Práctica Final (Mushroom Classifier)

¿Cómo he planteado la práctica?

Primero necesitamos descargar el dataset de Kaggle, en este caso, el dataset escogido es 'Mushroom clasification', que se puede descargar de:

https://www.kaggle.com/uciml/mushroom-classification

Tras ello, necesitaremos entender el dataset, para ello, estudiaremos la cabecera y su primera línea:

class	cap-shape	cap-surface	cap-color	bruises	odor	gill-attachment	gill-spacing	gill-size	…
P	X	S	N	T	P	F	C	N	…

…	gill-color	stalk-shape	stalk-root	stalk-surface-above-ring	stalk-surface-below-ring	stalk-color-above-ring	stalk-color-below-ring	veil-type	…
…	K	E	E	S	S	W	W	P	…

…	veil-color	ring-number	ring-type	spore-print-color	population	habitat
…	W	O	P	K	S	U

- **class**: La clase a la que pertenece, en este caso es nuestra 'Y'
  - e: edible → comestible
  - p: poisonous → venenosa

- **cap-shape**: Forma del sombrero o píleo
  - b: bell → campana
  - c: conical → cónica
  - x: convex → convexa
  - f: flat → plana
  - k: knobbed → nudosa
  - s: sunken → hundida

- **cap-surface**: Superficie del sombrero o píleo
  - f: fibrous → fibrosa
  - g: grooves → surcosa
  - y: scaly → escamosa
  - s: smooth → lisa

- **cap-color**: Color del sombrero o píleo
  - n: brown → marrón
  - b: buff → ante
  - c: cinnamon → canela
  - g: gray → gris
  - r: green → verde
  - p: pink → rosa
  - u: purple → morado
  - e: red → roja
  - w: white → blanca
  - y: yellow → amarilla

- **bruises**: Decoloración
  - t: true → tiene decoloración
  - f: false → no tiene decoloración

- **odor**: Olor
  - a: almond → almendrado
  - l: anise → anís
  - c: creosote → creosota
  - y: fishy → pescado
  - f: foul → mal olor
  - m: musty → rancio
  - n: none → ninguno
  - p: pungent → acre
  - s: spicy → picante

- **gill-attachment**: Forma de las láminas
  - a: attached → adherente
  - ~~d: descending → decurrente~~ NO EN EL DATASET
  - f: free → libre
  - ~~n: notched → sinuosa / con muescas~~ NO EN EL DATASET

- **gill-spacing**: Distancia de las láminas
  - c: close → cercana
  - w: crowded → apilada
  - ~~d: distant → distante~~ NO EN EL DATASET

- **gill-size**: Tamaño de las láminas
  - b: broad → amplia
  - n: narrow → estrecha

- **gill-color**: Color de las láminas
  - k: black → negro
  - n: brown → marrón
  - b: buff → ante
  - h: chocolate → chocolate

  - g: gray → gris
  - r: green → verde
  - o: orange → naranja
  - p: pink → rosa
  - u: purple → morado
  - e: red → rojo
  - w: white → blanco
  - y: yellow → amarillo

- **stalk-shape**: Forma del tallo
  - e: enlarging → ensanchamiento
  - t: tampering → estrechamiento

- **stalk-root**: Raíz del tallo
  - b: bulbous → bulboso
  - c: club → mazo
  - ~~u: cup → copa~~ NO EN EL DATASET
  - e: equal → igual
  - ~~z: rhizomorphs → rizomorfo~~ NO EN EL DATASET
  - r: rooted → arraigada
  - ?: missing → desconocida

- **stalk-surface-above-ring**: Superficie del tallo sobre el anillo
  - f: fibrous → fibrosa
  - y: scaly → escamosa
  - k: silky → sedosa
  - s: smooth → lisa

- **stalk-surface-below-ring**: Superficie del tallo bajo el anillo
  - f: fibrous → fibrosa
  - y: scaly → escamosa
  - k: silky → sedosa
  - s: smooth → lisa

- **stalk-color-above-ring**: Color del tallo sobre el anillo
  - n: brown → marrón
  - b: buff → ante
  - c: cinnamon → canela
  - g: gray → gris
  - o: orange → naranja
  - p: pink → rosa
  - e: red → roja
  - w: white → blanca
  - y: yellow → amarilla

- **stalk-color-below-ring**: Color del tallo bajo el anillo
  - n: brown → marrón
  - b: buff → ante
  - c: cinnamon → canela
  - g: gray → gris
  - o: orange → naranja
  - p: pink → rosa
  - e: red → roja
  - w: white → blanca
  - y: yellow → amarilla

- **veil-type**: Tipo de velo
  - p: partial → parcial
  - ~~u: universal → universal~~ NO EN EL DATASET

- **veil-color**: Color de velo
  - n: brown → marrón
  - o: orange → naranja
  - w: white → blanco
  - y: yellow → amarillo

- **ring-number**: Número de anillos
  - n: none → ninguno
  - o: one → uno
  - t: two → dos

- **ring-type**: Tipo de anillo
  - ~~c: cobwebby → telaraña~~ NO EN EL DATASET
  - e: evanescent → evanescente
  - f: flaring → llameante
  - l: large → largo
  - n: none → ninguno
  - p: pendant → pendiente
  - ~~s: sheathing → revestimiento~~ NO EN EL DATASET
  - ~~z: zone → zona~~ NO EN EL DATASET

- **spore-print-color**: Color de las esporas
  - k: black → negro
  - n: brown → marrón
  - b: buff → ante
  - h: chocolate → chocolate
  - r: green → verde
  - o: orange → naranja
  - u: purple → morado
  - w: white → blanca
  - y: yellow → amarilla

- **population**: Población

  - a: abundant → abundante
  - c: clustered → agrupada
  - n: numerous → numerosa
  - s: scattered → dispersa
  - v: several → severa
  - y: solitary → solitaria

- **habitat**: Hábitat
  - g: grasses → hierba
  - l: leaves → hojas
  - m: meadows→ prados
  - p: paths → caminos
  - u: urban → urbano
  - w: waste → desechos
  - d: woods → madera

Por ello, sabemos que la primera fila se traduce en:

p,x,s,n,t,p,f,c,n,k,e,e,s,s,w,w,p,w,o,p,k,s,u

Hongo venenoso, su sombrero es convexo, con superficie lisa y de color marrón, además, ofrece decoloración, tiene un olor acre y sus láminas tienen formas libres, además las láminas tienen una distancia corta y son de tamaño pequeño y de color negro. Su tallo se va ensanchando y su raíz tiene la misma forma. Respecto a la superficie por encima del anillo es lisa y de color blanco, al igual que por debajo del anillo. El velo es parcial y de color blanco, además, tiene un anillo pendiente y sus esporas son de color negro . El hongo suele estar disperso de otros hongos de su especie, y se encuentra en un ecosistema urbano.

Con estos datos es interesante clasificar una seta en comestible o venenosa a partir del resto de características.

Por ello, como es una tarea de clasificación, primero implementaré la solución de tres formas distintas:

- Regresión logística multivariable
- Redes neuronales
- SVM

Objetivos

- Comparar los distintos tipos de clasificadores.
- Comparar las distintas arquitecturas de cada clasificador.

Metodología

La metodología será la misma para entrenar cada clasificador:

1. Leer los datos del CSV.
   1.1. Hacer 'shuffle'.
   1.2. Dividir las X e Y.
   1.3. Pasar los datos de strings a enteros.
      1.3.1. Se puede usar onehot para las Y.
   1.4. Añadir la columna de 1's.

2. Dividir los datos en Train set, CV set y Test set.

3. Asignar los parámetros del modelo (Por ejemplo, el número de nodos de la hidden layer).

4. Entrenar el modelo con distintos parámetros para obtener el modelo óptimo.

5. Mostrar diferentes gráficas comparativas respecto a los distintos tipos de arquitectura propuestos en el clasificador (Modelo de 10 capas vs modelo de 25 capas vs…)

6. Guardar el modelo óptimo entrenado

Respecto a la metodología de uso del programa, será:

1. Seleccionar el tipo de clasificador.

2. Reentrenar el clasificador*.

3. Probar manualmente un ejemplo.

** NOTA: Reentrenar el clasificador con un distinto número de capas en una red neuronal, o en las SVM con valores C muy altos, conlleva mucho tiempo, por ello he decidido que, en lugar de probar todas las combinaciones, que el clasificador termine de entrenar en el momento que se llega al 100% de precisión. Además, al terminar de reentrenar la red, se mostrará el tiempo (segundos) que ha tardado en procesar el entrenamiento.*

Archivos y sus funciones:

- **source**: Menú que gestiona las llamadas a distintos archivos, permite, a través de la línea de comandos:
    - Seleccionar el tipo de clasificador.
    - Reentrenar dicho clasificador.
    - Probar manualmente un ejemplo.

- **logistic_regression**: Archivo que implementa la funcionalidad de una regresión logística, permite:
    - Calcular el coste y la gradiente regularizadas
    - Dibujar el coste en función de las λ
    - Calcular la precisión del clasificador
    - Obtener las θ óptimas
        - Sin regularización
        - Con regularización
    - Guardar las θ
    - Cargar las θ
    - Predecir un único ejemplo

- **neural_network**: Archivo que implementa la funcionalidad de una red neuronal, permite:
    - Aplicar forward y back propagation
    - Calcular el coste y la gradiente regularizadas
    - Establecer la arquitectura de la red automáticamente
    - Inicializar las θ aleatorias
    - Entrenar el modelo
    - Calcular la precisión del clasificador
    - Dibujar la precisión en función de las lambdas
    - Dibujar la evolución del error de aprendizaje (Train set vs CV set)
    - Obtener las θ óptimas
        - Con diferente número de capas
        - Con diferentes λ
        - Con diferentes iteraciones
    - Guardar las θ
    - Cargar las θ
    - Predecir un único ejemplo

- **svm**: Archivo que implementa la funcionalidad de una máquina de soporte vectorial, permite:
    - Obtener la mejor combinación de parámetros C y S basándonos en la precisión del clasificador
    - Calcular la precisión del clasificador
    - Guardar las θ
    - Cargar las θ
    - Predecir un único ejemplo

- **common_functions:** Archivo que permite centralizar funciones repetidas de los modelos:
    - Función sigmoide

- **dataset_functions**: Archivo que permite centralizar funciones de lectura y modificación de datos, como:
    - Lectura del dataset
    - Implementación de Onehot
    - Aleatorizar la posición de los datos (Shuffle)
    - Pasar los datos de string a int
    - Codificar un ejemplo
    - Dividir el dataset en:
        - $TRAIN \rightarrow 80\%$
        - $CV \rightarrow 10\%$
        - $TEST \rightarrow 10\%$

Aclaraciones

- **¿Por qué no implemento regresión lineal?**
  La regresión lineal se usa, normalmente, para predecir un resultado, por ejemplo, el precio de una casa, es decir, sus valores se basan en una recta y nosotros buscamos clasificar entre 0 y 1, y este tipo de regresión puede tener valores mayores y/o menores a este rango. Puede ser multivariable y se podría implementar, pero los resultados obtenidos no serían tan buenos como el resto de clasificadores, por lo que he decidido obviar este tipo de implementación.

