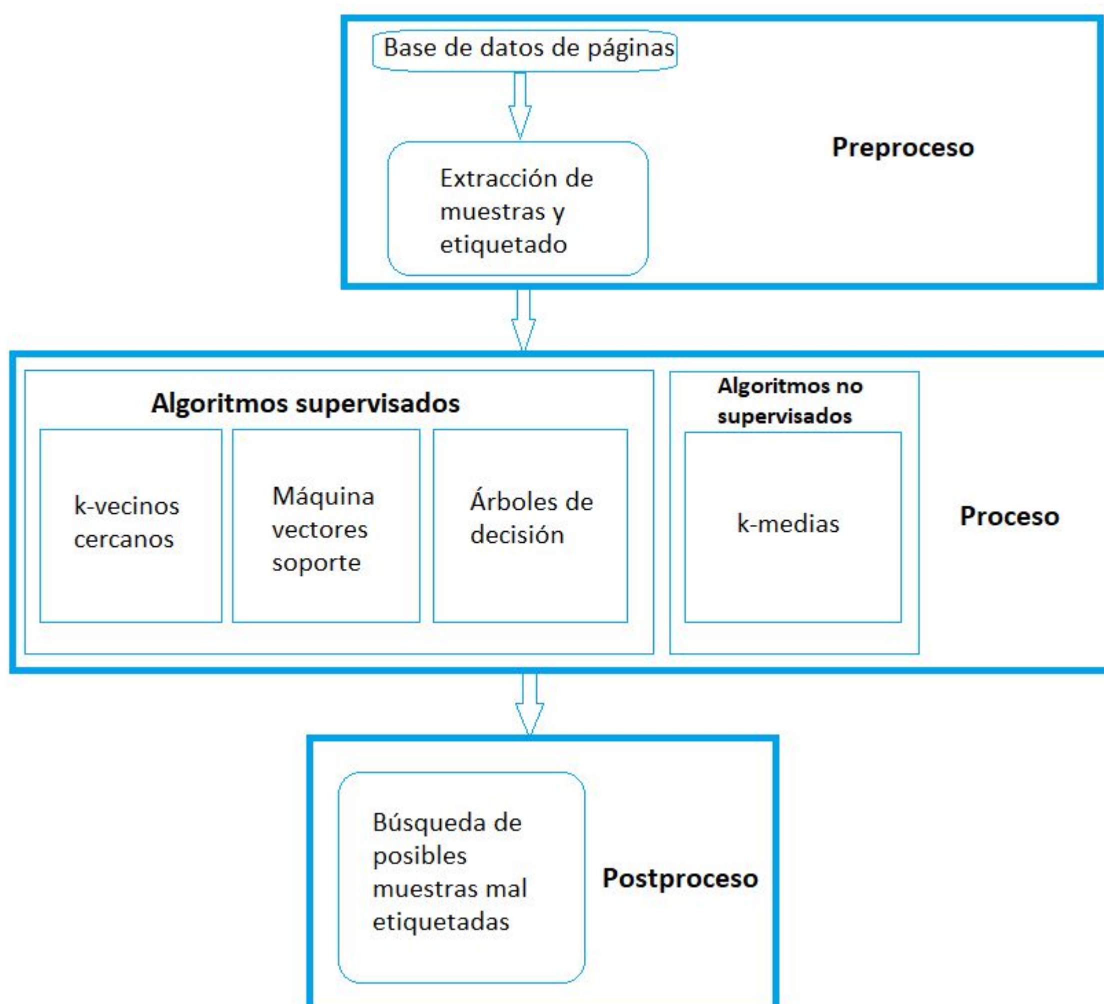


Figura 26. Diagrama de conexión de las partes del desarrollo del problema.

La arquitectura del sistema está dividida en tres grandes bloques, donde cada uno depende de los resultados del anterior para llevar a cabo su función. En la figura 26 se encuentra un esquema de dichos bloques, marcados como rectángulos azules; en el interior de cada bloque se encuentran las principales actividades que resolverá cada subproceso, pero eso se explicará con detalle en el apartado 4.2. En este punto daremos unas pinceladas sobre lo que se llevará a cabo en cada proceso.

En el primer proceso, también llamado preproceso en la figura 26, el objetivo principal es, con la base de datos de entrada formada por imágenes (las cuales son escaneos a color de páginas completas de nuestro libro manuscrito) generar una base de datos de muestras. Estas muestras son ventanas de las imágenes, etiquetadas, en escala de grises y normalizadas, y se guardará en un objeto en el disco para que pueda ser cargado por otros *scripts*.

En el segundo proceso cargaremos el objeto generado por el preproceso y lo usaremos para generar modelos predictivos con diferentes técnicas de *machine learning*, generando resultados.

En el tercer proceso, conocido como postproceso, recogeremos los resultados producidos por el proceso anterior y trataremos de mejorar dichos resultados con varios métodos.

4.2. Diseño detallado

En este punto hablaremos de los *scripts*[1] que hemos desarrollado y cuál es su funcionalidad, así como los diferentes objetos y archivos de texto necesarios para este proyecto.

4.2.1. Preproceso

En esta primera parte del proyecto nos encontramos con los *scripts* `Labeling.py` y `GrayScaleToBN.py`. El primer *script* será el que lleve a cabo la partición de las imágenes de nuestra base de datos a ventanas que recorren dicha imagen, su etiquetado (0 para la no aparición de palabra partida, 1 para lo contrario), la conversión a matriz de números y su posterior guardado a objeto en disco.

[1] https://github.com/Vicentaco-inf/TFG

Este archivo guardará dos objetos en disco, BD_50x100 y Names_X_XLab_NumPxls_50x100. El primer objeto es un diccionario de python donde la clave del diccionario es el nombre de la ventana, el cual es, por ejemplo 05_10_12.png; éste nos está indicando la imagen de pertenencia de la ventana (05), el desplazamiento hacia la derecha en la imagen (10) y el desplazamiento hacia abajo en la imagen (12). Este renombrado de la ventana también se hará en Labeling.py. Como valor de la pareja clave-valor del diccionario tendremos una lista de elementos: el primer elemento que encontramos será la imagen digital de la ventana como una matriz de valores y el segundo elemento será su etiqueta.

El segundo objeto generado, Names_X_XLab_NumPxls_50x100, está formado por una lista de elementos: en el primer elemento encontramos una lista con todos los nombres de todas las ventanas con la estructura comentada anteriormente, en el segundo elemento encontramos una lista con las imágenes digitales de las ventanas y en el tercer elemento encontramos una lista con las etiquetas de las ventanas. Obviamente, los índices de las listas coinciden; es decir, si el nombre de la ventana que queremos recuperar ocupa el índice i en la lista de los nombres, en dicho índice en la segunda y tercera lista encontraremos su imágen digital y su etiqueta.

El segundo *script* que se encuentra en esta primera parte, GrayScaleToBN.py, leerá el objeto Names_X_XLab_NumPxls_50x100 y sobreescribirá la lista de la imagen digital. Lo que hará es, a cada imagen de la lista, convertirla en escala de grises, eliminando así la dimensión de color, y convertirla de una matriz NxM a una fila de tamaño 1x(NxM) elementos, ya que las librerías que hemos usado para las diferentes técnicas de aprendizaje automático necesitan que los datos tengan esa forma para poder funcionar.

4.2.2. Proceso

Esta parte está formada por los *scripts* kmeans.py, knn.py, svm.py, svmGaussian.py, svmPolynomial.py, svmSigmoid.py y TreeClassification.py.

Cada uno está implementando una técnica a usar, sobre las cuales se harán pruebas para ver cuáles de las posibles configuraciones funciona mejor. kmeans implementa k-medias; knn implementa k-vecinos más cercanos; svm implementa máquina vectores soporte, y sus variantes con kernels son, svmGaussian para el kernel gaussiano y svmPolynomial para el kernel polinomial; por último,

`TreeClassification` implementa el algoritmo árbol de decisión. Todos estos *scripts* usan los objetos generados por el bloque de información anterior y generan a su vez un bloc de notas por *script* con el nombre del *script* más `Results` (por ejemplo `kmeansResults.txt`). Este archivo de texto contiene los resultados de tasa de acierto y precisión de las diferentes configuraciones del *script*.

4.2.3. Postproceso

Por último, el último bloque sólo contendrá el *script* `postProcess.py`, el cual recogerá los objetos generados en el primer bloque y actualizados en el segundo y sus bloc de notas, y tratará de hacer un postproceso para lograr una mejor clasificación de los datos ya clasificados.

4.3. Tecnología utilizada

En este punto vamos a exponer la tecnología hardware y software usada en este proyecto.

4.3.1. Software

En cuanto al software, el lenguaje usado para implementar el proyecto es python y las librerías a usar han sido las siguientes.

Después de investigar diferentes librerías de aprendizaje automático, una de las más usadas en otros TFG y blogs, con más apoyo de la comunidad a la hora de resolver dudas y con una muy buena documentación, es `scikit-learn` [17]. Por estos puntos expuestos se ha decidido su uso para desarrollar este proyecto. Para trabajar con matrices más cómodamente se ha usado la librería `numpy` [18]; ésta es necesaria para la entrada de datos de los algoritmos de las librerías de `scikit-learn`, ya que tienen que estar en formato array [19]. También se ha usado `opencv` [20] para el tratamiento de imagen, ya que prácticamente monopoliza esta rama por la gran cantidad de información sobre ella que hay en red y su fácil uso a la hora de conseguir resultados. Para manejar los diferentes objetos que se usan para la

lectura/escritura de estructuras de datos en diferentes *scripts* se ha usado la librería `pickle` [21] de python.

Todo esto se ha decidido implementar en base Linux, ya que a lo largo de la carrera hemos conseguido unas buenas bases de su *shell*. No obstante, como no queríamos usar máquinas virtuales, las cuales no nos dejan usar toda la potencia del ordenador, y tampoco queríamos hacer un particionado de disco, nos hemos decantado por usar un kernel de Linux [22] instalado en Windows (ver figura 27), lo cual nos permite usar un terminal de Linux en Windows y, por tanto, usar toda la potencia disponible del hardware, ya que no dependemos de máquinas virtuales.

