Cuando trabajamos con modelos que representan una distribución de probabilidad, nuestro objetivo es hacer que la distribución de los datos se acerque lo más posible a las probabilidades que nos da el modelo sobre esos datos. Existen muchas maneras de calcular esa diferencia, una común es usar funciones de divergencia, entre ellas la divergencia de Kullback-Leibler es la más usada. Dadas dos distribuciones de probabilidad P y Q se define asumiendo que sean distribuciones discretas como:

Función de divergencia Kullback – Leibler

$$KL(P|Q) = \sum_i P(i) \cdot \log \frac{P(i)}{Q(i)}$$

↳ Medida no simetrica de la similitud o diferencia entre dos funciones de distribución de probabilidad P y Q.

$x = \{0, 1, 2, 2, 3, 3, 3, 4, 4, 5, 6\}$

$len(x) = 11$

$P(x) = \{1, 1, 2, 3, 2, 1, 1\}$

$Q(x) = \{\frac{1}{11}, \frac{1}{11}, \frac{2}{11}, \frac{3}{11}, \frac{2}{11}, \frac{1}{11}, \frac{1}{11}\}$

$$KL(P|Q) = \log(1 \cdot 1/11) \cdot 4 + \log(2 \cdot 2/11) \cdot 2 + \log(3 \cdot 3/11)$$

En el caso de distribuciones continuas, simplemente substituimos el sumatorio por una integral.

a) Siendo X una muestra de datos $x_1 \dots x_n$ de valores discretos, donde podemos estimar su distribución P a partir de su frecuencia y Q es una distribución de probabilidad sobre el mismo rango de valores discretos. Demuestra que optimizar $KL(P|Q)$ es equivalente a optimizar la log verosimilitud negativa de Q sobre los datos.

> 💡 Existe una relación entre $P(i)$, la frequencia, y $Q(i)$, la distribución de probabilidad.
> $\forall i \ : \ i \in P \rightarrow (i \in Q \ \wedge \ Q(i) = \frac{P(i)}{N})$

Empezaremos optimizando la función de la Log Verosimilitud Negativa i la función de Kullback-Leiber.

$$\begin{aligned}
L(\theta) &= -\log(\prod_{i=1}^{N} Q(i)) \\
&= -\log(\prod_{i=1}^{N} \frac{P(i)}{N}) \\
&= -\sum_{i=1}^{N} \log(\frac{P(i)}{N}) \\
&= -\sum_{i=1}^{N} (\log(P(i)) - \log(N)) \\
&= -\sum_{i=1}^{N} (\log(P(i))) - \sum_{i=1}^{N} (\log(N)) \\
&= -\sum_{i=1}^{N} (\log(P(i))) - (N \cdot \log(N)) \\
&= \sum_{i=1}^{N} (-\log(P(i))) - (N \cdot \log(N)) \\
&= \sum_{i=1}^{N} (\log(\frac{1}{P(i)})) - (N \cdot \log(N))
\end{aligned}$$

$$\begin{aligned}
\frac{\partial}{\partial P} &= \sum_{i=1}^{N} (\log(\frac{1}{\frac{1}{P(i)}})) \\
&= \sum_{i=1}^{N} (P(i))
\end{aligned}$$

$$\begin{aligned}
\frac{\partial}{\partial P} &= 0 \\
0 &= \sum_{i=1}^{N} (P(i))
\end{aligned}$$