- **¿Por qué implemento estos tres clasificadores?**
  Son clasificadores que hemos implementado durante todo el curso, además son los más conocidos y se ajustan bien a la tarea de clasificación que quiero implementar.
    - Regresión Logística: Clasifica valores entre 0 y 1
    - Redes Neuronales: Se pueden utilizar como clasificador gracias a las funciones softmax y onehot
    - SVM: Usado para clasificar valores

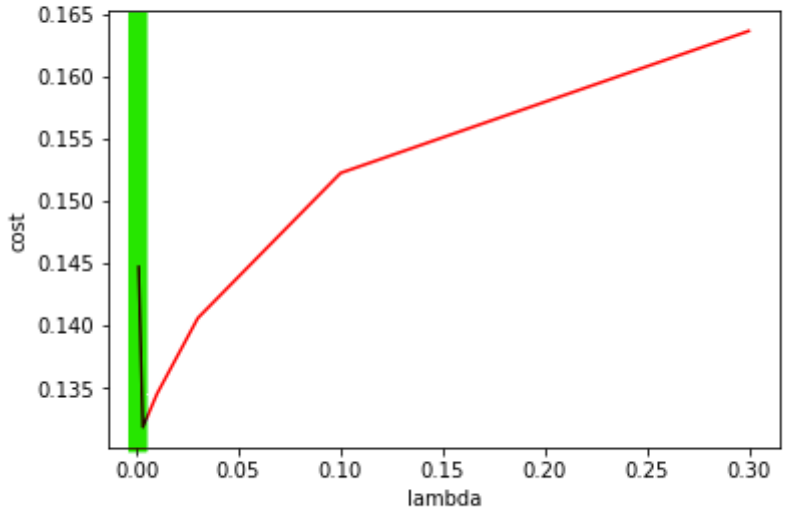
- **¿Por qué no implemento valores polinomiales?**
  Los valores polinomiales se implementan cuando no tenemos suficientes características y se produce un alto sesgo (Los valores no se ajustan a la función), en este caso tenemos 22 características independientes, por lo que considero que son características suficientes, ya que los resultados de los clasificadores están siempre por encima del 95%.

COMPARATIVAS

COMPARATIVA REGRESIÓN LOGÍSTICA

```
LOGISTIC REGRESSION WITHOUT REGULARIZATION
The logistic regression is reliable in 96.31% of the time

LOGISTIC REGRESSION WITH REGULARIZATION
```



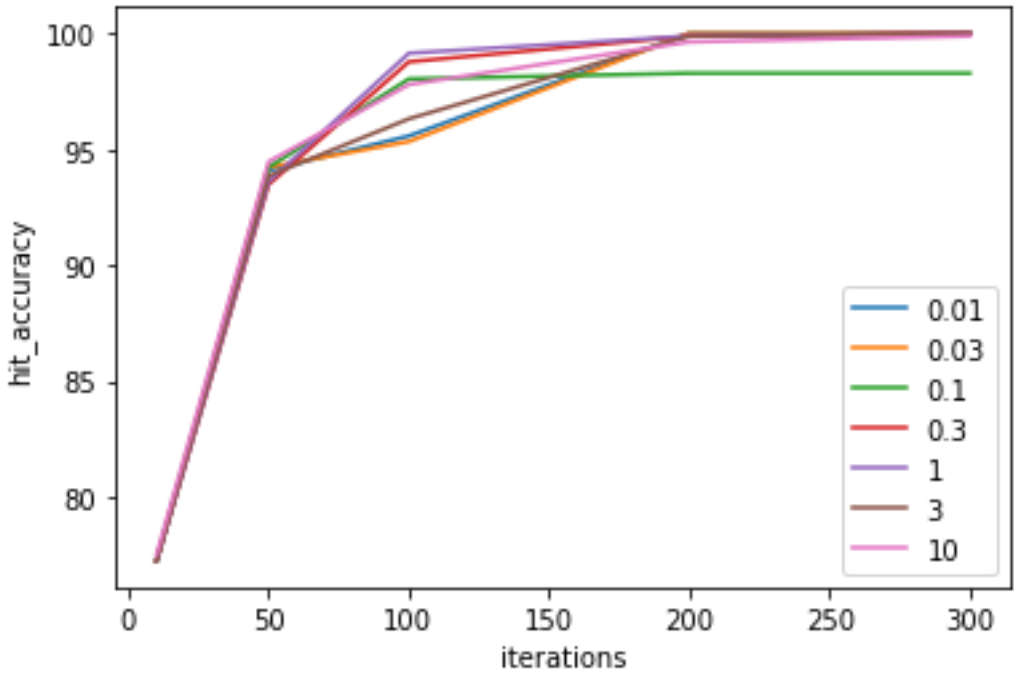

```
The logistic regression is reliable in 96.31% of the time
```

En este caso podemos observar que la regularización no afecta en nada a la precisión del clasificador, como veremos en comparativas posteriores, esto tiene sentido, ya que el clasificador no parece tener problemas de sobreajuste.

COMPARATIVA REDES NEURONALES

10 nodos en capa oculta

		λ						
ITERACIONES		**0,01**	**0,03**	**0,1**	**0,3**	**1**	**3**	**10**
	10	77.24%	77.24%	77.24%	77.24%	77.24%	77.24%	77.49%
	50	94.1%	94.22%	94.22%	93.48%	93.6%	93.85%	94.46%
	100	95.57%	95.33%	98.03%	98.77%	99.14%	96.31%	97.79%
	200	100.0%	100.0%	98.28%	99.88%	99.88%	99.88%	99.63%
	300	100.0%	100.0%	98.28%	100.0%	100.0%	100.0%	99.88%



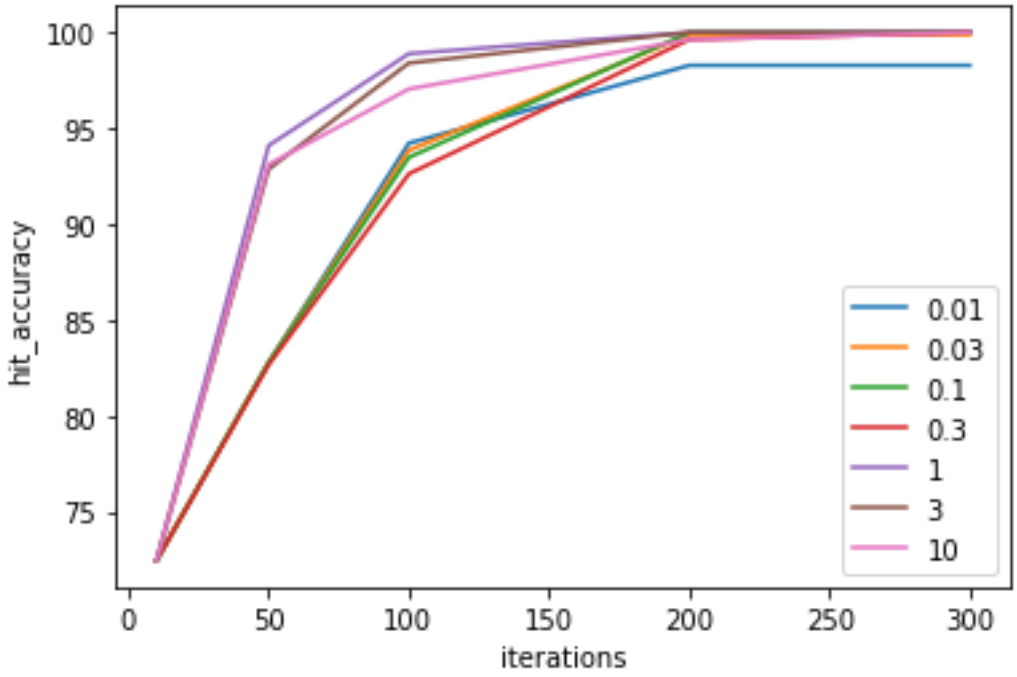
En este caso, y con las θ aleatorias, es muy probable que se llegue al 100% de acierto, ya que con una regularización de 0'01 y 0'03 se obtiene una puntuación perfecta en solo 200 iteraciones.

La peor opción es elegir una regularización de 0'1, se puede observar que el porcentaje de acierto prácticamente no varía entre las 100 y 300 iteraciones.

NOTA: *ESTOS RESULTADOS DEPENDEN DE LAS CARACTERÍSTICAS ALEATORIAS ESCOGIDAS, POR LO QUE PUEDEN VARIAR*

25 nodos en capa oculta

		λ						
ITERACIONES		**0,01**	**0,03**	**0,1**	**0,3**	**1**	**3**	**10**
	10	72.45%	72.45%	72.45%	72.45%	72.45%	72.45%	72.45%
	50	82.78%	82.78%	82.78%	82.66%	94.1%	92.87%	93.11%
	100	94.22%	93.85%	93.48%	92.62%	98.89%	98.4%	97.05%
	200	98.28%	99.88%	100.0%	99.63%	100.0%	100.0%	99.63%
	300	98.28%	99.88%	100.0%	100.0%	100.0%	100.0%	100.0%



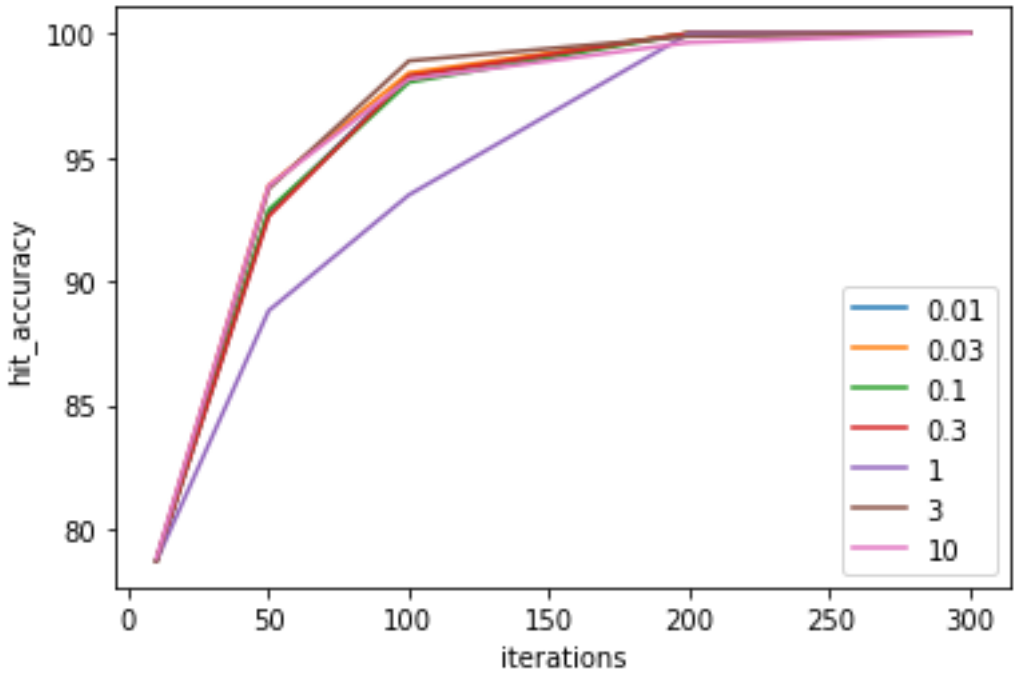
En este caso, y con las θ aleatorias, es muy probable que se llegue al 100% de acierto, ya que con una regularización superior a 0'1 se obtiene una puntuación perfecta en solo 200 iteraciones.