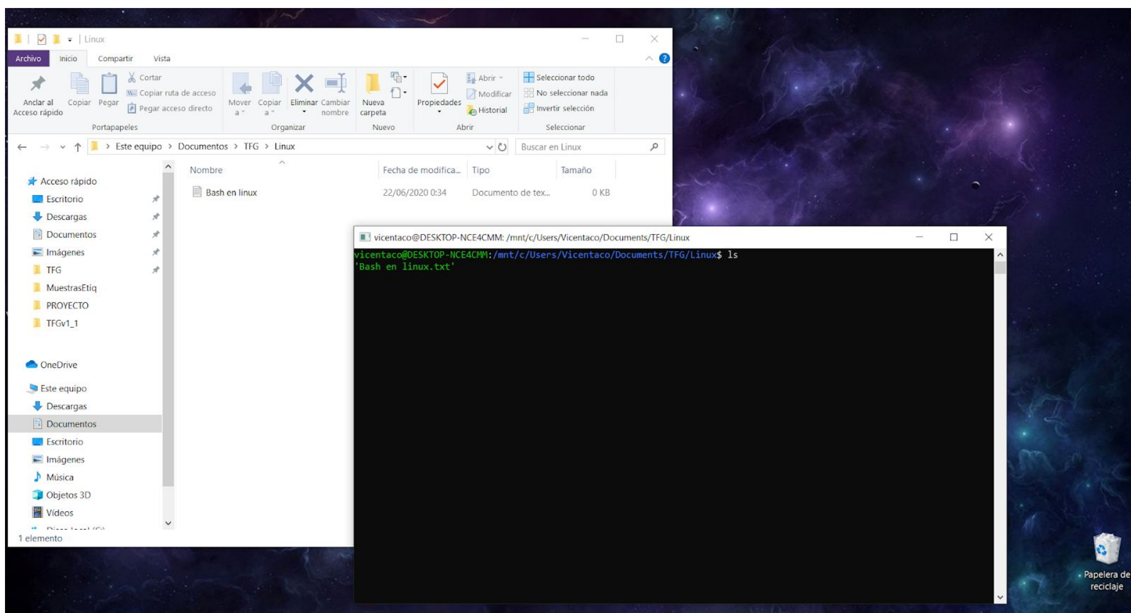


Figura 27. Ejemplo *shell* Linux ejecutándose en Windows 10.

4.3.2. Hardware

En cuanto al hardware, todo el proyecto se ha desarrollado en un ordenador Windows 10 de 64 bits con 8gb de RAM y un procesador Intel i5-8250U con ocho núcleos a 1.60GHz.

5. Desarrollo de la solución propuesta

El primer problema con el que nos encontramos es decidir cómo empezar a desarrollar nuestra aplicación. Para ello vamos a ver qué tenemos y qué necesitamos. Necesitamos un conjunto de muestras con la misma cantidad de características para crear modelos de aprendizaje automático y tenemos las páginas escaneadas de nuestro libro a color.

Por ello, lo primero que hacemos es empezar a partir las imágenes. Con la librería *numpy* y conociendo el tamaño de la imagen (imaginemos 2000x3000 píxeles), con un par de bucles podemos hacer un recorrido de las imágenes extrayendo las diferentes ventanas.

Aquí nos encontramos con un problema: se necesitan muestras etiquetadas. No obstante, nosotros sólo tenemos las imágenes sin ninguna marca, por lo que llegamos a la conclusión de que necesitamos alguna manera de conocer cuándo aparece una palabra partida en la imagen que se está procesando y así asignar a la ventana su clase en nuestra estructura de datos.

La primera opción que se nos ocurre es marcar de alguna manera la imagen para que cuando encontremos en nuestra ventana dicha marca, clasificarla de cierta manera. Leyendo las diferentes funciones que tenemos en `opencv` [23], vemos que podemos encontrar un color en las imágenes, por lo que procedemos a marcar las imágenes y así poder clasificarlas y guardarlas en una estructura de datos para su posterior uso.

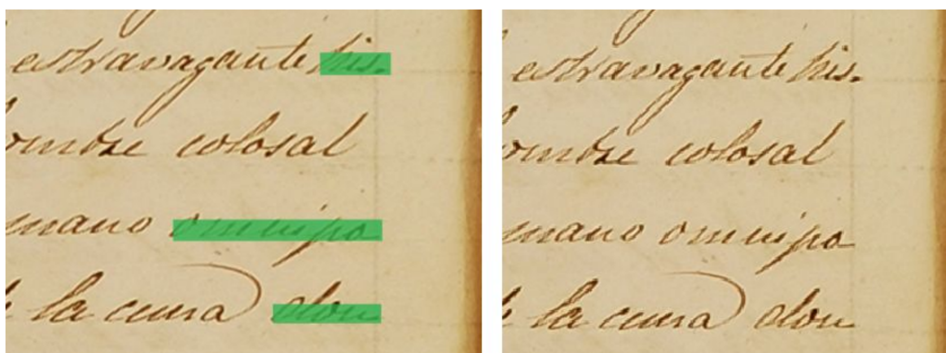


Figura 28. Marcas para la clasificación de datos.

Usaremos un rectángulo verde, tal y como se muestra en la figura 28, y mediante un método de `opencv` podremos averiguar el valor de un píxel. Si este píxel es verde, etiquetaremos como palabra partida a la muestra. Nada más etiquetar unas muestras y testearlas para empezar a fabricar nuestra base de datos etiquetada, nos damos cuenta de que no funciona del todo bien, ya que hay palabras etiquetadas que no se clasifican bien. El causante es el color de la imagen, ya que al ser un escáner a color, la mezcla con el verde del etiquetado con el color del píxel no siempre es el mísmo verde, lo que hace que haya palabras que, aunque tengan la etiqueta, se salgan del rango de colores verde donde estamos buscando, y si usamos un rango demasiado alto se cuelan etiquetas que no tienen el etiquetado. Por ello se decide usar un tono exacto que no aparezca en toda la imagen para que se encuentre la etiqueta sin fallo, tal y como se muestra en la figura 29.

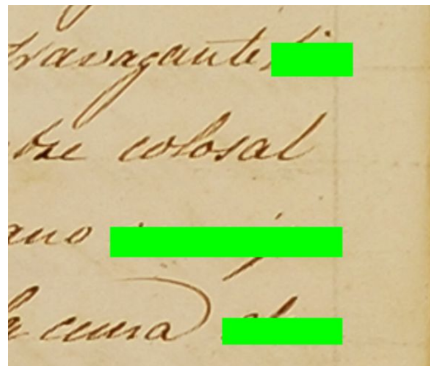


Figura 29. Marcas para la clasificación nuevas.

Con esto solucionamos este problema y ya disponemos de la manera de encontrar las etiquetas. Al intentar volver a generar la base de datos de muestras, vemos que el problema de etiquetado ya se ha resuelto. Sin embargo, seguimos encontrando problemas, y es que necesitamos que todas las muestras tengan la misma cantidad de características, en este caso de píxeles, y nuestro *script* periódicamente produce muestras más pequeñas del tamaño de ventana estipulado.

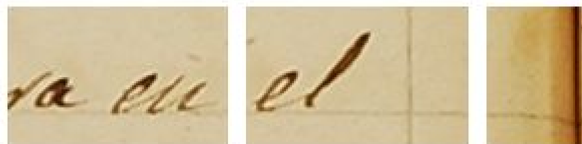


Figura 30. Empezando por la izquierda, las dos primeras muestras tienen el tamaño de ventana estipulado, la tercera no.

Revisando las muestras enseguida nos fijamos que cuando nos acercamos al borde derecho y/o al borde inferior de la imagen aparecen estas ventanas más pequeñas, y esto es porque al tener un tamaño de ventana fijado y las imágenes no tener siempre el mismo tamaño, estos pueden ser no múltiplos, y se genera la ventana con el tamaño sobrante de la imagen. Esto se puede apreciar en la figura 30, donde la tercera ventana empezando por la izquierda se genera en esta situación.

Con una condición *if* para observar si el tamaño de la ventana recién creada es menor al decidido, podemos detectar estas ventanas y añadirle las características que le hagan falta para que tenga dicho tamaño. El sistema es sencillo: cuando nos encontramos frente a una ventana que es menor de lo que debería, y conociendo este tamaño fijado, añadimos a la ventana tantas características como haga falta en orden para generar una matriz del mismo tamaño. Un ejemplo gráfico se muestra en la figura 31.

Figura 31. Arriba, una ventana con un tamaño correcto, abajo de izquierda a derecha una ventana menor al tamaño esperado se convierte en una ventana con el tamaño correcto.

A estas muestras se les añadirá un color plano, como el blanco, donde no se puedan encontrar formas o patrones, para no generar ruido. Así cuando convirtamos las muestras a escala de grises esto no nos generará problema alguno.

Ahora ya tenemos todo listo para generar una base de datos de muestras etiquetadas todas del mismo tamaño y sin fallo de clasificación. Sin embargo, aún nos quedan parámetros a decidir, como de qué tamaño será la ventana, qué características consideraremos para cada muestra y qué consideramos como un buen etiquetado.

Respecto al primer parámetro, hay un dato que nos ayudará a decidir un tamaño de ventana aceptable, y es que, a pesar de que las imágenes de donde vamos a generar las ventanas no siempre tienen el mismo tamaño, este tamaño se mueve en

un rango pequeño, el cual es entre 1500 y 1600 píxeles de ancho y 2100 y 2200 de largo. Con este tamaño de imagen tenemos entre 3150000 y 3520000 píxeles por imagen. Teniendo esto en cuenta, buscaremos un equilibrio entre una menor cantidad de características para no caer en la *maldición de la hiperdimensionalidad* [24] sin perder las cantidades mínimas para poder definir el problema, y tampoco generar muchísimas muestras para que no exista *overfitting*, lo cual hace que al tener tantas muestras para construir un modelo, éste empieza a memorizar aspectos del conjunto de muestras y pierda la habilidad de generalizar para conseguir una mejor clasificación.

Figura 32. Por cada fila, un ejemplo de posibles marcos de etiquetado.

Enseguida nos damos cuenta de que esta tarea se va a resolver por prueba y error dentro de una regla, y es que las palabras se puedan identificar bien dentro de la ventana. Por ejemplo, la segunda fila de la figura 32 no cumple esta característica, ya que la palabra se parte. En la tercera fila se diferencian varias palabras, por lo que no nos interesa ya que podemos estar mostrando al modelo unas características para las palabras partidas que no se correspondan con la realidad, ya que las palabras partidas sólo dependen de sus propias características. La primera fila de la figura en cuestión tampoco nos interesa, ya que al ser una ventana muy pequeña generamos muchísimas muestras y, como hemos expuesto anteriormente, puede provocarnos problemas.

Por tanto, en vista de las posibles muestras de la figura 32, viendo que la altura de las muestras de la primera fila permite ver la palabra bien será esta altura la que usaremos. De anchura se ha decidido usar 100 píxeles, generando unas 20 ventanas por fila. Esto nos dejará una ventana de 50 x 100 píxeles.

Una vez tenemos decidido el tamaño de ventana en el cual vamos a reconocer las palabras partidas vamos a decidir qué características formarán cada muestra de nuestra base de datos. De momento sabemos que nuestras muestras son ventanas de 50 x 100 píxeles, pero como contienen RGB, por cada posición de la ventana tenemos 3 valores por posición, lo cual multiplica por tres el número de características por ventana. Como queremos las muestras en escala de grises, como hemos argumentado anteriormente, convertiremos los píxeles a blanco/negro, lo que elimina los 3 valores y lo representa en 1, quedándonos 5000 características (50x100x1) por muestra. Estas características con valores de 0 a 255 representan la ventana en escala de grises. Otra característica que nos define la ventana es el desplazamiento relativo que tiene la ventana en la imagen, el cual es importante a la hora de reconocer las palabras partidas. Como es una característica conocida se la podemos asignar a la muestra. La figura 33 nos ayudará a explicar esto.