La peor opción es elegir una regularización de 0'01, se puede observar que el porcentaje de acierto no varía entre las 200 y 300 iteraciones, y se queda en un 98% de acierto.

NOTA: *ESTOS RESULTADOS DEPENDEN DE LAS CARACTERÍSTICAS ALEATORIAS ESCOGIDAS, POR LO QUE PUEDEN VARIAR*

50 nodos en capa oculta

		λ						
ITERACIONES		**0,01**	**0,03**	**0,1**	**0,3**	**1**	**3**	**10**
	10	78.72%	78.72%	78.72%	78.72%	78.72%	78.72%	78.84%
	50	92.74%	93.85%	92.87%	92.62%	88.81%	93.73%	93.85%
	100	98.28%	98.4%	98.03%	98.28%	93.48%	98.89%	98.15%
	200	100.0%	100.0%	100.0%	100.0%	100.0%	99.88%	99.63%
	300	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%



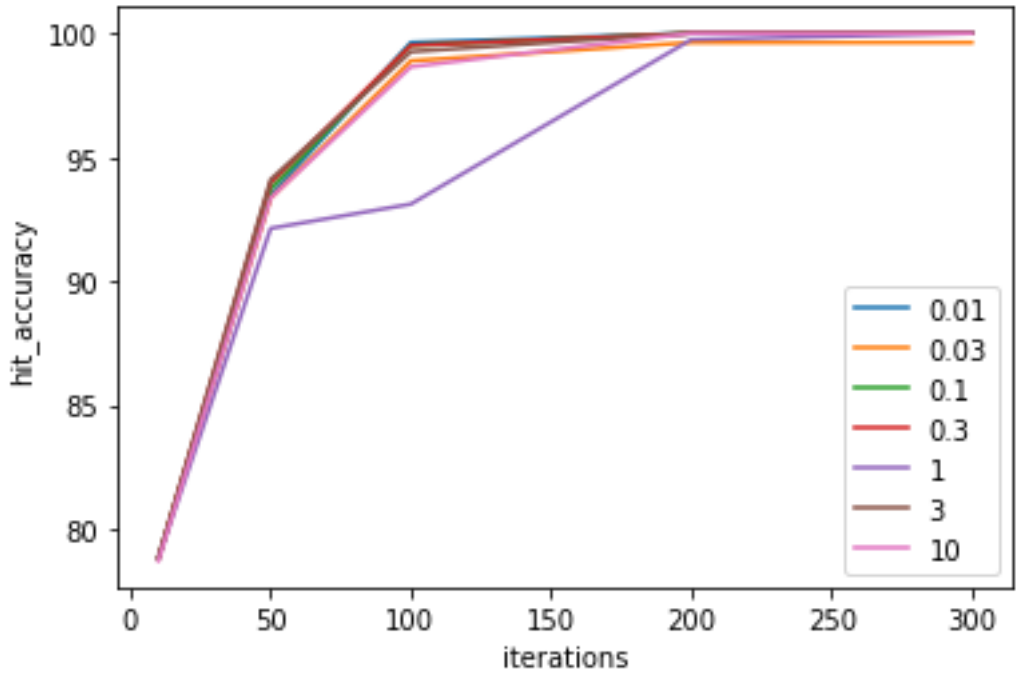
En este caso, y con las θ aleatorias, es seguro que se llegue al 100% de acierto, la mejor opción, si cabe, es la regularización de 0'03, ya que es ligeramente más rápida que las demás.

Las peores regularizaciones son las más grandes, ya que llegan al 100% en 300 iteraciones.

NOTA: *ESTOS RESULTADOS DEPENDEN DE LAS CARACTERÍSTICAS ALEATORIAS ESCOGIDAS, POR LO QUE PUEDEN VARIAR*

75 nodos en capa oculta

		λ						
ITERACIONES		**0,01**	**0,03**	**0,1**	**0,3**	**1**	**3**	**10**
	10	78.84%	78.84%	78.84%	78.84%	78.84%	78.84%	78.72%
	50	93.48%	93.36%	93.73%	93.97%	92.13%	94.1%	93.36%
	100	99.63%	98.89%	99.51%	99.51%	93.11%	99.26%	98.65%
	200	100.0%	99.63%	100.0%	100.0%	99.75%	100.0%	100.0%
	300	100.0%	99.63%	100.0%	100.0%	100.0%	100.0%	100.0%

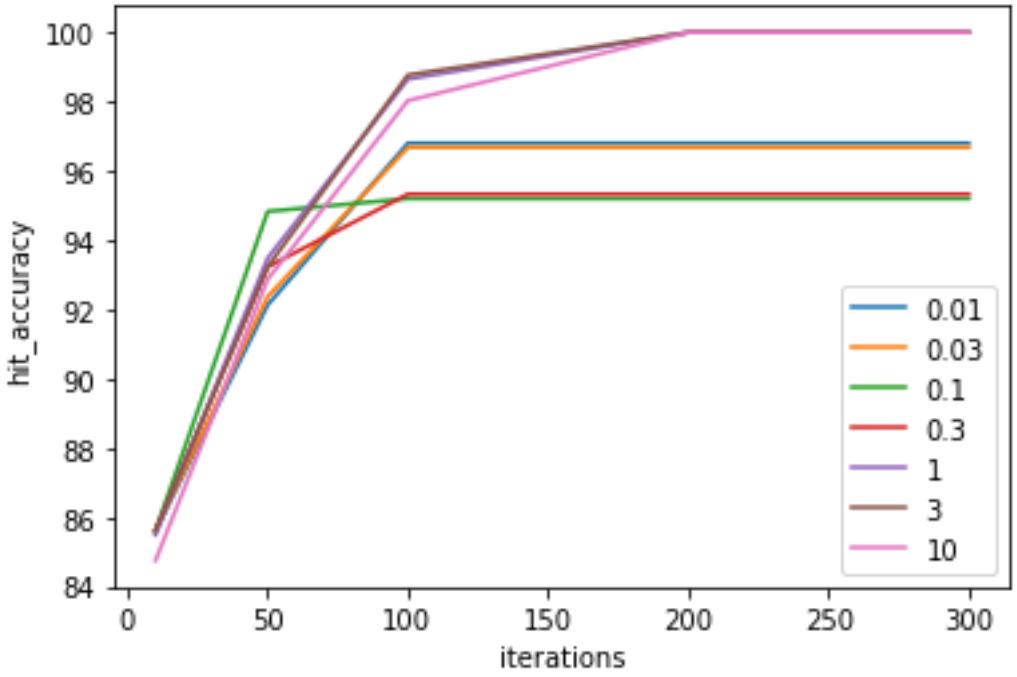


En este caso, y con las θ aleatorias, es muy probable que se llegue al 100% de acierto, todas las regularizaciones son muy buenas, a excepción de la regularización de 0'03, que no llega al 100% de acierto en 300 iteraciones, y la de 1, que llega al 100% de acierto en 300 iteraciones, a diferencia de la demás que llegan en 200 iteraciones.

NOTA: *ESTOS RESULTADOS DEPENDEN DE LAS CARACTERÍSTICAS ALEATORIAS ESCOGIDAS, POR LO QUE PUEDEN VARIAR*

100 nodos en capa oculta

		λ						
ITERACIONES		**0,01**	**0,03**	**0,1**	**0,3**	**1**	**3**	**10**
	10	85.61%	85.61%	85.61%	85.61%	85.49%	85.61%	84.75%
	50	92.13%	92.37%	94.83%	93.23%	93.48%	93.23%	92.87%
	100	96.8%	96.68%	95.2%	95.33%	98.65%	98.77%	98.03%
	200	96.8%	96.68%	95.2%	95.33%	100.0%	100.0%	100.0%
	300	96.8%	96.68%	95.2%	95.33%	100.0%	100.0%	100.0%



En este caso, y con las θ aleatorias, es probable que se llegue al 100% de acierto con una regularización alta, en este caso, podemos observar que cuando existe una regularización superior a la unidad, se obtiene el 100% de acierto en las 200 iteraciones, siendo la mejor opción, la regularización de 3.

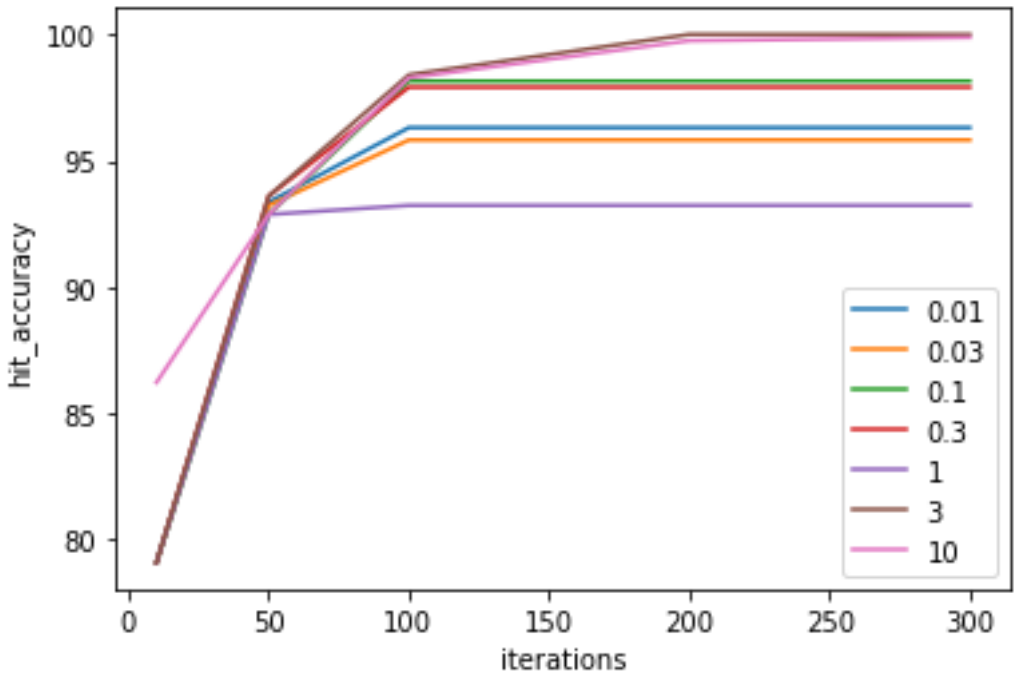
Las peores regularizaciones son las de 0'1 y 0,3, ya que apenas llegan al 95% de acierto.

Además, como dato curioso, las cuatro regularizaciones más bajas han tenido el mejor resultado en las 10 primeras iteraciones, y al llegar a las 300 iteraciones han sido las que peor resultado han tenido.

NOTA: *ESTOS RESULTADOS DEPENDEN DE LAS CARACTERÍSTICAS ALEATORIAS ESCOGIDAS, POR LO QUE PUEDEN VARIAR*

150 nodos en capa oculta

		λ						
		0,01	**0,03**	**0,1**	**0,3**	**1**	**3**	**10**
ITERACIONES	**10**	79.09%	79.09%	79.09%	79.09%	79.09%	79.09%	86.22%
	50	93.36%	93.23%	92.87%	93.6%	92.87%	93.6%	92.87%
	100	96.31%	95.82%	98.15%	97.91%	93.23%	98.4%	98.28%
	200	96.31%	95.82%	98.15%	97.91%	93.23%	100.0%	99.75%
	300	96.31%	95.82%	98.15%	97.91%	93.23%	100.0%	99.88%

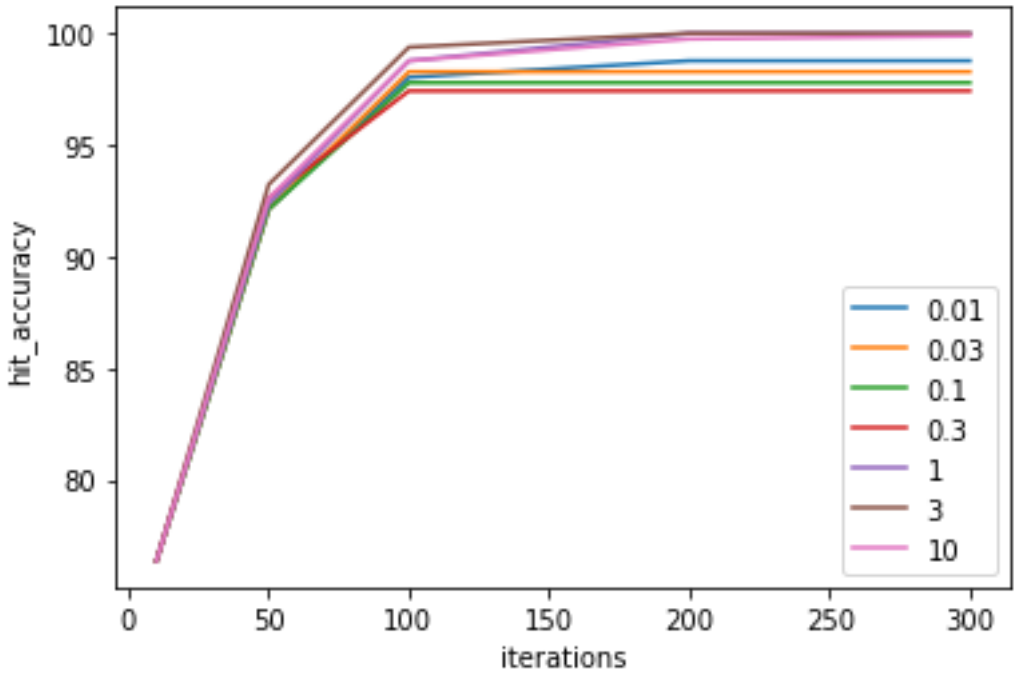


En este caso, y con las θ aleatorias, es poco probable que se llegue al 100% de acierto, y la regularización de 3 es, con diferencia, la regularización que mejores resultados ofrece, siendo la peor regularización la de 1, llegando al 93%.

NOTA: *ESTOS RESULTADOS DEPENDEN DE LAS CARACTERÍSTICAS ALEATORIAS ESCOGIDAS, POR LO QUE PUEDEN VARIAR*

200 nodos en capa oculta

		λ						
ITERACIONES		**0,01**	**0,03**	**0,1**	**0,3**	**1**	**3**	**10**
	10	76.38%	76.38%	76.38%	76.38%	76.38%	76.38%	76.38%
	50	92.13%	92.13%	92.13%	92.62%	92.37%	93.23%	92.62%
	100	98.03%	98.28%	97.79%	97.42%	98.77%	99.38%	98.77%
	200	98.77%	98.28%	97.79%	97.42%	100.0%	100.0%	99.75%
	300	98.77%	98.28%	97.79%	97.42%	100.0%	100.0%	99.88%



En este caso, y con las θ aleatorias, es poco probable que se llegue al 100% de acierto, las regularizaciones altas son las que mejores resultados ofrecen, siendo la mejor elección la regularización de 3, ya que llega al 99% de acierto en 100 iteraciones.

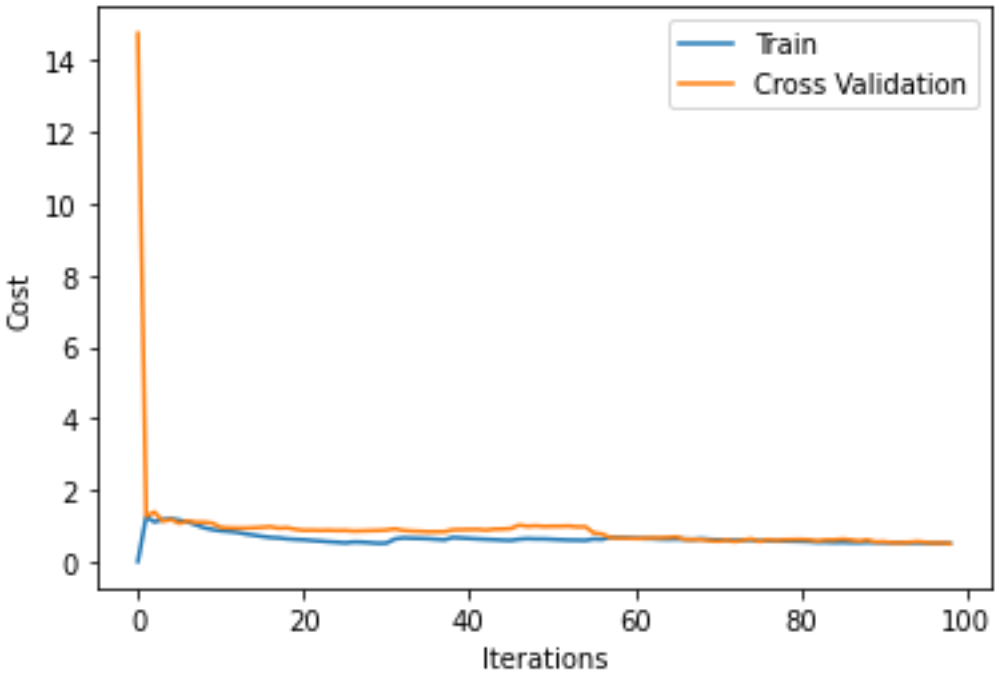
La regularización de 0'3 es la peor regularización, ya que se queda en el 97'4% de acierto en las 300 iteraciones.

NOTA: *ESTOS RESULTADOS DEPENDEN DE LAS CARACTERÍSTICAS ALEATORIAS ESCOGIDAS, POR LO QUE PUEDEN VARIAR*

Comentarios de comparativa de Redes Neuronales

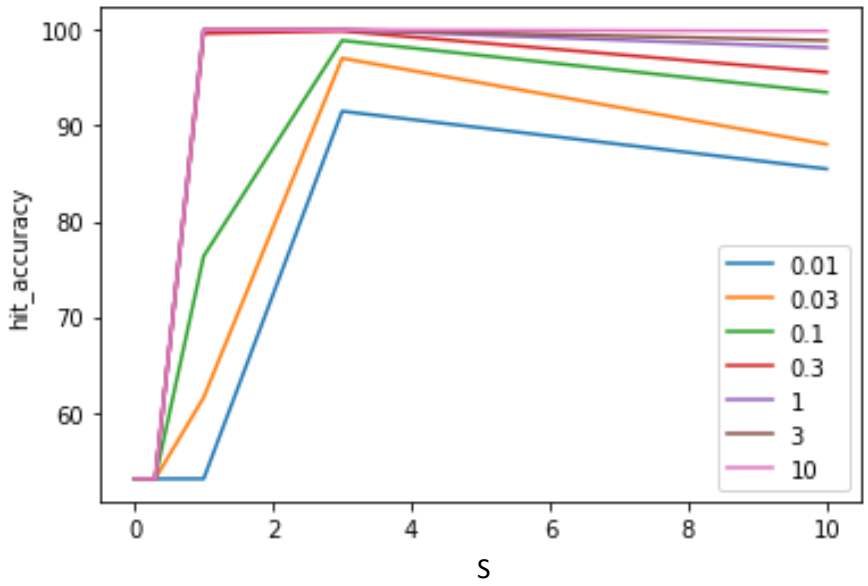
Es interesante probar las distintas combinaciones de parámetros y comparar sus resultados, pero funcionalmente no tiene mucho sentido, ya que en varias ocasiones hemos conseguido el máximo de precisión, por lo cual he decidido que la red neuronal deje de entrenar en el momento que alguna combinación llegue al 100% de acierto.

Como he comentado anteriormente, el dataset está dividido en 80% train set, 10% cross validation set y 10% test set, por lo que ahora deberemos estudiar la evolución del coste respecto a las iteraciones. Como tiene un 100% de acierto, la inteligencia artificial no tiene sesgo ni varianza, por lo que el coste del train set y el cross validation set debe ser muy bajo.



COMPARATIVA SVM RBF (Radial Basis Function)

		S						
C		**0,01**	**0,03**	**0,1**	**0,3**	**1**	**3**	**10**
	0,01	53.08%	53.08%	53.08%	53.08%	53.08%	91.5%	85.47%
	0,03	53.08%	53.08%	53.08%	53.08%	61.58%	97.04%	88.05%
	0,1	53.08%	53.08%	53.08%	53.08%	76.35%	98.89%	93.47%
	0,3	53.08%	53.08%	53.08%	53.08%	99.63%	99.88%	95.57%
	1	53.08%	53.08%	53.08%	53.08%	100.0%	100.0%	98.15%
	3	53.08%	53.08%	53.08%	53.08%	100.0%	100.0%	98.89%
	10	53.08%	53.08%	53.08%	53.08%	100.0%	100.0%	99.88%



En el caso de SVM, es relativamente sencillo ver un patrón, los mejores valores están en C > 0'1 y S > 0'3, además, si los valores de S > 3, la tendencia del acierto aparentemente decrece por lo que:

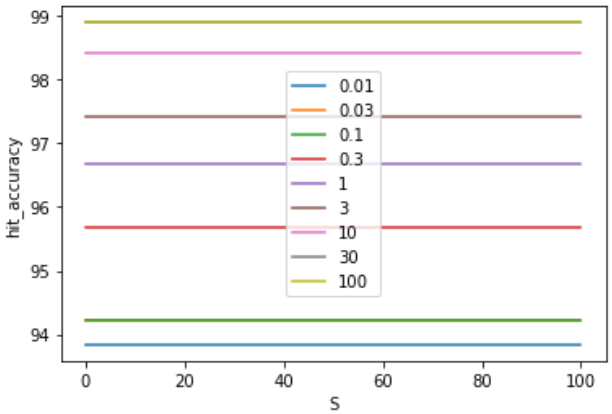
$$1 \leq C \leq 10$$

$$1 \leq S < 10$$

NOTA: *ESTOS RESULTADOS DEPENDEN DE LAS CARACTERÍSTICAS ALEATORIAS ESCOGIDAS, POR LO QUE PUEDEN VARIAR*

COMPARATIVA SVM LINEAL

		S								
C		**0,01**	**0,03**	**0,1**	**0,3**	**1**	**3**	**10**	**30**	**100**
	0,01	93.84%	93.84%	93.84%	93.84%	93.84%	93.84%	93.84%	93.84%	93.84%
	0,03	94.21%	94.21%	94.21%	94.21%	94.21%	94.21%	94.21%	94.21%	94.21%
	0,1	94.21%	94.21%	94.21%	94.21%	94.21%	94.21%	94.21%	94.21%	94.21%
	0,3	95.69%	95.69%	95.69%	95.69%	95.69%	95.69%	95.69%	95.69%	95.69%
	1	96.67%	96.67%	96.67%	96.67%	96.67%	96.67%	96.67%	96.67%	96.67%
	3	97.41%	97.41%	97.41%	97.41%	97.41%	97.41%	97.41%	97.41%	97.41%
	10	98.4%	98.4%	98.4%	98.4%	98.4%	98.4%	98.4%	98.4%	98.4%
	30	98.89%	98.89%	98.89%	98.89%	98.89%	98.89%	98.89%	98.89%	98.89%
	100	98.89%	98.89%	98.89%	98.89%	98.89%	98.89%	98.89%	98.89%	98.89%



En esta comparativa he usado más valores, ya que se puede comprobar que, cuanto más alto es el valor de C, más porcentaje de precisión tiene, aunque tiene una penalización importante y es el tiempo de ejecución, siendo muy alto para C = 100.