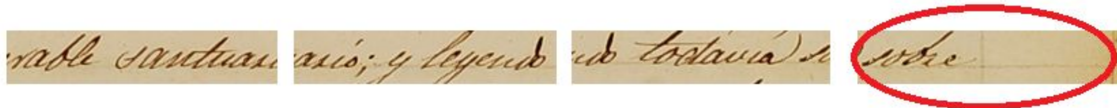


Figura 33. Cuatro muestras sin etiquetar.

En la figura 33 se pueden ver cuatro muestras. Suponiendo que la primera por la derecha es la muestra con desplazamiento hacia abajo 0 y hacia la izquierda 0, la muestra marcada con un círculo rojo tendrá desplazamiento hacia abajo 0 y hacia la izquierda 3. Estas dos características nos dan mucha información, ya que nos están indicando la posición de la muestra en el texto. En este problema es una gran información, ya que en las palabras partidas la primera parte se encuentra al final de una línea y la segunda parte al principio de la siguiente.

Nos queda por decidir qué consideramos una buena clasificación. Con esto nos referimos a que la máquina no sabe qué es una palabra partida a la hora de procesar la imagen, pues sólo diferencia entre ventanas que contienen color verde y ventanas que no a la hora de clasificarlo. Esto a veces es bueno, ya que si, por ejemplo, en una ventana hay un único píxel verde porque dicha ventana toca la esquina de una muestra, habría que decidir si esta ventana se considera que forma parte de una palabra partida. En principio, la respuesta es no, ya que no aparece ninguna característica que diga que es una palabra partida. Simplemente la ventana

ha cogido 1 píxel verde que está marcando una palabra cercana. En la figura 34 vemos un ejemplo de este problema.

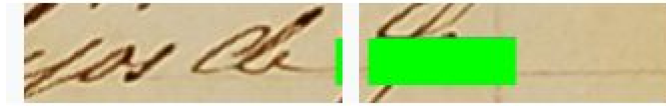


Figura 34. Muestras etiquetadas como palabra partida.

En la figura 34 ambas muestras contienen píxeles en verde; no obstante, no nos proporcionan la misma información las dos. En la segunda aparece una palabra partida, por lo que es muy útil para la enseñanza de nuestro modelo; sin embargo, la primera sólo proporciona ruido, ya que estará etiquetada como palabra partida pero no aparece ninguna característica que nos ayude a clasificar muestras como palabra partida, ya que no aparece ninguna. La manera de hacer que esto no pase es indicar una cantidad mínima de píxeles en verde para la ventana, para que, aún apareciendo verde, si éste es muy poco porque está capturando una muy pequeña fracción de lo que corresponde realmente a la palabra partida, no se clasifique como palabra partida, para no enseñar al modelo con datos erróneos.

La cantidad mínima que se decide después de unas pruebas será de 200 píxeles verdes para poder considerar una ventana de la clase palabra partida. Una vez decidimos estos tres parámetros, generamos nuestra base de datos, obteniendo 55982 muestras. 52590 no contienen palabras partidas, mientras que 3392 sí que las contienen.

Las muestras están formadas por 5000 características representadas por una matriz de 50x100 elementos y dos enteros que nos indican su posición en el texto. Esto no puede ser usado para entrenar nuestros modelos, ya que no tienen un formato homogéneo ni están en la misma escala. Por ello, se decide estructurar cada muestra como una matriz de 1x5000 (matriz) + 2 (posición), teniendo 5002 características en una fila. Una vez hemos hecho esto, procederemos a estandarizar los datos [25], escalando todas las muestras. Así todas las muestras tendrán el mismo rango a la hora de entrenar el modelo.

Al empezar a generar las muestras en escala de grises, nos damos cuenta de que al ser imágenes digitales de páginas de libros antiguos (como ya hemos comentado en el problema del etiquetado) todas las páginas no tienen el mismo color,

ya que no han envejecido igual. Por tanto, al convertirlas a escala de grises no todas las muestras tienen el mismo aspecto.

Figura 35. Diferentes ventanas en escala de grises.

Como se observa en la figura 35, al no tener un mismo color siempre la pagina, no todas las muestras tienen la misma escala de grises, lo que nos genera ruido ya que lo único que queremos que nuestros modelos aprendan es su silueta y su posición en el texto. Por este motivo pasaremos todas las imágenes por un filtro para convertirlas a blanco y negro, como se puede ver en la figura 36.

Figura 36. Ventanas de la figura 35 a escala blanco y negro.

Al hacer esto conseguimos que las muestras sólo contengan como características la silueta de la palabra y su posición. Convirtiéndolas también a una línea ya podemos usarlas para entrenar nuestros modelos.

Una vez conocemos cuántas muestras tenemos en nuestra base de datos (55000) y qué número de características hay por muestra (5002), vamos a proceder a decidir el siguiente parámetro. Éste es el porcentaje de datos que se usará para entrenamiento del modelo y, consecuentemente, el porcentaje para el testeo del modelo. Dadas la cantidad de muestras y características se decide usar como parámetro un 80% de las muestras para entrenamiento y un 20% para el test del modelo generado [26]. No obstante, en la fase de pruebas expuesta en el capítulo 6 se probarán otras configuraciones en algún algoritmo.

Al implementar los algoritmos nos encontramos con problemas como interpretar los datos de salida. Leyendo la documentación de la librería `sci-kit learn` solucionamos éstos. Nos damos cuenta de que, dada la naturaleza de las muestras (contienen un alto número de características), éstas hacen que los algoritmos sean muy lentos, ya que en los cálculos las matrices usadas son muy

grandes (dadas nuestra cantidad de muestras y características). También, dada la alta cantidad de características y el pequeño número de muestras clasificadas como palabras partidas (3392) frente a las clasificadas como palabras no partidas (52590), obtenemos errores en la clasificación muy altos. Para resolver este problema existe la posibilidad de reducir la dimensión de las muestras mediante el estudio de qué características son las más relevantes y/o cuáles se pueden obviar por su poca trascendencia.

Existen varias técnicas de reducción de la dimensionalidad que funcionan bien para una alta cantidad de características. Nosotros vamos a usar una técnica conocida como PCA [27], la cual consiste en conocer las componentes con la más alta cantidad de varianza original que describen los datos y seleccionar las características que mejor representan a la muestra, obviando las que no y así reduciendo la dimensionalidad de la muestra. Con esta técnica probaremos varias dimensiones de las muestras en la fase de pruebas y decidiremos en qué algoritmo será una buena técnica a tener en cuenta y en cual no.

Una vez etiquetamos las muestras que hemos apartado para el test, y teniendo en cuenta la precisión que hemos tenido sobre ellas a la hora de clasificarlas, podemos reetiquetar esta base de datos dependiendo de la probabilidad de que sea una muestra que contenga una palabra partida o no. Por ejemplo, si una muestra clasificada como palabra no partida aparece al final del texto y las muestras adyacentes han sido clasificadas por nuestro modelo como palabras partidas, cabe la posibilidad de cambiarle la etiqueta a palabra partida. Esta técnica de postproceso tiene el inconveniente de que necesitamos un modelo entrenado y un conjunto de muestras de test clasificadas y no plasmará realmente la precisión del modelo, ya que el postproceso es una técnica ajena a los algoritmos de clasificación. No obstante, puede hacer que la resolución del problema sea mejor de lo que se ha conseguido en un principio mediante el estudio de los algoritmos.

6. Pruebas

En este apartado intentaremos encontrar una solución al problema que intentamos resolver probando los diferentes algoritmos expuestos con diferentes parámetros. El objetivo es encontrar el mejor modelo que clasifique correctamente las muestras que no contienen palabras partidas (etiqueta 0) y las que sí que contienen palabras partidas (etiqueta 1).

k - vecinos cercanos

Este algoritmo busca posicionar a las muestras en un hiperespacio y clasifica las nuevas muestras en relación a la etiqueta de las muestras más cercanas. La primera prueba que haremos tendrá como parámetros (prototipos) el 80% de las muestras para el entrenamiento del modelo y 20% para el test (esto nos deja 44785 muestras para el entrenamiento, y 11197 para el test) y cinco vecinos más cercanos a tener en cuenta para ver cómo funciona el algoritmo con los datos en su dimensión original y sin postproceso.

Al hacer el test al modelo generado obtenemos un 96.23% de acierto a la hora de clasificar las muestras del test. No obstante, este resultado no expresa la realidad, y por ello vamos a interpretar los datos devueltos.

Cuando miramos la tasa de acierto de clasificación de las clases, vemos que mientras que nuestro modelo acierta el 94.59% de las veces que decide clasificar en el grupo de las palabras no partidas, al clasificar una palabra partida acierta un 2.89% de las veces. Esto se puede deber a que la cantidad de muestras con palabras no partidas (10689) es muy superior a la de muestras que contienen palabras partidas (508) y la cantidad de características es muy alta, por lo perdemos la capacidad de generalizar y tendemos a clasificar como palabra no partida. Una solución que hemos aplicado, como hemos argumentado en el capítulo 5, es aplicar alguna técnica de reducción de la dimensionalidad, concretamente PCA. Al hacer una prueba con una baja dimensionalidad (dimensiones de 2 a 5000 en intervalos de 1000), los resultados son los que se muestran en la figura 37.