Por otro lado, también se puede observar que la S no afecta al resultado.

NOTA: *ESTOS RESULTADOS DEPENDEN DE LAS CARACTERÍSTICAS ALEATORIAS ESCOGIDAS, POR LO QUE PUEDEN VARIAR*

SVM RBF contra SVM Lineal

Considero que el kernel radial es más interesante en este dataset, ya que si usamos C > 1 y S > 1, conseguiremos un 100% de precisión en un tiempo menor que si intentamos C > 100 en un kernel lineal. Estas observaciones son orientativas, ya que el resultado depende en gran medida de los datos aleatorios, aún así, en términos generales, escogería RBF como kernel, con C > 1 y S > 1. Por ello, he dejado 'rbf' como kernel por defecto.

COMPARATIVA ENTRE CLASIFICADORES

	Precisión	Tiempo hasta óptimo	Dificultad de programación
Regresión Logística	> 96%	4 segundos	Normal
Red Neuronal	100%	200 < t < 2800 segundos	Difícil
SVM	100%	100 < t < 130 segundos	Fácil

NOTA 1: El tiempo óptimo es el tiempo que tarda en terminar el programa desde el punto de vista del usuario, es decir, cuando termina todas las combinaciones o cuando consigue un 100% de precisión.

NOTA 2: El tiempo depende del ordenador, en este caso el tiempo se ha medido con un i7 3700K con SSD.

NOTA 3: La dificultad de programación la he medido en base a la cantidad de código escrito y los problemas que puede haber dado.

La regresión logística ofrece los mejores resultados en cuanto a tiempo se refiere, ya que tan solo tarda 4 segundos en ofrecer más de un 96% de acierto. La dificultad radica en programar correctamente las funciones de coste y gradiente.

Las redes neuronales ofrecen un 100% de precisión en más de 3 minutos. La dificultad de programación alta, ya que, a parte de las funciones de coste y gradiente, se tienen que implementar las funciones de propagación y probar distintas arquitecturas para escoger la óptima, en la comparativa podemos observar que 50 nodos en la hidden layer son mejor que 25 y 75 nodos.

Las SVM ofrecen un 100% de precisión con un kernel RBF en un tiempo superior a un minuto y medio. En este caso son más fáciles de implementar ya que delegamos gran parte de la lógica a sklearn.

La mejor clasificación para este problema concreto es el uso de SVM, ya que son fáciles de implementar y ofrecen un 100% de precisión, aunque esto no quiere decir que las otras dos sean peores ni mucho menos, ya que la implementación de la SVM ha sido más sencilla por el uso de una biblioteca externa. Tanto la regresión logística como las redes neuronales tienen bibliotecas o frameworks que facilitan su implementación, el tipo de clasificador que se debe escoger depende de lo que el desarrollador busque y de la dificultad del problema.

Código

common_functions.py

```
#!/usr/bin/env python
# coding: utf-8
# # ESTE ARCHIVO CONTIENE FUNCIONES COMUNES DE LOS
CLASIFICADORES

import numpy as np

def sigmoid(z):
    """
    Función sigmoide
    recibe como entrada un número / array
    devuelve la función sigmoide
    """
    return 1 / (1 + np.exp(-z))
```

dataset_functions.py

```
#!/usr/bin/env python
# coding: utf-8
# # ESTE ARCHIVO CONTIENE FUNCIONES QUE PERMITEN LEER Y
MODIFICAR EL FORMATO DE LAS 'X' E 'Y'


import numpy as np
from pandas.io.parsers import read_csv
from sklearn.preprocessing import LabelEncoder

def read_dataset():
    """
    Lee el dataset 'mushrooms.csv' y devuelve los datos
    divididos en X e Y
    """
    df = read_csv("mushrooms.csv")
    data = df.sample(frac = 1).to_numpy()          # SHUFFLE
DATA

    X = np.array(data[:, 1:])
    Y = np.array(data[:, 0])


    return X, Y
```

```
def onehot(Y):
    """
    A partir de un vector, normalmente de resultados, se aplica
    una codificación onehot y la devuelve en números enteros

    Por ejemplo:
    Y = [1, 2, 3, 1]

    Y_ONEHOT = [
        [1,0,0]
        [0,1,0]
        [0,0,1]
        [1,0,0]
    ]

    """

    le = LabelEncoder()
    labels = le.fit(Y[:]).classes_
    m = len(Y)

    Y = (Y - 1)
    Y_onehot = np.zeros((m, labels.shape[0]))

    for i in range(len(Y)):
        Y_onehot[i, int(Y[i])] = 1

    return Y_onehot.astype(int)



def manage_data(X, Y, use_onehot = False):
    """
    Función que recibe como parámetros de entrada X, Y, y un
booleano
    que transforma la Y a onehot para transformar los valores de
dichos
    parámetros a int
    Devuelve la X y la Y en el formato deseado
    """
    X = string2int(X, X.shape[1])    # OBLIGATORIO EXP CON
FLOATS
    Y = string2int(Y)

    if use_onehot:
        Y = onehot(Y)

    X = np.hstack([np.ones([X.shape[0], 1], dtype=float), X])

    return X, Y
```

```
def string2int(v, dim = 0):
    """
    Recibe como entrada un array de strings y las dimensiones
que tiene,
    devuelve el mismo array con los strings cambiados a int, por
ejemplo:
        ENTRADA:   ['A', 'B', 'C', 'A']
        SALIDA:    [0, 1, 2, 1]
    """
    le = LabelEncoder()

    if dim == 0:
        le.fit(v[:]).classes_       # 0 = EDIBLE
        v = le.transform(v)         # 1 = POISONOUS

    else:
        for i in range(0, dim):
            le.fit(v[:, i]).classes_
            v[:, i] = le.transform(v[:, i])

    return v.astype(float)



def encode_example(example):
    """
    A partir de un ejemplo (Array de chars o strings) se
devuelve
    un array de numeros codificados
    Por ejemplo:
        ENTRADA:   ['A', 'B', 'C', 'A']
        SALIDA:    [0, 1, 4, 0]
    """
    example = np.array([example])
    X, _ = read_dataset()
    le = LabelEncoder()


    for i in range(0, example.shape[1]):
        le.fit(X[:, i]).classes_
        example[:, i] = le.transform(example[:, i])

    example = np.hstack([1, example[0]])     # 'ONES COLUMN

    return example.astype(float).ravel()
```

```
def divide_dataset(X, Y, train_set = 0.8, cv_set = 0.1,
test_set=0.1):
    """
    Divide el dataset con porcentajes que entran como parámetro,
por ejemplo:
        Train_set = 80%
        Cv_set = 10%
        Test_set = 10%
    Devuelve el dataset dividido en Train set y Test set
    """

    train_set_size = int(train_set * X.shape[0])
    cv_set_size = int(cv_set * X.shape[0])

    X_train = X[:train_set_size, :]
    X_cv = X[train_set_size:train_set_size + cv_set_size, :]
    X_test = X[train_set_size + cv_set_size:, :]

    Y_train = Y[:train_set_size]
    Y_cv = Y[train_set_size:train_set_size + cv_set_size]
    Y_test = Y[train_set_size + cv_set_size:]

    return X_train, Y_train, X_cv, Y_cv, X_test, Y_test
```