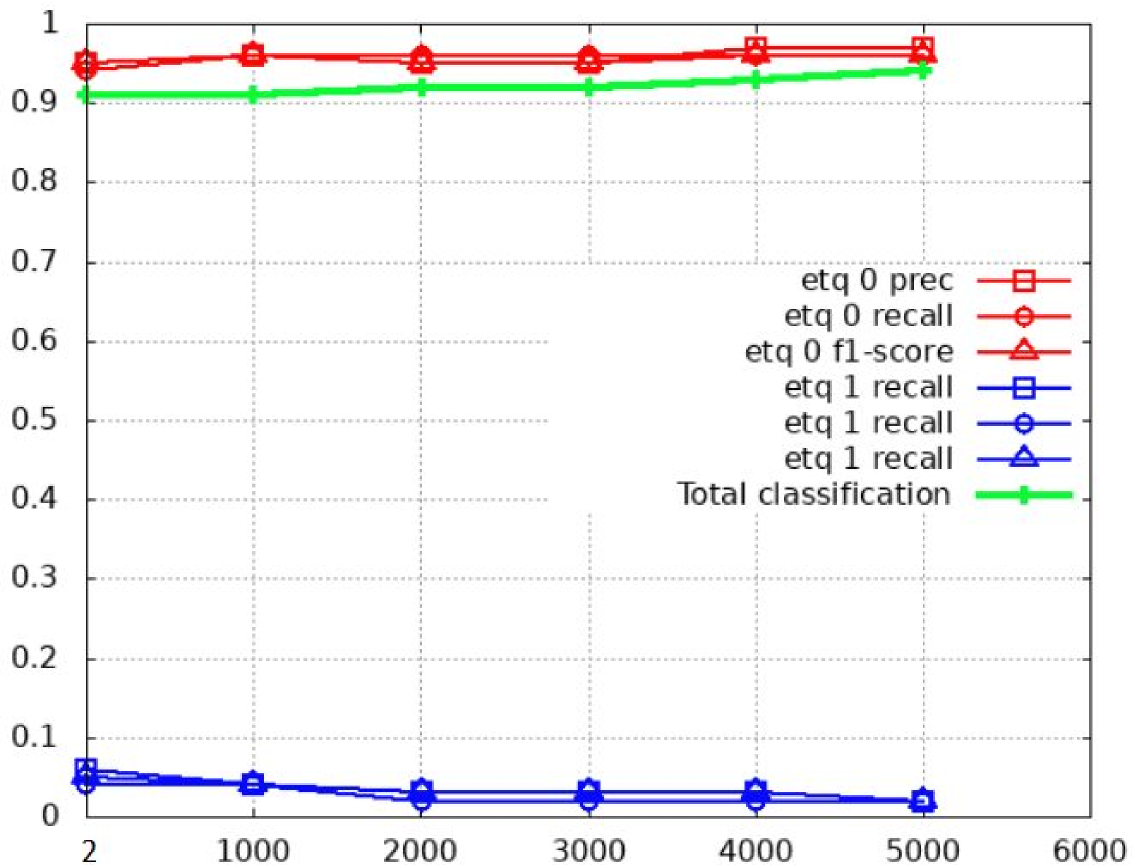


Figura 37. Tasas de resultados de clasificación de ambas clases y global. En el eje X está representada la dimensión a la que las muestras han sido reducidas. En el eje Y la tasa correspondiente siendo 1 el 100%.

En la figura 37 se representan siete curvas agrupadas en 3 grupos codificados en colores: la tasa de clasificación correcta de las palabras no partidas como etiqueta 0 en rojo, la tasa de clasificación correcta de las palabras partidas como etiqueta 1 en azul y en verde el porcentaje de acierto de todas las muestras. De cada etiqueta tenemos una curva de *precision*, *recall* y *f1-score* [28]. Cuando nos referimos a *precision* nos referimos a la cantidad de muestras de la clase que se han clasificado bien (es decir, las bien clasificadas en esa clase dividido por el total de las que se han clasificado en dicha clase); en esta curva, los puntos serán cuadrados. *Recall* cuantifica el número de predicciones positivas en el total de predicciones positivas (es decir, las clasificadas como de la clase dividido el total de elementos de dicha clase en el test); se marcará como círculos en la curva. *F1-score* nos da un valor que balancea *precision* y *recall* para darnos un valor único en cuanto a la calidad de la clasificación de la muestra; se marcará como triángulos. La curva verde, que es la precisión total de todas las muestras, representa el *f1-score* de éstas. Al representar tantas curvas en una sola gráfica no se aprecia bien la información, así que a partir de ahora se

representarán dos gráficas, una para cada clase, y en ambas estará representado el *f1-score* global.

Otro detalle del que nos damos cuenta al analizar estos datos es que hemos perdido mucha precisión a la hora de saber la mejor clasificación porque los puntos decididos para representar las dimensiones están muy alejados entre ellos, es decir, si hacemos dos pruebas, una con 1000 dimensiones y la otra con 2000, entre ambas hay muchas dimensiones no usadas que pueden tener un mejor comportamiento que estas dos. Lo que vemos es que tanto la clasificación de la clase 0 como la clasificación global tienen una tasa de acierto muy alta, mientras que la clase 1 no, y que cuanto más baja es la dimensión, mejor es esta tasa de acierto. Así que vamos a ver cómo se desarrolla este algoritmo con dimensiones muy bajas. En la figura 38 haremos un visionado de las 100 primeras dimensiones de 10 en 10 para ver con qué dimensión se comporta mejor.

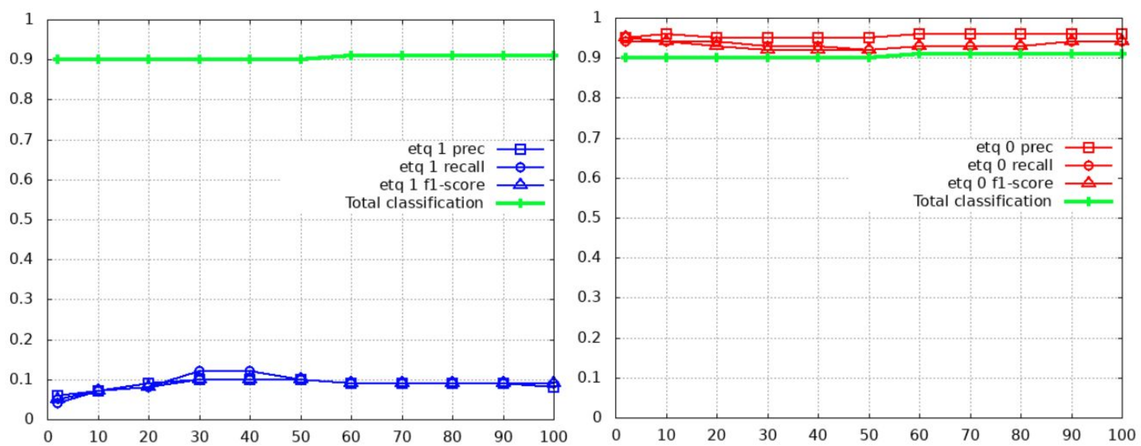


Figura 38. Tasas de resultados de clasificación para las etiquetas 1 (izquierda) y 0 (derecha) con reducción de dimensionalidad mediante PCA de 2 dimensiones y de 10 a 100 dimensiones de 10 en 10.

En esta figura vemos que la mejor solución que vamos a obtener para la clasificación de la etiqueta 1 es reduciendo la dimensión a 30 dimensiones. En esta obtenemos una clasificación global de 92.23% y la etiqueta 1 tiene un 10.57% de acierto (*f1-score*). Nos vamos a quedar con esta para seguir las pruebas sobre este algoritmo, ya que dada la naturaleza de las muestras es difícil conseguir mejores resultados. Esto se debe a que tenemos muchas muestras de la etiqueta 0 y muy pocas de la etiqueta 1 y al reducir las muestras ganamos capacidad de generalizar las muestras para un mejor entrenado del modelo.

Con este nuevo parámetro decidido, vamos a cambiar otro parámetro fijado anteriormente y que puede afectar a nuestra clasificación, y es la cantidad de vecinos más cercanos relevantes para la clasificación. Todas las pruebas hechas hasta ahora se han hecho con 5 vecinos más cercanos; por tanto, se va a probar con 1 y 10 vecinos más para ver cómo se comporta nuestro modelo incrementando/decrementando la importancia de su 'vecindario' sobre el que hemos hecho todas las pruebas.

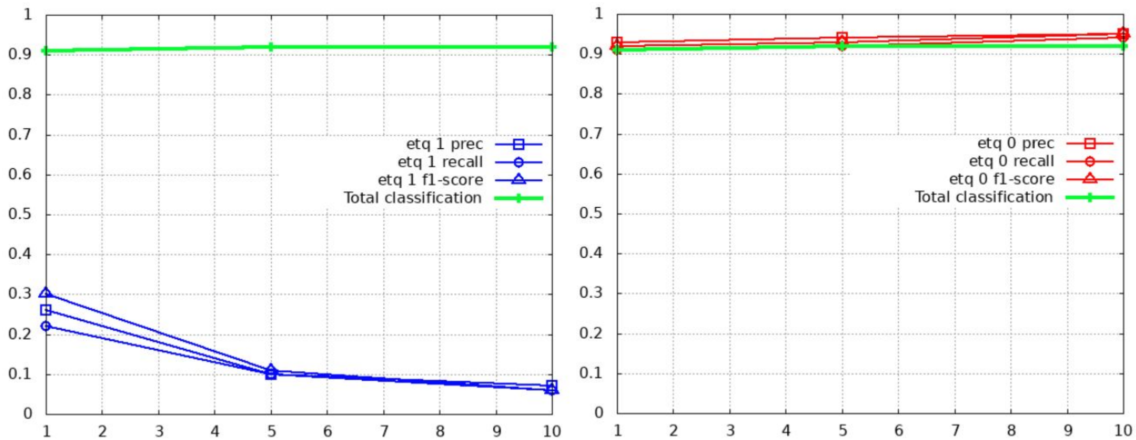


Figura 39. Tasas de resultados de clasificación para las etiquetas 1 (izquierda) y 0 (derecha) con reducción de dimensionalidad mediante PCA de 30 dimensiones y diferentes vecinos más cercanos (eje X).

Como se ve en la figura 39, la *f1-score* de la etiqueta 0 y la global rondan entre 90% y 95% de acierto. Lo que nos llama la atención es la *f1-score* de la etiqueta 1 es de 30.45% para 1 vecino y un 6.33% para los 10 vecinos. Como vemos, nos afecta mucho nuestro vecindario a la hora de clasificar muestras. Esto es lógico, dado que el gran desequilibrio de clases hace que, al ampliar el vecindario, encontremos con más frecuencia muestras de la clase 0. Por tanto, vamos a probar dos técnicas de ponderación de la importancia del vecindario para mejorar nuestra clasificación.

k-vecinos cercanos distancia media

En este caso, dados los k vecinos más cercanos, se hará la suma de las distancias de las muestras separadas por clase y se clasificará en la menor de estas. Conociendo cómo se ha comportado k-vecinos más cercanos, vamos a probar con una reducción a 30 dimensiones mediante PCA de las muestras y pocos vecinos. En la figura 40 vemos los resultados.

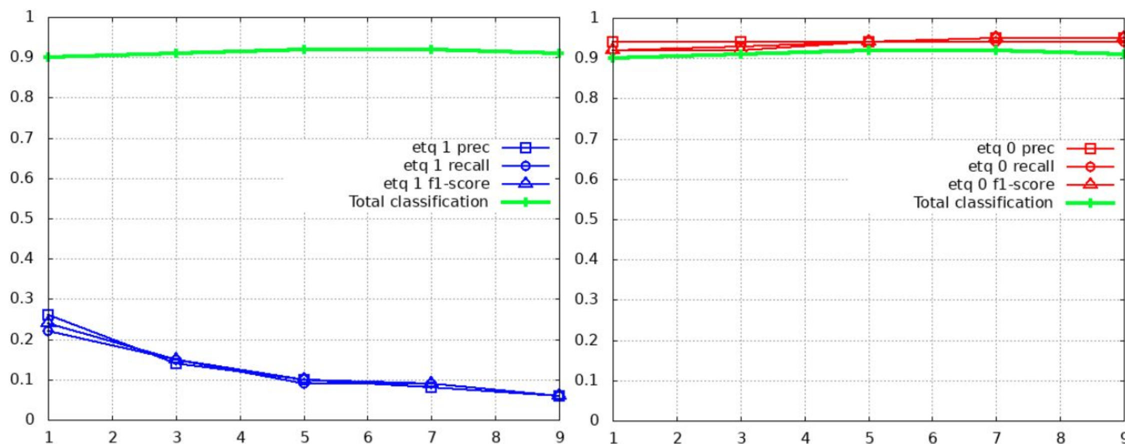


Figura 40. Tasas de resultados de clasificación para las etiquetas 1 (izquierda) y 0 (derecha) con k-vecinos más cercanos considerando la distancia media de las clases.