logistic_regression.py

```
#!/usr/bin/env python
# coding: utf-8
# # IMPORTS

import numpy as np
import scipy.optimize as opt
import matplotlib.pyplot as plt

from dataset_functions import *
from common_functions import *


# # MULTIVARIABLE LOGISTIC REGRESSION

# ## FUNCTIONS

# ### COST AND GRADIENT FUNCTIONS
```

```
def coste(theta, X, Y, lam):
    """
    Función de coste, recibe como entrada las thetas, las
características,
    los resultados y el parámetro lambda de regularización,
    devuelve el coste regularizado
    """
    m = X.shape[0]
    n = X.shape[1]

    h_theta = np.dot(X, theta)
    sig = sigmoid(h_theta)
    positive = np.dot(np.log(sig).T, Y)
    negative = np.dot(np.log(1 - sig).T, 1 - Y)
    J_theta = (-1 / m) * (positive + negative)

    # Regularizacion
    reg = (lam /(2 * m)) * np.sum(np.square(theta))

    # Coste Regularizado
    J_theta += reg

    return J_theta



def gradiente(theta, X, Y, lam):
    """
    Función de gradiente, recibe como entrada las thetas, las
características,
    los resultados y el parámetro lambda de regularización,
    devuelve la gradiente regularizada
    """
    m = X.shape[0]
    n = X.shape[1]

    h_theta = np.dot(X, theta.T)
    sig = sigmoid(h_theta)
    gradient = (1/m) * np.dot(sig.T - Y, X)

    # Regularizacion
    reg = (lam / m) * theta

    # Gradiente Regularizada
    gradient += reg

    return gradient
```

```
# ### PLOT LAMBDAS / COST FUNCTION

def print_cost_lambdas(lambdas_list, costs_list):
    """
    Función que imprime las lambdas en el eje X,
    y el coste en el eje Y
    """
    plt.figure()
    plt.plot(lambdas_list, costs_list, c = 'r')
    plt.xlabel('lambda')
    plt.ylabel('cost')
    plt.show()


# ### PREDICTION FUNCTIONS

def predict(theta, X, Y):
    """
    Función que, a partir de las thetas, las características,
    los resultados, calcula el porcentaje de acierto que tiene
nuestra IA
    Imprime el porcentaje con dos decimales
    """
    predictions = np.dot(X, theta) > 0

    hits = np.sum(predictions == Y)
    percentage = hits / X.shape[0] * 100

    print("The logistic regression is reliable in {:.2f}% of the
time\n".format(percentage))

    return percentage


# ### CHOOSE OPTIMAL VALUES FUNCTION

def get_opt_thetas(theta, X, Y, reg = True):
    """
    Función que recibe como entrada las thetas, las
características,
    los resultados y si se quiere aplicar un parámetro de
regularización.
    Devuelve la theta optima con diferentes regularizaciones (en
el caso de
    que reg = True)
    Además, llama a la función 'print_cost_lambdas' para
comparar los
    distintos costes con las diferentes regularizaciones
    """
    # Inicializamos los valores
    lambdas = [0.001, 0.003, 0.01, 0.03, 0.1, 0.3]
```

```
    n = X.shape[1]

    theta_opt = np.zeros(n)
    lambd_opt = lambdas[0]
    cost_list = []


    # Probamos los distintos parámetros de regularización
    if reg:
        for lambd in lambdas:

            theta, _, _ = opt.fmin_tnc(
                func=coste,
                x0 = theta,
                fprime=gradiente,
                args=(X, Y, lambd)
            )

            actual_cost = coste(theta, X, Y, lambd)
            opt_cost = coste(theta_opt, X, Y, lambd_opt)


            # Coste actual vs Coste óptimo
            if (opt_cost > actual_cost):
                theta_opt = theta
                lambd_opt = lambd

            cost_list.append(actual_cost)

        # Dibujamos la evolución del coste respecto a la
regularización
        print_cost_lambdas(lambdas, cost_list)


    else:
        theta_opt, _, _ = opt.fmin_tnc(
            func=coste,
            x0 = theta,
            fprime=gradiente,
            args=(X, Y, 0)
        )


    return theta_opt
```

```
# ## EXTERNAL FUNCTIONS


def save_lr_model(thetas):
    """
    Guarda un array de thetas en la ruta models/theta_lr.npy
    """
    np.save("models/theta_lr.npy", thetas)



def load_lr_model():
    """
    Carga un array de thetas en la ruta models/theta_lr.npy
    """
    thetas = np.load("models/theta_lr.npy")
    return thetas



def show_lr_prediction():
    """
    Carga los datos y las thetas óptimas, divide los datos y
    prueba las thetas óptimas sobre esos datos.
    Finalmente muestra el porcentaje de acierto de esos datos
    """
    X, Y = read_dataset()
    X, Y = manage_data(X, Y, use_onehot = False)
    _, _, _, _, X_test, Y_test = divide_dataset(X, Y)
    theta = load_lr_model()

    predict(theta, X_test, Y_test)



def predict_example_lr(example):
    """
    Carga las theta óptimas y, a partir de un ejemplo, predice
su resultado
    devuelve la predicción como booleano
    """
    theta = load_lr_model()
    prediction = np.dot(example, theta) > 0

    return prediction
```

```
def main_lr():
    """
    Función que entrena el clasificador, obtiene las theta
óptimas y
    guarda el modelo óptimo.
    """
    X, Y = read_dataset()
    X, Y = manage_data(X, Y, use_onehot = False)

    m = X.shape[0]
    n = X.shape[1]

    X_train, Y_train, _, _, X_test, Y_test = divide_dataset(X,
Y)

    theta = np.zeros(n)

    # CLASIFICACIÓN SIN REGULARIZACIÓN
    print('LOGISTIC REGRESSION WITHOUT REGULARIZATION')
    theta_opt = get_opt_thetas(theta, X_train, Y_train,
reg=False)
    predict_no_reg = predict(theta_opt, X_test, Y_test)

    # CLASIFICACIÓN CON REGULARIZACIÓN
    print('LOGISTIC REGRESSION WITH REGULARIZATION')
    theta_opt_reg = get_opt_thetas(theta, X_train, Y_train,
reg=True)
    predict_reg = predict(theta_opt_reg, X_test, Y_test)

    if predict_no_reg > predict_reg:
        save_lr_model(theta_opt)
    else:
        save_lr_model(theta_opt_reg)



# main_lr()
```