Se puede observar que no ha habido mejora en cuanto al algoritmo estándar con un vecino más cercano (se tiene un f1-score de 24.34%). Habría que considerar también otras dimensiones. Esta prueba se está haciendo reduciendo las muestras a 30 dimensiones, pero se podría hacer un análisis de esta prueba con diferentes reducciones ya que ahora el vecindario de la muestra a clasificar interactúa de forma diferente con la muestra para decidir una clasificación. No obstante, dado el coste computacional al que nos está sometiendo este algoritmo, no se va a realizar en este proyecto.

k-vecinos cercanos con pesado de casos seleccionados

En esta prueba se va a dar peso a las muestras en cuanto a la clase. Hemos decidido que ya que las muestras clasificadas como 1 son minoría, les vamos a dar más peso que a las pertenecientes a la clase 0. Teniendo en cuenta la relación de cantidades entre entre ambas (por cada muestra etiquetada como palabra partida hay 10 de palabras no partidas) vamos a dar un peso de 0.8 a las muestras de la clase 0 y un 0.2 a las muestras de clase 1. Haciendo esto obtenemos los resultados plasmados en la figura 41.

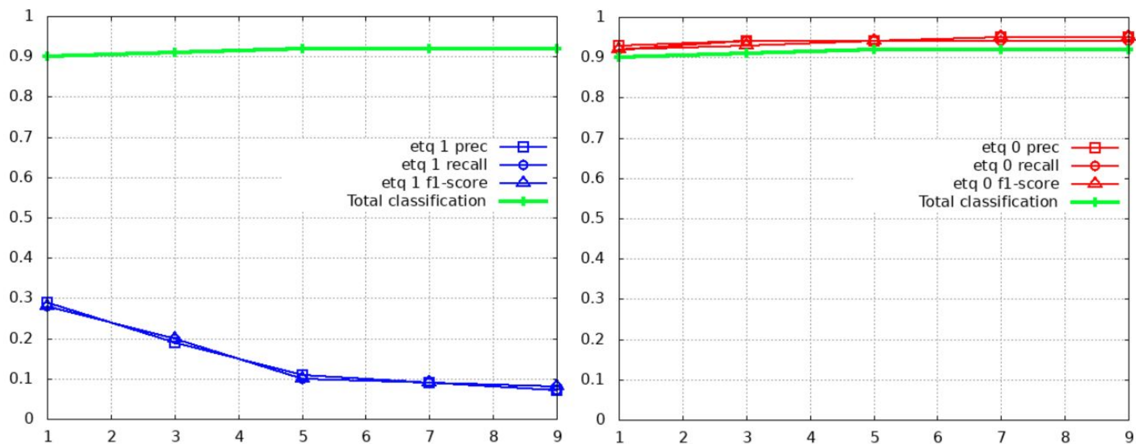


Figura 41. Tasas de resultados de clasificación para las etiquetas 1 (izquierda) y 0 (derecha) con k-vecinos más cercanos con peso.

Esta prueba nos confirma que cuantos menos vecinos, mejor va a funcionar la clasificación de la clase 1. En cambio, para la clase 0 mejora a cuantos más vecinos tengamos en cuenta. Dada la prioridad que tenemos hacia la clasificación de la clase 1, vamos a considerar su mayor importancia para el postprocesado que se hará al final de las pruebas con los mejores modelos generados. La mejor *f1-score* de estos modelos es 93.57% para la clase 0, 30.45% para la clase 1 y 91.28% del global de las muestras. Esto lo ha conseguido con el algoritmo k vecinos más cercanos estándar con un 80% del conjunto de datos para el entrenamiento y un 20% para el test, las muestras reducidas a la dimensión 30 y etiquetando con el primer vecino más cercano.

Máquinas de vectores soporte

Al ejecutar el algoritmo con los parámetros base (80% de las muestras para entrenamiento, 20% restante para el test, sin reducción de dimensionalidad), obtenemos una tasa global de acierto de 94.56%. No obstante, al mirar la *f1-score* de las clases, nos damos cuenta de que la clase 0 tiene una muy buena tasa (95.23%), mientras que la tasa de la clase 1 es muy baja (2.84%). Esto puede ser porque este algoritmo funciona muy bien cuando la dimensión del problema es mayor que la cantidad de muestras que tenemos para el entrenamiento [29] y en nuestro caso estamos usando 44000 muestras para el entrenamiento mientras que tenemos 5002 características en cada muestra. Por ello vamos a cambiar el tamaño de entrenamiento-test usado y vamos a ejecutar el algoritmo, sin reducir la dimensionalidad de las muestras. El nuevo tamaño elegido es un 10% para el

entrenamiento y un 90% para el test. En el experimento previo ha ocurrido que los valores de *precision* y *recall* eran iguales que el valor *f1-score*. En esta nueva partición de los datos se repite ese comportamiento, con lo que sólo se va a indicar el valor *f1-score*. En este nuevo caso, es de 90.84% para el acierto global, 91.43% para las muestras clasificadas de etiqueta 0 y 15.23% para las muestras etiquetadas con un 1. Al reducir la dimensión perdemos precisión en todas las muestras, así que, en vista de esta prueba, vamos a usar estos parámetros para los diferentes kernels.

Kernel polinomial

En este kernel tenemos como parámetro el grado del polinomio de la ecuación que aplica esta función a cada muestra, distribuyendo de nuevo las muestras en el hiperplano. Haremos pruebas con diferentes grados polinomiales, empezando por el 1 incrementando en 2 hasta grado 9. En la figura 42 se muestran los resultados.

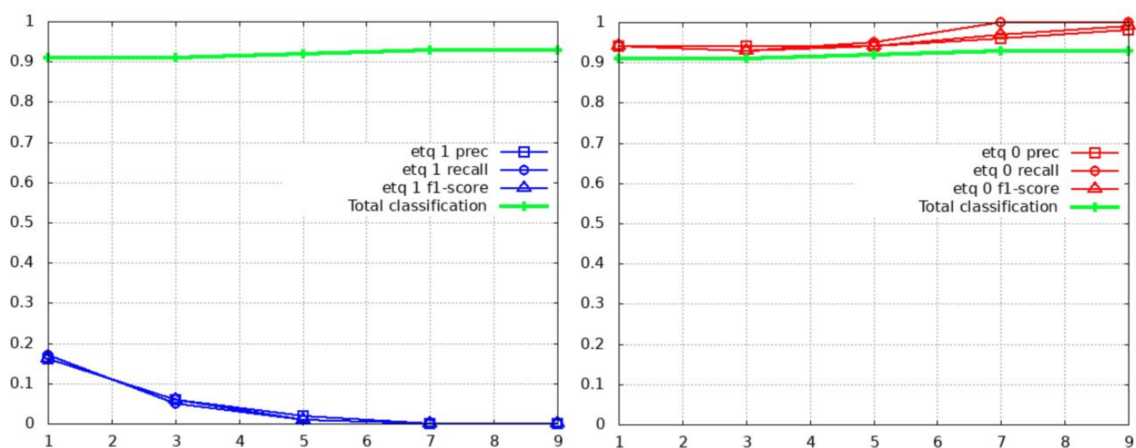


Figura 42. En el eje X se muestra el grado al que sometemos al kernel y en el eje Y las tasas de resultados para las muestras con etiquetas 1 (izquierda) y 0 (derecha).

El resultado de la prueba mostrado en la figura 42 nos muestra que para la etiqueta 1, usando el kernel polinomial de primer grado obtenemos un 17.11% de *f1-score*, mientras que en la clase 0 y la clasificación global se mantienen por encima del 90% de acierto. Este kernel no nos lleva a ningún sitio, ya que cuando aplica la función kernel, al haber pocas muestras de la clase 1, éstas se quedan mezcladas con las muchas muestras que tenemos de la clase 0, haciendo que muchas acaben mal clasificadas. Si nos fijamos, cuanto mayor sea el grado, el valor de *recall* pasa a ser un 100% y los valores para la clase 1 son del 0%. Esto es porque llega a desplazar tanto

las muestras por el hiperespacio formado que el algoritmo no llega a construir una frontera de decisión correcta y prácticamente todas las muestras se clasifican como 0.

Kernel base radial gaussiana

En este kernel tenemos dos parámetros: *Gamma* y *C*. *Gamma* es la influencia que tiene una muestra en su entorno, a la hora de decidir o no sobre la posición de la frontera de decisión. Cuanto más alto sea el valor, más cercana es la influencia de la muestra sobre su entorno y cuanto más bajo el valor, la influencia de esta muestra será más amplia en su entorno, El parámetro *C* sirve para priorizar la tasa de acierto de clasificación sobre el tamaño de los márgenes, prefiriendo una buena tasa de acierto y un margen muy estrecho sobre la frontera de decisión que unos márgenes más anchos en la frontera sobre una tasa de acierto menor.

En esta prueba vamos a hacer cuatro tests con los parámetros base que hemos decidido al principio de este apartado (80% de entrenamiento, 20% de test y las muestras sin reducción de dimensionalidad). En la tabla 1 se muestran los resultados de estas combinaciones.

Tabla 1. Valores de *f1-score* de las clases específicas y global para los diferentes valores de los parámetros *gamma* y *C*.

C	Clases	Gamma		
		0.1	1	10
0.001	Global	0.91	0.91	0.92
	0	0.92	0.92	0.92
	1	0.12	0.13	0.14
1	Global	0.92	0.92	0.93
	0	0.93	0.94	0.94
	1	0.12	0.12	0.20
100	Global	0.92	0.93	0.94
	0	0.93	0.94	0.95
	1	0.13	0.17	0.31

En la tabla 1 tenemos las *f1-score* global y de las muestras, ya que los valores de *precision* y *recall* son muy similares a éste y no aportan más información de la que ya nos aporta *f1-score*. En esta prueba podemos observar la naturaleza de los datos; se puede observar que están bastante agrupados, ya que cuanto más pequeños son los márgenes de la frontera de decisión (lo que implica que se tiene más espacio de clasificación en los subespacios divididos y que la influencia de la muestra es menor cuanta más distancia se recorre en el hiperespacio) mejor es la clasificación, obteniendo unos *f1-score* globales de 94.44%, 95.28% para la clase 0 y 30.89% para la clase 1.

Kernel sigmoid

En este kernel no vamos a entrar en detalles en los parámetros. Tenemos *gamma*, del cual usaremos el valor por defecto, y un término independiente de la función, del que se usará el valor 0 puesto por defecto. Lo que nos interesa de este kernel es su función, ya que esta función funciona como una red neuronal simple con dos estados, y resulta interesante ver cómo se comporta con nuestros datos. Al ejecutar este kernel con los parámetros por defecto (un 80% de las muestras para el entrenamiento, 20% para el test y los datos en la dimensión original del problema) obtenemos un *f1-score* del 91.23% en la tasa global de las muestras, un 90.74% para la clase 0 y un 32.52% para la clase 1. En este test clasifica en general muy bien, pero la clase 0 sólo acierta 1 de cada 3 muestras.

Árboles de decisión

Este algoritmo usará las muestras para construir un árbol de decisión donde las decisiones para tomar un camino u otro desde la raíz hasta un nodo hoja serán las características presentes en la muestra a clasificar. Dependiendo de la etiqueta del nodo hoja alcanzado se etiquetará la muestra. Este algoritmo consta de parámetros para elegir una cantidad de nodos mínimos (por defecto los necesarios), una cantidad máxima de profundidad (por defecto no hay máximo), una cantidad máxima de anchura (por defecto no hay máximo), así como un criterio para dividir un nodo en dos nodos hijos (por defecto se busca la mejor función de decisión posible, el otro parámetro que no se va a testar es usar una función aleatoria). Se dejarán como el valor por defecto para las pruebas.

Vamos a probar la mínima dimensión posible y la máxima dimensión posible, siendo ésta la original. También haremos pruebas con dimensiones intermedias entre estas dos, como se ve en la figura 43.

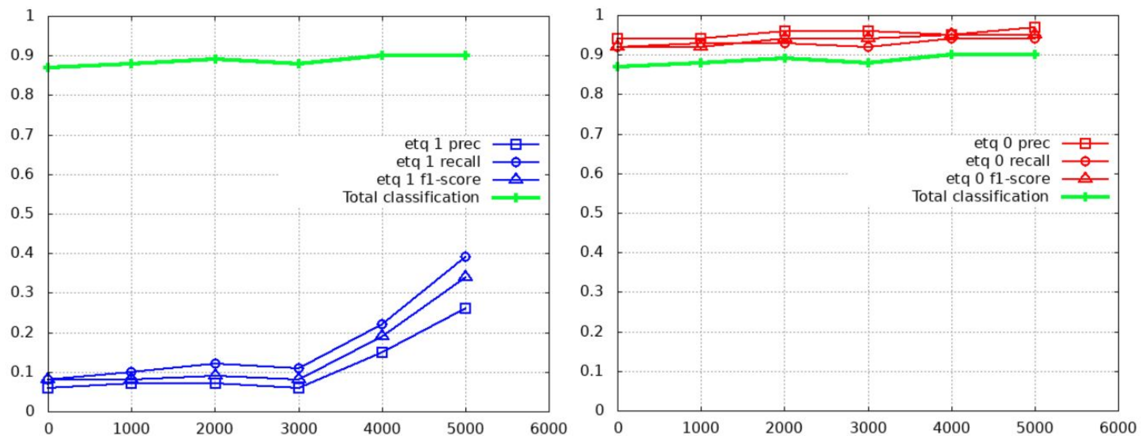


Figura 43. En el eje x se muestra la subdimensión de las muestras y en el eje y la tasa de acierto de las muestras 0 (izquierda) y 1 (derecha).

Como se muestra en la figura 43, la clasificación global y el porcentaje de acierto del etiquetado está por encima del 90%. Al ver las tasas de resultados de la etiqueta 1 observamos que cuantas más características por muestra tengamos, mayor será el valor de las distintas medidas. Esto puede ser debido a que al tener tantos valores, la decisión de ramificación y/o clasificación pueda tener más opciones para decidir. La mejor tasa de clasificación que obtenemos guiándonos por el valor *f1-score* es hacer la clasificación con las muestras en la dimensión original. En este caso obtenemos un 90.45% para todas las etiquetas, 95.23% para la clase 0 y 34.56% para la clase 1. Algo a tener en cuenta es la diferencia entre *precision* (27.31%) y *recall* (38.21%) para la clase 1; esto puede ser debido a que existen elementos repetitivos, lo que hace que el modelo vea las diferencias a la hora de clasificar la muestra en el grupo 0 o grupo 1 (*recall*), pero el modelo también ve estos patrones en muestras que no son etiqueta 1 y las clasifica como tal, bajando el valor *precision*. Teniendo en cuenta las *f1-score* [30], este modelo se considera una opción para aplicarle técnicas de postproceso, ya que es uno de los mejores obtenidos de las pruebas que se han hecho con los algoritmos de clasificación supervisado hasta el momento.

k-medias

Este algoritmo es el único que no usará el etiquetado de las muestras para entrenar un modelo para el posterior testeo de otras muestras, ya que es un algoritmo

de aprendizaje no supervisado. Tenemos hasta once parámetros a decidir: asignar una precomputación de distancias entre las muestras, un número máximo de iteraciones del algoritmo para calcular las agrupaciones, tiempo máximo que va a durar el cálculo de los centroides en cada agrupación, así como el número de agrupaciones que se van a hacer en el hiperespacio (este tiene un valor mínimo de dos por defecto). Ya que hay muchos parámetros y este algoritmo tiene un coste computacional y temporal alto, vamos a variar sólo uno, la cantidad de agrupaciones que se van a hacer. Dada la alta posibilidad de que nuestros datos no sean separables linealmente por su alta dimensión de representación, puede que necesitemos más de dos grupos para clasificar las muestras en 0 o 1, es decir, varias agrupaciones que clasifiquen como alguna etiqueta. Otro parámetro que probaremos será la dimensionalidad de los datos.

Al hacer pruebas vemos que el *f1-score* es de un 0% en las primeras pruebas realizadas. Esto se debe a que estandarizar los datos hace que *k-means* funcione de una manera diferente a la que debería según la naturaleza de los datos [31]. Al probar los datos sin estandarizar a una escala obtenemos los resultados que se muestran en la tabla 2.

Tabla 2. Valores de *f1-score* de las clases específicas y global para las diferentes combinaciones de reducción de dimensión (PCA -> 2, 2502 y 5000) y cantidad de *clústeres*.

PCA	Clases	Clústers				
		2	4	6	8	10
2	Global	0.90	0.91	0.91	0.91	0.90
	0	0.91	0.92	0.92	0.92	0.92
	1	0.08	0.10	0.09	0.08	0.06
2502	Global	0.91	0.91	0.91	0.91	0.90
	0	0.92	0.92	0.92	0.92	0.91
	1	0.10	0.13	0.10	0.09	0.06
5002	Global	0.91	0.92	0.92	0.91	0.90
	0	0.92	0.92	0.92	0.92	0.91
	1	0.16	0.21	0.12	0.08	0.06

Teniendo en cuenta las *f1-score* de la tabla 2 (*precision* y *recall* no se representan ya que tienen unos valores parecidos a ésta), la mejor clasificación ha sido con los datos en la dimensión original y cuatro agrupaciones a conseguir en el hiperespacio. Esta prueba ha conseguido unos *f1-score* de 92.23% para todas las muestras, 92.44% para la etiqueta 0 y 21.34% para la etiqueta 1. Dado que se extenderían mucho si probamos diversas configuraciones del algoritmo y los resultados obtenidos, se decide no continuar las pruebas (dado que tenemos mejores modelos) y pasar al siguiente paso.

Postproceso

En este punto vamos a aplicar dos postprocesos al mejor modelo que se ha conseguido. Este ha sido el árbol de decisión con las muestras en dimensión original, un 80% de las muestras para el entrenamiento y un 20% de éstas para el test. Hemos obtenido un 90.45% en la tasa de acierto de clasificación de todas las etiquetas, 95.23% para la clase 1 y 34.56% para la clase 0. Mientras que el valor de *precision* y *recall* para la etiqueta 0 se mantenía parejo a su *f1-score*, no era el caso de la etiqueta 1, que tenía un valor *precision* de 27.31% y un *recall* de 38.21%.

El primer postproceso que podemos aplicar es, con las muestras clasificadas generadas por este modelo, observar las características que nos dice la posición de la ventana que representa la muestra en la imagen y ver si en su vecindario próximo de ventanas existe una que ha sido clasificada como palabra partida durante la fase de test del modelo. Si ésta aparece en una posición horizontal cercana a la muestra que se está intentando reetiquetar cambiaremos su etiqueta a palabra partida (etiqueta 1), ya que es posible que ésta lo sea. También pasará lo mismo con la relación de ventanas en diferentes filas, ya que si la última ventana de la fila primera está clasificada como 1 y la primera ventana de la fila segunda no, ésta considerará su etiqueta ya que su vecindario es prometedor.

Otra técnica de postproceso complementario a éste que se puede hacer para ayudar a la clasificación es cambiar la etiqueta de las muestras que tengan etiqueta 1 estando en una posición centrada en el texto. Esto es, sabiendo que nuestras ventanas van de la 0 (ventana más a la izquierda) a la 19 (muestra más a la derecha) y conociendo que las palabras partidas sólo se dan a principio y a final de línea, podemos afirmar que en las posiciones entre estas dos (posición 10 por

ejemplo) no aparecerán palabras partidas. Así pues, conociendo su vecindario se puede tomar una decisión sobre reclasificar la muestra.

El problema ahora se trata de encontrar las fronteras de posición sobre las cuales se hará un postprocesado (ver si hay palabras partidas cerca para tomar una decisión de reclasificado) u otro (cambiar etiquetas a las ventanas centrales). En la figura 44 se muestran diferentes resultados para varias posiciones de decisión.