neural_network.py

```
#!/usr/bin/env python
# coding: utf-8

# # IMPORTS

import numpy as np
import scipy.optimize as opt
import matplotlib.pyplot as plt

from dataset_functions import *
from common_functions import *


# # NEURAL NETWORKS

# ## FUNCTIONS

# ### ARQUITECTURE OF NEURAL NETWORK


def model(input_size, hidden_layer, num_labels):
    """
    A partir de tres números enteros (entrada, capas ocultas y
etiquetas)
    genera una arquitectura de red neuronal con los valores de
theta
    aleatorios.
    Devuelve un único array con todas las theta
    """
    eIni = 0.12

    Theta1_sh = (hidden_layer, input_size + 1)
    Theta2_sh = (num_labels, hidden_layer + 1)

    thetas_random = dict()

    thetas_random["Theta1"] = random_thetas(Theta1_sh, eIni)
    thetas_random["Theta2"] = random_thetas(Theta2_sh, eIni)

    th_random = np.concatenate(
        (
            np.ravel(thetas_random["Theta1"]),
            np.ravel(thetas_random["Theta2"])
        )
    )

    return th_random
```

```
# ### COST AND GRADIENT FUNCTIONS
def coste(X, Y, w, lam):
    """
    Función de coste, recibe como entrada las características,
    los resultados, los pesos (las thetas) y el parámetro lambda
    de regularización.
    Devuelve el coste regularizado
    """
    m = X.shape[0]
    J_theta = 0
    aux = 0
    _, _, _, _, h_theta = forward_prop(X, w)

    for i in range(m):
        aux +=np.sum(-Y[i] * np.log(h_theta[i])
                     - (1 - Y[i])* np.log(1 - h_theta[i]))


    J_theta = (1 / m) * aux

    # Regularizacion
    reg = 0

    for theta in w.keys():
        if "__" not in theta:
            reg += np.sum(np.square(w[theta][:, 1:]))

    reg *= (lam /(2 * m))

    # Coste Regularizado
    J_theta += reg

    return J_theta


def gradiente(X, Y, w, lam):
    """
    Función de gradiente, recibe como entrada las
características,
    los resultados, los pesos (las thetas) y el parámetro lambda
    de regularización.
    Devuelve la gradiente regularizada
    """
    m = X.shape[0]

    d = dict()
    d["delta1"] = np.zeros(w["Theta1"].shape)

    d["delta2"] = np.zeros(w["Theta2"].shape)
    a1, z2, a2, z3, h_theta = forward_prop(X, w)
```

```
    for i in range(m):
        # Calcular d2 y d3
        d3 = h_theta[i] - Y[i]                  # (10, )
        g_z2 = a2[i] * (1 - a2[i])              # (26, )
        d2 = np.dot(d3, w["Theta2"]) * g_z2     # (26, )
        #d2 = d2[1:]                             # (25, )

        # Actualizar deltas
        d["delta1"] += np.dot(d2[1:, np.newaxis],
a1[i][np.newaxis, :])
        d["delta2"] += np.dot(d3[:, np.newaxis],
a2[i][np.newaxis, :])

    d["delta1"] /= m
    d["delta2"] /= m

    #Regularizar deltas
    reg1 = ((lam / m) * w["Theta1"][:, 1:]) # No theta primera
columna
    reg2 = ((lam / m) * w["Theta2"][:, 1:]) # No theta primera
columna
    d["delta1"][:, 1:] += reg1              # j = 0 no tiene
regularización
    d["delta2"][:, 1:] += reg2              # j = 0 no tiene
regularización


    return np.concatenate(
        (np.ravel(d["delta1"]),
         np.ravel(d["delta2"]))
    )


# ### NEURAL NETWORK FUNCTIONS
def forward_prop(X, w):
    """
    Función de propagación hacia adelante, aplica la lógica de
las
    Redes Neuronales para únicamente dos theta.
    """
    a1 = X
    a1 = np.hstack([np.ones([X.shape[0], 1]), a1])

    z2 = np.dot(w['Theta1'], a1.T)
    a2 = sigmoid(z2).T
    a2 = np.hstack([np.ones([a2.shape[0], 1]), a2])

    z3 = np.dot(w['Theta2'], a2.T)
    h = sigmoid(z3).T

    return a1, z2, a2, z3, h
```

```
def backprop(params_rn, num_entradas, num_ocultas,
num_etiquetas, X, Y, reg):
    """
    Esta función devuelve una tupla (coste, gradiente) con el
coste y
    el gradiente de una red neuronal de tres capas, con
num_entradas,
    num_ocultas nodos en la capa oculta y num_etiquetas nodos en
la
    capa de salida. Si m es el número de ejemplos de
entrenamiento,
    la dimensiónde de 'X' es (m, num_entradas) y la de 'y'
    es (m, num_etiquetas)
    """
    w = dict()
    w["Theta1"], w["Theta2"] = relocate(params_rn, num_entradas,
num_ocultas, num_etiquetas)

    return coste(X, Y, w, reg), gradiente(X, Y, w, reg)


# ### SUPPORT FUNCTIONS

def relocate(params_rn, num_entradas, num_ocultas,
num_etiquetas):
    """
    A partir de un único vector y el número de entradas, de
nodos ocultos
    y etiquetas se devuelven los vectores Theta1 y Theta2, con
los tamaños
    establecidos
    """
    Theta1 = np.reshape(params_rn[:num_ocultas * (num_entradas +
1)],
                        (num_ocultas, (num_entradas + 1)))
    Theta2 = np.reshape(params_rn[num_ocultas * (num_entradas +
1):],
                        (num_etiquetas, (num_ocultas + 1)))
    return Theta1, Theta2


def random_thetas(shape, E):
    """
    Recibe como parámetros las dimensiones y el Epsilon (rango -
> [-E, E])

    Primero creamos una matriz de positivos y negativos
    Posteriormente creamos una matriz con números aleatorios
positivos < Epsilon
    Multiplicamos las dos matrices, tenemos aleatorios positivos
y negativos.
```

```
    Devuelve las dimensiones con valores aleatorios

    """
    posNeg = np.random.random((shape))
    pos = np.where(posNeg < .5)
    neg = np.where(posNeg >= .5)
    posNeg[pos] = 1
    posNeg[neg] = -1

    return (np.random.random((shape)) %E ) * posNeg


# ### TRAIN NEURAL NETWORK FUNCTION

def train(backprop, thetas, X, Y, input_size, hidden_layer,
              num_labels, reg = 1, iterations = 70):
    """
    Función que, a partir de la función de backprop, las
características,
    los resultados, los tamaños de las capas, la regularización
y las
    iteraciones, entrena la red neuronal para obtener los
parámetros theta
    óptimos.
    Devuelve las theta óptimas en forma de diccionario
    """

    # Thetas optimas
    res = opt.minimize(
        fun = backprop,
        x0 = thetas,
        args=(input_size, hidden_layer, num_labels, X, Y, reg),
        options={'maxiter': iterations},
        method='TNC',
        jac=True
    )

    # Recolocamos
    w = dict()
    w["Theta1"], w["Theta2"] = relocate(res.x,
                                        X.shape[1],
                                        hidden_layer,
                                        num_labels)

    return w
```

```
# ### PREDICTION FUNCTIONS
def predict(X, w):
    """
    Función que a partir de las características y los pesos
(thetas)
    predice los resultados de dichas características.
    Devuelve un array con esas predicciones
    """
    Y_hat = []
    _, _, _, _, pred = forward_prop(X, w)

    for i in range(pred.shape[0]):
        ejemplo = pred[i]
        num = np.argmax(ejemplo)
        Y_hat.append(num)

    Y_hat = np.array(Y_hat)

    return Y_hat


def acc(X, Y, w):
    """
    Función que a partir de las características, los resultados
    y los pesos (thetas), calcula el porcentaje de acierto del
clasificador.
    Devuelve un float con el porcentaje de acierto.
    """
    m = Y.shape[0]

    Y_hat = predict(X, w)

    percentage = np.round(
        np.sum(Y_hat == Y.argmax(1)) / m * 100,
        decimals = 2
    )

    return percentage


# ### PLOT ITERATIONS / COST FIGURE
def print_opt_lambdas(iterations, hit_history, lambdas):
    """
    Función que imprime las iteraciones en el eje X,
    y la precisión en el eje Y
    """
    plt.figure()

    for hit in hit_history:
        plt.plot(iterations, hit)
```

```
    plt.xlabel('iterations')
    plt.ylabel('hit_accuracy')
    plt.legend(lambdas)

    plt.show()


# ### CHOOSE OPTIMAL VALUES FUNCTION
def get_opt_thetas(th_random, X_train, Y_train, X_test, Y_test,
                   input_size, hidden_layer, num_labels):
    """
    Función que recibe unas theta aleatorias, las
características y
    los resultados, tanto de train set como de test set, y el
número de
    capas.
    Prueba, a partir de unas iteraciones y unos parámetros de
    regularización preestablecidos, todas las combinaciones,
    de esta forma obtiene las theta óptimas (basándonos en la
tasa de
    acierto que tiene)
    Devuelve la precisión, las thetas optimas y su combinación
óptima
    de parámetros
    """
    iterations = [ 10, 50, 100, 200, 300 ]
    lambdas = [ 0.01, 0.03, 0.1, 0.3, 1, 3, 10 ]
#     iterations = [ 1,2,3 ]
#     lambdas = [ 0.01, 0.03, 0.1, 0.3 ]

    precision_opt = -1
    thetas_opt = np.array(0)
    it_opt = iterations[0]
    lambd_opt = lambdas[0]

    hit_lambda_history = []
    hit_total_history = []

    for lambd in lambdas:
        hit_lambda_history = []

        for i in iterations:
            thetas = train(backprop, th_random, X_train,
Y_train,
                           input_size, hidden_layer,
num_labels,
                           lambd, i)

            p = acc(X_test, Y_test, thetas)
            hit_lambda_history.append(p)
```

```
            print("Lambda: {}".format(lambd),
                  "\tIteraciones: {}".format(i),
                  "\tPrecisión:{}%\n".format(p)
                 )

            if(p > precision_opt):
                precision_opt = p
                thetas_opt = thetas
                it_opt = i
                lambd_opt = lambd

            if(precision_opt == 100.0):
                break


        hit_total_history.append(np.array(hit_lambda_history))

        if(precision_opt == 100.0):
            break

    if(precision_opt < 100.0):
        print_opt_lambdas(iterations, hit_total_history,
lambdas)

    return precision_opt, thetas_opt, it_opt, lambd_opt


# ### STUDY BIAS AND VARIANCE
def learning_errors(th_random,
                    num_entradas, num_ocultas, num_etiquetas,
                    X_train, Y_train, X_cv, Y_cv, lambd = 0,
                    iterations = 100, max_examples = 100
                   ):
    """
    A partir de unas theta aleatorias, las capas, las
características
    y los resultados, tanto del train set como de la CV, la
regularización,
    las iteraciones, y el número máximo de ejemplos a probar,
calcula la
    tasa de error del trainset y de la CV set.
    Devuelve los vectores de error del entrenamiento y la CV
    """

    trainError = []
    cvError = []
    m = X.shape[0]

    for i in range(1, max_examples):
#         print(i, " de ", max_examples)
        X_i = X_train[0:i]
```

```
        Y_i = Y_train[0:i]

        res = opt.minimize(
            fun = backprop,
            x0 = th_random,
            args=(num_entradas, num_ocultas, num_etiquetas,
                  X_i, Y_i, lambd),
            method='TNC',
            jac=True,
            options={'maxiter': iterations},
        )

        trainError.append(backprop(res.x,
                                   num_entradas, num_ocultas,
num_etiquetas,
                                   X_i, Y_i, lambd)
                          [0])
        cvError.append(backprop(res.x,
                                num_entradas, num_ocultas,
num_etiquetas,
                                X_cv, Y_cv, lambd)
                       [0])

    return np.array(trainError), np.array(cvError)



def print_learning_errors(train_error, cv_error):
    """
    Dibuja el error del entrenamiento y de la CV
    """

    plt.figure()

    r = range(0, len(train_error))

    plt.plot(r, train_error)
    plt.plot(r, cv_error)
    plt.legend(['Train', 'Cross Validation'])
    plt.xlabel('Iterations')
    plt.ylabel('Cost')

    plt.show()


# ## EXTERNAL FUNCTIONS
def save_nn_model(thetas):
    """
    Guarda dos array de thetas en la ruta models,
    con los archivos theta1_nn.npy  y  theta2_nn.npy
    """
```

```
    np.save("models/theta1_nn.npy", thetas["Theta1"])
    np.save("models/theta2_nn.npy", thetas["Theta2"])



def load_nn_model():
    """
    Carga dos array de thetas en la ruta models,
    con los archivos theta1_nn.npy  y  theta2_nn.npy
    y los devuelve en un único diccionario
    """
    thetas = dict()

    thetas["Theta1"] = np.load("models/theta1_nn.npy")
    thetas["Theta2"] = np.load("models/theta2_nn.npy")

    return thetas



def show_nn_prediction():
    """
    Carga los datos y las thetas óptimas, divide los datos y
    prueba las thetas óptimas sobre esos datos.
    Finalmente muestra el porcentaje de acierto de esos datos
    """
    X, Y = read_dataset()
    X, Y = manage_data(X, Y, use_onehot = True)
    _, _, _, _, X_test, Y_test = divide_dataset(X, Y)
    w = load_nn_model()

    print("The neural network is reliable in {:.2f}% of the
time\n"
      .format(acc(X_test, Y_test, w)))



def predict_example_nn(example):
    """
    Carga las theta óptimas y, a partir de un ejemplo, predice
su resultado
    devuelve la predicción como booleano
    """
    w = load_nn_model()
    _, _, _, _, prediction = forward_prop(np.array([example]),
w)

    return bool(prediction.argmin())
```

```
def main_nn():
    """
    Función que entrena el clasificador, obtiene las theta
óptimas y
    guarda el modelo óptimo.
    """
    X, Y = read_dataset()
    X, Y = manage_data(X, Y, use_onehot = True)

    m = X.shape[0]
    n = X.shape[1]

    X_train, Y_train, X_cv, Y_cv, X_test, Y_test =
divide_dataset(X, Y)

    input_size = X_train.shape[1]
    num_labels = 2

    hidden_layer_nodes = [ 10, 25, 50, 75, 100, 150, 200 ]

    acc_opt = 0
    thetas_opt = np.array(0)
    th_random_opt = np.array(0)
    it_opt = np.inf
    lambd_opt = np.inf
    hidden_layer_opt = hidden_layer_nodes[0]

    for nodes in hidden_layer_nodes:

        th_random = model(input_size, nodes, num_labels)

        acc_act, thetas_act, it_act, lambd_act = get_opt_thetas(
            th_random,
            X_train,
            Y_train,
            X_test,
            Y_test,
            input_size,
            nodes,
            num_labels)

        if(acc_act > acc_opt):
            acc_opt = acc_act
            thetas_opt = thetas_act
            it_opt = it_act
            lambd_opt = lambd_act
            hidden_layer_opt = nodes

        if(acc_act == 100.0):
            break
```

```
        else:
            train_error, cv_error =
learning_errors(np.concatenate((np.ravel(thetas_opt["Theta1"]),
np.ravel(thetas_opt["Theta2"]))),
                    input_size, hidden_layer_opt, num_labels,
                    X_train, Y_train, X_cv, Y_cv,
                    lambd = lambd_opt, iterations = it_opt)

            print_learning_errors(train_error, cv_error)

    save_nn_model(thetas_opt)



# main_nn()
```

svm.py

```
#!/usr/bin/env python
# coding: utf-8

# # IMPORTS

import numpy as np
import matplotlib.pyplot as plt

from dataset_functions import *
from common_functions import *

from sklearn.svm import SVC
from joblib import dump, load
from sklearn.metrics import accuracy_score


# # SUPPORT VECTOR MACHINE

# ## FUNCTIONS

# ### CHOOSE OPTIMAL VALUES FUNCTION


def print_parameters(values, hit_total_history):
    """
    A partir de unos valores y un array de aciertos, dibuja una
gráfica que
    los relaciona.
    """
    for hit in hit_total_history:
        plt.plot(values, hit)

    plt.xlabel('S')
    plt.ylabel('hit_accuracy')
    plt.legend(values)

    plt.show()


def parameter_election(X, Y, X_val, Y_val, kernel):
    """
    A partir de unas características y unos resultados, tanto de
    entrenamiento como de CV, y un tipo de kernel, obtiene los
parámetros
    óptimos del modelo. Para conocer los parámetros óptimos nos
basamos
    en el acierto que tiene el modelo con dichos parámetros.
    Devuelve la precisión, y los parámetros óptimos.
    """
```

```
values = [ .01, .03, .1, .3, 1, 3, 10 ]

c_opt = None
s_opt = None
acc_opt = 0
hit_history = []
hit_total_history = []

for C in values:
    hit_history = []
    for S in values:
        gamma = 1 / (2 * S**2)

        svm = SVC(kernel=kernel, C=C, gamma=gamma)
        svm.fit(X, Y)
        acc = np.round(
            accuracy_score(Y_val, svm.predict(X_val)) * 100,
            decimals = 2
        )

        hit_history.append(acc)
        print('C: {}\tS: {}\t-> {}%'.format(C, S, acc))

        if(acc > acc_opt):
            acc_opt = acc
            c_opt = C
            s_opt = S


        if(acc_opt == 100.0):
            save_svm_model(svm)
            break

    hit_total_history.append(hit_history)

    if(acc_opt == 100.0):
        break

if(acc_opt < 100.0):
    print_parameters(values, hit_total_history)

return acc_opt, c_opt, s_opt
```

```
# ## EXTERNAL FUNCTIONS
def save_svm_model(model):
    """
    Guarda el modelo SVM en la ruta models/model_svm.joblib
    """
    dump(model, 'models/model_svm.joblib')



def load_svm_model():
    """
    Carga el modelo SVM en la ruta models/model_svm.joblib
    """
    return load('models/model_svm.joblib')



def show_svm_prediction():
    """
    Carga los datos y el modelo óptimo, divide los datos y
    prueba el modelo sobre esos datos.
    Finalmente muestra el porcentaje de acierto de esos datos
    """
    X, Y = read_dataset()
    X, Y = manage_data(X, Y, use_onehot = False)

    _, _, _, _, X_test, Y_test = divide_dataset(X, Y)
    svm = load_svm_model()

    percentage = np.round(
        accuracy_score(Y_test, svm.predict(X_test)) * 100,
        decimals = 2
    )

    print("The SVM is reliable in {:.2f}% of the time\n"
      .format(percentage))



def predict_example_svm(example):
    """
    Carga el modelo óptimo y, a partir de un ejemplo, predice su
resultado
    devuelve la predicción como booleano
    """
    svm = load_svm_model()
    return bool(svm.predict(np.array([example])))
```

```
def main_svm():
    """
    Función que entrena el clasificador, obtiene el modelo
óptimo y
    lo guarda.
    """
    X, Y = read_dataset()
    X, Y = manage_data(X, Y, use_onehot = False)
    m = X.shape[0]
    n = X.shape[1]

    X_train, Y_train, X_cv, Y_cv, X_test, Y_test =
divide_dataset(X, Y)

    acc_opt, c_opt, s_opt = parameter_election(X_train, Y_train,
                                               X_cv, Y_cv,
'rbf')



# main_svm()
```