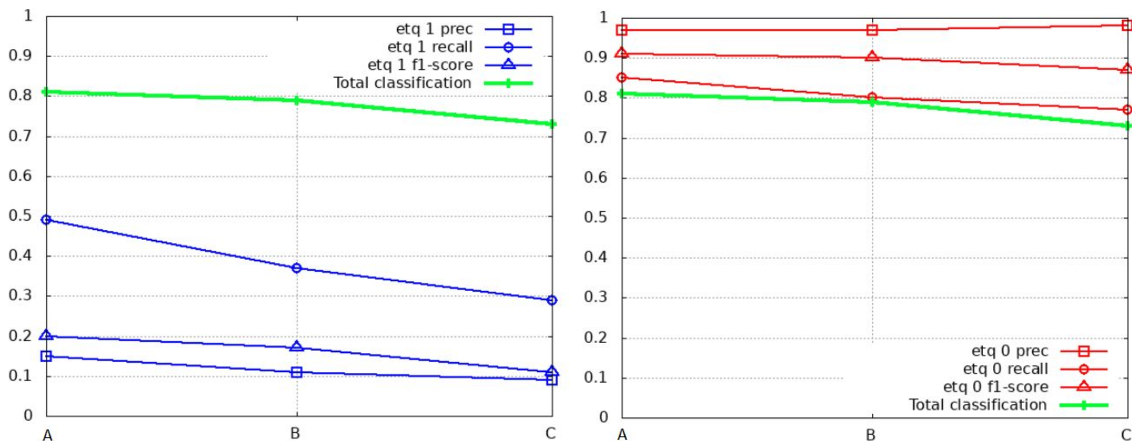


Figura 44. En el eje X se contemplan tres parámetros usados, A,B y C. Estos dan un valor para el rango de posición donde se toma la decisión de qué postproceso aplicar, teniendo en cuenta el rango 0-19. Considerando la nomenclatura: Parámetro {(rango)->reeclasificación a 0 o 1, ...}, se definen como **A** {(0-4)-> 1, (5-14)-> 0,(15-19)-> 1};**B** {(0-6)-> 1, (7-12)-> 0,(13-19)-> 1};**C** {(0-9)-> 1, (10)-> 0,(11-19)-> 1}

En la figura 44 tenemos representamos los valores *f1-score*, *precision* y *recall* de las muestras y del global. Teniendo en cuenta que nuestro rango de posición es de 0 a 19, hemos probado el parámetro A, el cual es de 0 a 4 y de 15 a 19 para la posible reclasificación a 1 de las muestras y de 5 a 14 para 0 (en cada rango los números extremos están incluídos). El parámetro B responde a los rangos 0 a 6 y 13 a 19 para reclasificar a etiqueta 1 y 7 a 11 para etiqueta 0. El parámetro C sólo considera el reetiquetado a 0 para las muestras que ocupan la posición 10, para el resto se considerará reetiquetarlas a 1.

En los resultados se muestra que nuestro *f1-score* global (80%) ha descendido, como también el de la etiqueta 1 (21.41%); el de la etiqueta 0 sigue por encima del 90%. Lo que nos llama la atención es el valor de las nuevas *precision* y *recall*. Mientras que la primera tiene un valor de 14.87%, la segunda ha aumentado al 48.72%, lo cual es una mejora sobre este valor importante. No obstante, el sistema en global es peor, y esto se debe a que con este postproceso hemos forzado más el

conocimiento a priori de las muestras y hemos reetiquetado más muestras como 1. Nuestra precisión es menor ya que nos equivocamos más al reetiquetar como 1, pero también hemos ganado mayor cantidad de muestras 1 partidas, lo cual queda reflejado en el valor de *recall*. Mirando números, en este test habían 508 muestras etiquetadas como palabras partidas. Sin postproceso, de estas hemos encontrado 167 y nos hemos equivocado al etiquetar como clase 1 con 687, mientras que usando postproceso hemos encontrado 281 muestras; sin embargo, esta vez nos hemos equivocado 1934 veces. Ha habido una diferencia de 1245 muestras mal etiquetadas para conseguir etiquetar bien 114.

A pesar de que la *f1-score* del postprocesado es menor que la del modelo sin ninguna clase de postproceso esto no significa que sea peor, ya que puede que los datos, al tener un *recall* mayor, se vean mejor representados. Así que, por último, generaremos dos muestras etiquetadas automáticamente gracias al conocimiento adquirido en visión artificial cruzando una imagen con nuestra base de datos etiquetada generada después de las muestras.

La imagen escogida por la cantidad de etiquetas de la clase 1 contenidas es la imagen 47 de nuestro libro. Se le aplicará el etiquetado automático del modelo generado a partir del algoritmo árbol de decisión y del postprocesado de éste. En la figura 46 tenemos la imagen original etiquetada para el test, en la figura 47 tenemos la imagen etiquetada automáticamente sin postproceso y en la figura 48 con postproceso. Las ventanas con etiqueta 1 en las imágenes etiquetadas automáticamente se marcan con un borde negro.

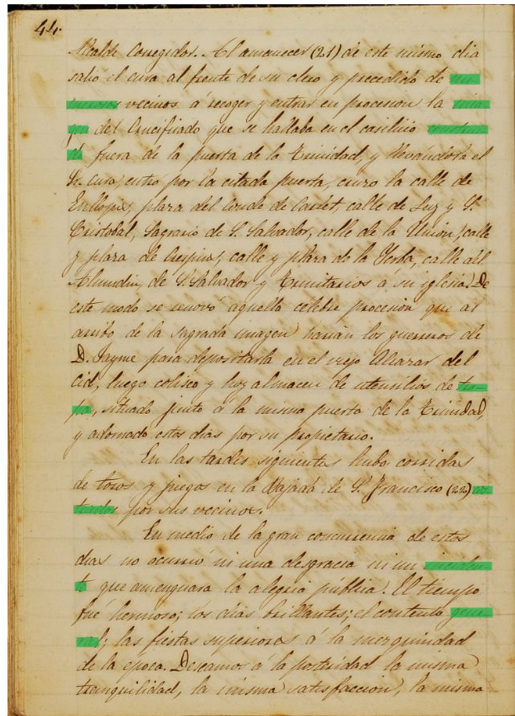


Figura 46. Imagen original etiquetada para la fase de test del modelo.

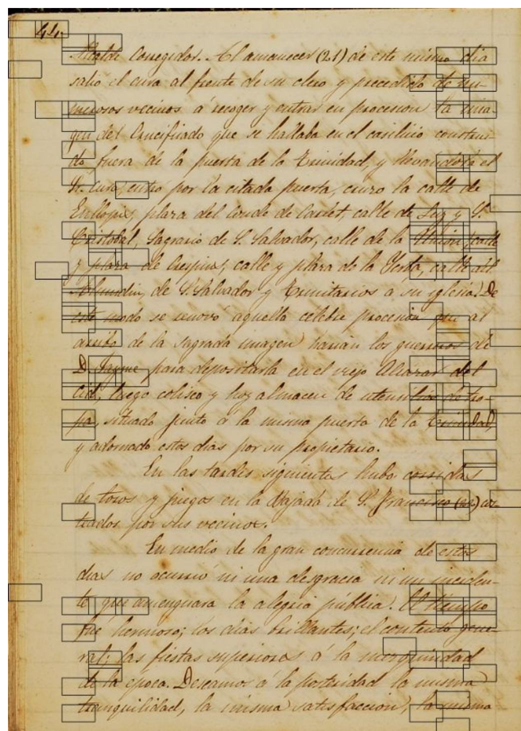


Figura 47. Imagen etiquetada automáticamente a partir del modelo generado con el algoritmo árbol de decisión.

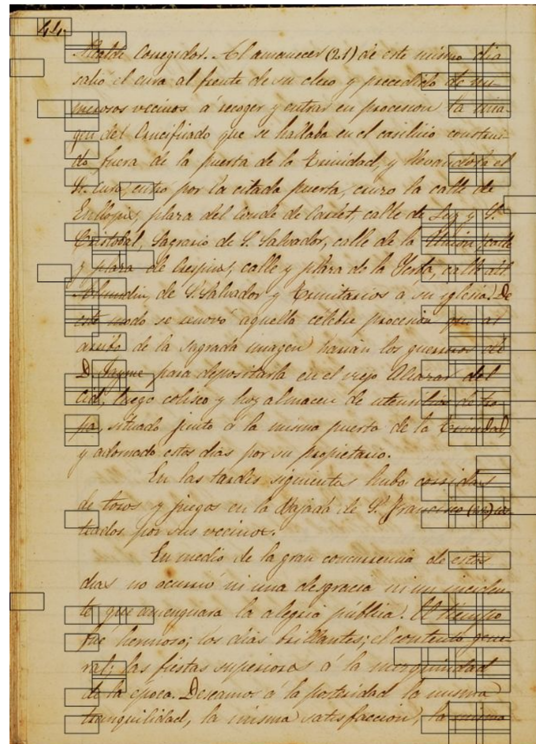


Figura 48. Imagen etiquetada automáticamente a partir del modelo generado con el algoritmo árbol de decisión, aplicando postproceso.

Al observar las imágenes etiquetadas automáticamente (figuras 47 y 48) se corrobora lo que hemos comentado anteriormente; y es que se aumenta la cantidad de aciertos al etiquetar palabras partidas como 1 pero perdemos precisión de etiquetado, lo que se resume en prácticamente marcar muchísimas muestras y no dejar clara la solución. Al observar más de cerca las muestras nos damos cuenta de que se acumulan muchas etiquetas en palabras que no contienen palabras partidas y se dejan de etiquetar muchas ventanas que contienen palabras partidas. Se podría estudiar un nuevo postproceso en el cual cambiamos etiquetado a todo el borde derecho, es decir, si la posición de la etiqueta en cuanto a desplazamiento a derechas es mayor que 16 por ejemplo, su etiqueta se invierte. Observando las figuras 47 y 48 puede ser que el *recall* se quede parecido pero ganemos mucho en *precision*. Esta idea se queda para futuras implementaciones.

7. Conclusiones

Reconocer patrones en imágenes siempre es una tarea complicada, pero dependiendo de la naturaleza de los datos puede disminuir o aumentar esta dificultad. Al tener que reconocer palabras partidas en textos manuscritos antiguos, nuestros datos han sido del tipo de los que aumenta las dificultades, ya que, como hemos visto, no es tarea fácil conseguir una manera homogénea de conseguir muestras. Hay muchas trabas: las ventanas no quedan perfectamente alineadas con las palabras, hay mucho ruido de formas que no son palabras, la muestra no siempre tiene la misma definición en la forma de la palabra, etc.

Dadas las circunstancias, los clasificadores desarrollados no son muy buenos. Su rendimiento global es bueno, y el etiquetado de las muestras que no contienen palabras partidas también. Pero esto cambia al clasificar ventanas que contengan palabras partidas, ya que lo máximo que hemos conseguido es acertar el 50% de las muestras, con un error demasiado alto como para considerarlo bueno.

Han sido muchos los problemas abordados, desde cómo crear las muestras y sacarle el máximo rendimiento posible con técnicas de escalado de grises, extracción de características y estandarización de las muestras, así como cómo utilizar las herramientas elegidas para el uso de los algoritmos de aprendizaje automático seleccionados y cómo aplicar técnicas de postprocesado. Pero a todos estos problemas se ha ido encontrando solución.

Considero los objetivos cumplidos, ya que hemos ganado el conocimiento buscado en cómo abordar un problema de aprendizaje automático y visión artificial y hemos aprendido sobre herramientas del tema.

Nos hemos desenvuelto muy bien en la librería *opencv* y hemos ganado muchos conocimientos de cómo usar las librerías de *scikit-learn.* La lástima ha sido no poder profundizar más por cuestiones de coste de recursos (en el equipo donde se ha realizado este proyecto habían algoritmos que acababan abruptamente por la falta de RAM). No obstante, se ha extendido la base que se tenía sobre la materia convirtiéndola en una fortaleza a la hora de enfrentarnos al mundo laboral.