source.py

```
#!/usr/bin/env python
# coding: utf-8


import time

from dataset_functions import *
from common_functions import *
from logistic_regression import *
from neural_network import *
from svm import *


def manage_example_options(text, correct_answers):
    """
    A partir de un texto y un array de respuestas, valida la
entrada
    recibida.
    Devuelve la opción seleccionada sabiendo que está entre las
respuestas
    esperadas.
    """
    opt = None
    while(True):
        print(text)

        opt = input().lower()

        if opt in correct_answers:
            break
        else:
            print('----- WRONG TYPE -----')

    return opt




def example_menu():
    """
    Función que muestra un menú por pantalla, y espera a una
entrada.
    Cuando la respuesta esperada es correcta se añade a un
array, que
    acabará siendo el ejemplar de hongo/seta.
    Devuelve dicho array
    """
    example = []
```

```
text = """
Cap shape?
    b: bell
    c: conical
    x: convex
    f: flat
    k: knobbed
    s: sunken
"""
options = ['b', 'c', 'x', 'f', 'k', 's']
example.append(manage_example_options(text, options))


text = """
Cap surface?
    f: fibrous
    g: grooves
    y: scaly
    s: smooth
"""
options = ['f', 'g', 'y', 's']
example.append(manage_example_options(text, options))


text = """
Cap color?
    n: brown
    b: buff
    c: cinnamon
    g: gray
    r: green
    p: pink
    u: purple
    e: red
    w: white
    y: yellow
"""
options = ['n', 'b', 'c', 'g', 'r', 'p', 'u', 'e', 'w', 'y']
example.append(manage_example_options(text, options))


text = """
Bruises?
    t: true
    f: false
"""
options = ['t', 'f']
example.append(manage_example_options(text, options))
```

```
text = """
Odor?
    a: almond
    l: anise
    c: creosote
    y: fishy
    f: foul
    m: musty
    n: none
    p: pungent
    s: spicy
"""
options = ['a', 'l', 'c', 'y', 'f', 'm', 'n', 'p', 's']
example.append(manage_example_options(text, options))


text = """
Gill attachment?
    a: attached
    f: free
"""
options = ['a', 'f']
example.append(manage_example_options(text, options))


text = """
Gill spacing?
    c: close
    w: crowded
"""
options = ['c', 'w']
example.append(manage_example_options(text, options))


text = """
Gill size?
    b: broad
    n: narrow
"""
options = ['b', 'n']
example.append(manage_example_options(text, options))


text = """
Gill color?
    k: black
    n: brown
    b: buff
    h: chocolate
    g: gray
    r: green
```

```
        o: orange
        p: pink
        u: purple
        e: red
        w: white
        y: yellow
    """
    options = ['k', 'n', 'b', 'h', 'g', 'r', 'o', 'p', 'u', 'e',
'w', 'y']
    example.append(manage_example_options(text, options))


    text = """
    Stalk shape?
        e: enlarging
        t: tampering
    """
    options = ['e', 't']
    example.append(manage_example_options(text, options))


    text = """
    Stalk root?
        b: bulbous
        c: club
        e: equal
        r: rooted
        ?: missing
    """
    options = ['b', 'c', 'e', 'r', '?']
    example.append(manage_example_options(text, options))


    text = """
    Stalk surface above ring?
        f: fibrous
        y: scaly
        k: silky
        s: smooth

    """
    options = ['f', 'y', 'k', 's']
    example.append(manage_example_options(text, options))


    text = """
    Stalk surface below ring?
        f: fibrous
        y: scaly
        k: silky
        s: smooth
```

```
"""
options = ['f', 'y', 'k', 's']
example.append(manage_example_options(text, options))


text = """
Stalk color above ring?
    n: brown
    b: buff
    c: cinnamon
    g: gray
    o: orange
    p: pink
    e: red
    w: white
    y: yellow
"""
options = ['n', 'b', 'c', 'g', 'o', 'p', 'e', 'w', 'y']
example.append(manage_example_options(text, options))


text = """
Stalk color below ring?
    n: brown
    b: buff
    c: cinnamon
    g: gray
    o: orange
    p: pink
    e: red
    w: white
    y: yellow
"""
options = ['n', 'b', 'c', 'g', 'o', 'p', 'e', 'w', 'y']
example.append(manage_example_options(text, options))


text = """
Veil type?
    p: partial
"""
options = ['p']
example.append(manage_example_options(text, options))


text = """
Veil color?
    n: brown
    o: orange
    w: white
```

```
    y: yellow
"""
options = ['n', 'o', 'w', 'y']
example.append(manage_example_options(text, options))


text = """
Ring number?
    n: none
    o: one
    t: two
"""
options = ['n', 'o', 't']
example.append(manage_example_options(text, options))


text = """
Ring type?
    e: evanescent
    f: flaring
    l: large
    n: none
    p: pendant
"""
options = ['e', 'f', 'l', 'n', 'p']
example.append(manage_example_options(text, options))


text = """
Spore print color?
    k: black
    n: brown
    b: buff
    h: chocolate
    r: green
    o: orange
    u: purple
    w: white
    y: yellow
"""
options = ['k', 'n', 'b', 'h', 'r', 'o', 'u', 'w', 'y']
example.append(manage_example_options(text, options))


text = """
Population?
    a: abundant
    c: clustered
    n: numerous
    s: scattered
    v: several
```

```
        y: solitary
    """
    options = ['a', 'c', 'n', 's', 'v', 'y']
    example.append(manage_example_options(text, options))


    text = """
    Habitat?
        g: grasses
        l: leaves
        m: meadows
        p: paths
        u: urban
        w: waste
        d: woods

    """
    options = ['g', 'l', 'm', 'p', 'u', 'w', 'd']
    example.append(manage_example_options(text, options))

    return example



def print_mushroom_state(state):
    """
    A partir de un booleano, muestra por pantalla si el ejemplar
de hongo
    puede ser ingerido o no
    """
    if(state):
        print("WARNING! THE MUSHROOM IS POISONOUS")
    else:
        print("You can eat the mushroom safely\n\n")



def manage_lr():
    """
    Función que gestiona el clasificador de regresión logística.
    Pregunta si se puede reentrenar o si se quiere introducir
manualmente
    un ejemplo.
    En el caso de querer reentrenar el clasificador, se
cronometrará y
    se llamará a la función main_lr().
    En el caso de querer probar un ejemplar, se llamará a la
función
    example_menu() y posteriormente predecirá si puede ser
comestible o no.
    """
```

```
    opt = input("Do you want to retrain the LR? (y/n)")

    if(opt.lower() == 'y'):
        print('----- RETRAINING LR -----')
        tic = time.process_time()
        main_lr()
        toc = time.process_time()
        print("TIME: {} seconds".format(toc - tic))

    opt = input("Do you want to try an example? (y/n)")

    if(opt.lower() == 'y'):
        e = example_menu()
        e = encode_example(e)
        state = predict_example_lr(e)
        print("REMEMBER!")
        show_lr_prediction()
        print_mushroom_state(state)


def manage_nn():
    """
    Función que gestiona el clasificador de la red neuronal.
    Pregunta si se puede reentrenar o si se quiere introducir
manualmente
    un ejemplo.
    En el caso de querer reentrenar el clasificador, se
cronometrará y
    se llamará a la función main_nn().
    En el caso de querer probar un ejemplar, se llamará a la
función
    example_menu() y posteriormente predecirá si puede ser
comestible o no.
    """
    opt = input("Do you want to retrain the NN? (y/n)")
    if(opt.lower() == 'y'):
        print('----- RETRAINING NN -----')
        tic = time.process_time()
        main_nn()
        toc = time.process_time()
        print("TIME: {} seconds".format(toc - tic))

    opt = input("Do you want to try an example? (y/n)")

    if(opt.lower() == 'y'):
        e = example_menu()
        e = encode_example(e)
        state = predict_example_nn(e)
        print("REMEMBER!")
        show_nn_prediction()
        print_mushroom_state(state)
```

```
def manage_svm():
    """
    Función que gestiona el clasificador de la SVM.
    Pregunta si se puede reentrenar o si se quiere introducir
manualmente
    un ejemplo.
    En el caso de querer reentrenar el clasificador, se
cronometrará y
    se llamará a la función main_svm().
    En el caso de querer probar un ejemplar, se llamará a la
función
    example_menu() y posteriormente predecirá si puede ser
comestible o no.
    """
    opt = input("Do you want to retrain the SVM? (y/n)")
    if(opt.lower() == 'y'):
        print('----- RETRAINING SVM -----')
        tic = time.process_time()
        main_svm()
        toc = time.process_time()
        print("TIME: {} seconds".format(toc - tic))

    opt = input("Do you want to try an example? (y/n)")

    if(opt.lower() == 'y'):
        e = example_menu()
        e = encode_example(e)
        state = predict_example_svm(e)
        print("REMEMBER!")
        show_svm_prediction()
        print_mushroom_state(state)



def menu():
    """
    Menu principal del programa, permite escoger un tipo de
clasificador
    o salir del programa
    """

    print("Welcome to the final subject project - MUSHROOM
CLASSIFIER")

    while (True):
        print(
    """
    You can choose:

        1.\t\t\tLogistic Regression Classifier
        2.\t\t\tNeural Network Classifier
```

```
        3.\t\t\tSupport Vector Machine Classifier
        0 or other key.\t\tExit
    """
        )


        opt = input("Write the number or the first letter: ")

        switcher = {
            'L': 1,
            'N': 2,
            'S': 3,
            'E': 0,
            '1': 1,
            '2': 2,
            '3': 3,
            '0': 0
        }

        opt = switcher.get(opt.upper(), 0)

        if opt == 0:
            break
        elif opt == 1:
            manage_lr()
        elif opt == 2:
            manage_nn()
        else:
            manage_svm()



menu()
```