Hemos visto que hemos sido capaces de desarrollar una solución a un problema desde cero. Se ha aprendido cómo fabricar una base de datos, generar clasificadores para éstos y entender el comportamiento de los datos. Ha sido muy útil toda la labor realizada ya que, a pesar de que cada problema es completamente diferente en esta rama de la computación, tenemos la certeza de que conocemos cómo estructurar el problema y buscar una solución con cualquier herramienta parecida a las empleadas, ya que también se ha ganado en experiencia a la hora de entender librerías a partir de la documentación de éstas.

7.1. Relación del trabajo desarrollado con los estudios cursados

Este trabajo ha ayudado a reforzar unas bases mostradas a lo largo del grado. Desde la decisión de organización y estructuración de código para desarrollar una base de datos hasta pararse a pensar qué tecnologías son las que necesitamos y cómo buscarlas.

Cuando se empezó a plantear el tema que plantea este TFG, acudimos a los conocimientos obtenidos en asignaturas que nos han mostrado paradigmas de la programación, sistemas operativos para ejecutar las pruebas y algo implícito conseguido a lo largo de los años en el grado, que es buscar información a lo largo de la red. Gracias a estos conocimientos hemos podido localizar cómo tratar el problema y qué herramientas pueden ser las mejores tanto como para el visionado artificial como para el aprendizaje automático de modelos para clasificar muestras. En este proyecto se ha usado el conocimiento adquirido por las siguientes competencias transversales: Comprensión e integración, Aplicación y pensamiento práctico, Análisis y resolución de problemas, Innovación, creatividad y emprendimiento, Diseño y proyecto, Pensamiento crítico, Conocimiento de problemas contemporáneos, Aprendizaje permanente y Planificación y gestión del tiempo.

8. Trabajos futuros

En este proyecto se han usado técnicas de visión artificial y algoritmos de aprendizaje automático con cuatro técnicas: k-vecinos más cercanos, máquina de vectores soporte con cuatro kernels diferentes (lineal, polinomial, gaussiano y sigmoid), árboles de decisión y k-medias (los tres primeros del tipo aprendizaje supervisado y el último de aprendizaje no supervisado). Se optó por no investigar en profundidad estos algoritmos ya que dada la naturaleza de los datos (gran cantidad de datos y alto número de características) era inviable por coste computacional y/o temporal. También quedarían por estudiar las redes neuronales profundas [32] para ver cómo se comportan con estos datos.

La creación de las muestras es un proceso clave. Se propone investigar otras técnicas y métodos para intentar eliminar el ruido de éstas. Un ejemplo de ruido son las figuras 35 y 36 en la página 44, en donde se aprecia una muestra (tercera muestra empezando por la izquierda) que contiene una figura que se considerará a la hora de entrenar el modelo y clasificar las muestras, y ésta no es ninguna figura relevante, es una mancha de antigüedad en la página escaneada. También queda pendiente encontrar una mejor solución a la hora de encontrar el borde de las palabras, ya que hay muestras donde aparecen las palabras manuscritas en un tono más marcado que en otras, por lo que hay muestras en las que no se recupera bien la silueta de la palabra.

En estos futuros trabajos hay que evitar usar técnicas de separación lineal, ya que al tener tantas muestras y tantas características por muestra, lo más probable es que estas muestras no sean separables linealmente y puede ser una pérdida de recursos.

9. Referencias

[1] Guerrero, Teresa. G.Lucio, Cristina (2019).”Un nuevo aliado de los médicos”. Periódico: El mundo.

[2] Maisueche Cuadrado, Alberto (2019). “Utilización del Machine Learning en la industria 4.0”, Universidad de Valladolid. Escuela de Ingenierías Industriales. 2019

[3] Amadoz, Sergio. (2019). “Los ojos de los vehiculos autónomos, mejores que los de los seres humanos”. Períodico: La nación

[4] Diferentes codificaciones de color digital:

https://negliadesign.com/ask-a-designer/whats-the-difference-between-pms-cmyk-rgb-and-hex/

[5] “La visión por comuptador, una disciplina en auge”. 2012. Universitat oberta de catalunya.

[6]Conjunto de datos flor iris obtenido en

https://es.wikipedia.org/wiki/Conjunto_de_datos_flor_iris

[7]Conjunto de datos de números manuscritos obtenidos en

https://en.wikipedia.org/wiki/MNIST_database

[8] Bishop, Christopher (2006) Pattern Recognition and Machine Learning.

[9] Ian H. Witten, Eibe Frank, Mark A. Hall (2011) Data Mining: Practical Machine Learning Tools and Techniques.

[10] Leif E.Peterson (2009) K-nearest neighbor.

[11] Statnikov, Alexander; Hardin, Douglas; & Aliferis, Constantin; (2006); "Using SVM weight-based methods to identify causally relevant and non-causally relevant variables"

[12] Hofmann, Thomas; Scholkopf, Bernhard; Smola, Alexander J. (2008). "Kernel Methods in Machine Learning"

[13] Rokach, Lior; Maimon, O. (2008); *Data mining with decision trees: theory and applications*. World Scientific Pub Co Inc

[14] Hamerly, G. and Elkan, C. (2002). «Alternatives to the k-medias algorithm that find better clusterings». *Proceedings of the eleventh international conference on Information and knowledge management (CIKM*)

[15] K. Terasawa and Y. Tanaka,(*2009*). "Slit Style HOG Feature for Document Image Word Spotting," *10th International Conference on Document Analysis and Recognition*, Barcelona, 2009, pp. 116-120, doi: 10.1109/ICDAR.2009.118.

[16] J. A. Sánchez, V. Romero, A. H. Toselli and E. Vidal, (*2016).* "ICFHR2016 Competition on Handwritten Text Recognition on the READ Dataset," *15th International Conference on Frontiers in Handwriting Recognition (ICFHR)*, Shenzhen, 2016, pp. 630-635, doi: 10.1109/ICFHR.2016.0120.

[17] Documentación de *scikit-learn* consultada en https://scikit-learn.org/stable/

[18] Documentación de *numpy* consultada en https://numpy.org/

[19] Información sobre el formato de entrada de los algorítmos obtenida en https://scikit-learn.org/stable/modules/preprocessing.html y https://scikit-learn.org/stable/modules/feature_extraction.html

[20] Documentación de *opencv* consultada en https://opencv.org/

[21] Documentación pickle consultada en https://docs.python.org/3/library/pickle.html

[22] Pastor, Javier (2019). Cómo tener un kernel de Linux en Windows 10. Revista: Xataka.

[23] Información obtenida en https://www.pyimagesearch.com/2014/08/04/opencv-python-color-detection/ por Adrian Rosebrock. 2014.

[24] Shetty, Badreesh (2020). “Curse of dimensionality”

[25] Juszczak, P.; D. M. J. Tax; R. P. W. Dui (2002). "Feature scaling in support vector data descriptions".

[26] Imola K.Fodor (2002). “A Survey of Dimension Reduction Techniques”. Center for Applied Scientific Computing, Lawrence Livermore National Laboratory. California (CA 94551)

[27] Shlens, Jonathon. Smith, Lindsay. (2002). .A Tutorial on Principal Component Analysis.

[28] Powers, David M W (2011). "Evaluation: From Precision, Recall and F-Measure to ROC, Informedness, Markedness & Correlation"

[29] D. Anguita, A. Ghio, N. Greco, L. Oneto and S. Ridella (2010), "Model selection for support vector machines: Advantages and disadvantages of the Machine Learning Theory," *The 2010 International Joint Conference on Neural Networks (IJCNN)*.

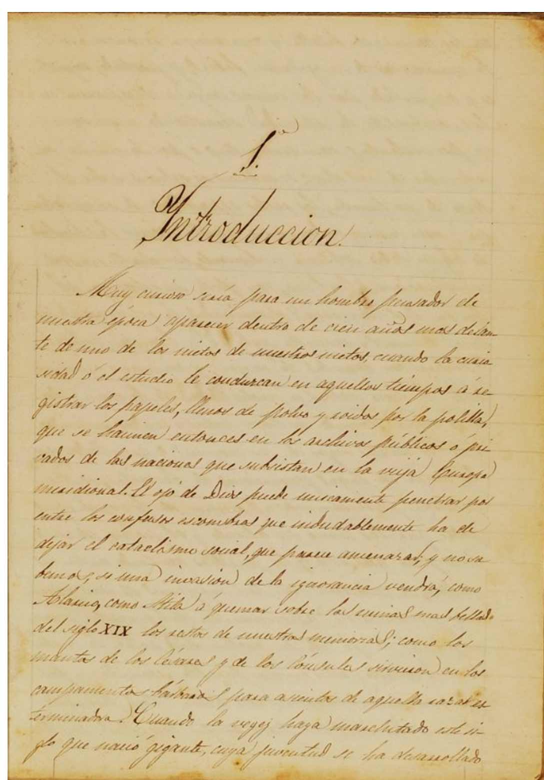
[30] Zita, Ana, “Exactitud y precisión”, en: Diferenciador

[31] Santhini, Marina.(2016) “Advantages & Disadvantages of k--Means and Hierarchical clustering (Unsupervised Learning)”, Learning for Language Technology ML4LT , Department of Linguistics and Philology Uppsala University

[32] Aldabas-Rubira, Emiliano (2002). Introducción al reconocimiento de patrones mediante redes neuronales. UPC-Campus Terrassa-DEE-EUETIT. Terrassa Barcelona

APÉNDICE I

En este apéndice se recogen una parte de las imágenes donde se quieren reconocer palabras partidas. El corpus entero de imágenes contiene 53 páginas del libro ‘Noticia histórica de las fiestas con que Valencia celebró el siglo sexto de la venida a esta capital de la milagrosa imagen del salvador por D.Vicente Boix, cronista de la misma ciudad, año 1853’ escaneadas a color.



1.

Introduccion

Muy curioso sería para un hombre pensador de
nuestra época aparecer dentro de cien años mas delan-
te de uno de los nietos de nuestros nietos, cuando la curio-
sidad ó el estudio le condujeran en aquellos tiempos á re-
gistrar los papeles, llenos de polvo y roidos por la polilla,
que se hallen entonces en los archivos públicos ó pri-
vados de las naciones que subsistan en la vieja Europa
meridional. El ojo de Dios puede únicamente penetrar por
entre los confusos escombros que indudablemente ha de
dejar el cataclismo social, que parece amenazarnos, y no sa-
bemos, si una invasión de la ignorancia vendrá, como
Alarico, como Atila á quemar sobre las ruinas mas bellas
del siglo XIX los restos de nuestras memorias; como los
mantos de los Césares y de los Cónsules sirvieron en los
campamentos bárbaros para asientos de aquella raza ex-
terminadora. Cuando la vejez haya marchitado este si-
glo que nació gigante, cuya juventud se ha desarrollado

Página 4 del libro

página 16 del libro. Aproximadamente un 80% de las páginas son como esta.

página 50 del libro