$$\begin{aligned}
KL(P|Q) &= \sum_{i=1}^{N} (P(i) \cdot \log(\frac{P(i)}{Q(i)})) \\
&= \sum_{i=1}^{N} (P(i) \cdot (\log(P(i)) - \log(Q(i)))) \\
&= \sum_{i=1}^{N} (P(i) \cdot (\log(P(i)) - \log(\frac{P(i)}{N}))) \\
&= \sum_{i=1}^{N} (P(i) \cdot (\log(P(i)) - (\log(P(i)) - \log(N)))) \\
&= \sum_{i=1}^{N} (P(i) \cdot (\log(P(i)) - \log(P(i)) + \log(N))) \\
&= \sum_{i=1}^{N} (P(i) \cdot (1 + log(N))) \\
&= N \cdot (1 + log(N)) \cdot \sum_{i=1}^{N} P(i) \\
&= (N + N \cdot \log(N)) \cdot \sum_{i=1}^{N} P(i)
\end{aligned}$$

$$\frac{\partial}{\partial P} = (N + N \cdot \log(N)) \cdot \sum_{i=1}^{N} P(i)$$

$$\begin{aligned}
\frac{\partial}{\partial P} &= 0 \\
0 &= (N + N \cdot \log(N)) \cdot \sum_{i=1}^{N} P(i)
\end{aligned}$$

Se puede ver como en ambas optimizaciones encontramos un mismo punto de inflexión, en ambos casos es $\sum_{i=1}^{N} P(i)$. Además, si nos fijamos en este mismo proceso de optimización, podemos ver como la función de Kullback-Leiber está formada, en parte, por la función del log versimilitud de $Q(i)$.

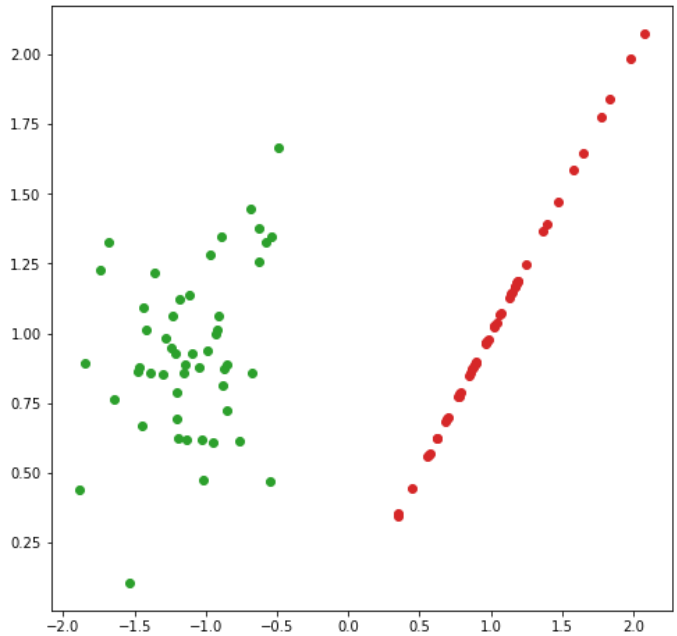
$$
\begin{aligned}
L(\theta) &= -\log(\textstyle\prod_{i=1}^{N} Q(i)) \\
&= \textstyle\sum_{i=1}^{N}(-\log(Q(i)) \\
&= \textstyle\sum_{i=1}^{N}(\log(\frac{1}{Q(i)})
\end{aligned}
\qquad
\begin{aligned}
KL(P|Q) &= \textstyle\sum_{i=1}^{N}(P(i)\cdot\log(\frac{P(i)}{Q(i)}) \\
&= \textstyle\sum_{i=1}^{N}(P(i)\cdot(\log(P(i))-\log(Q(i))) \\
&= \textstyle\sum_{i=1}^{N}(P(i)\cdot(\log(P(i))+\log(\frac{1}{Q(i)}))
\end{aligned}
$$

b) Todo modelo de clasificación es una distribución de probabilidad sobre un conjunto de valores discretos, por lo que podemos ajustar un modelo probabilístico para clasificación haciendo que las probabilidades que obtenga para una muestra se ajusten a las de los datos. Usa la función make_classification de scikit-learn para crear un conjunto de datos de clasificación de dos dimensiones y 100 ejemplos. Tendrás dar un valor 0 al parámetro n_redundant y un valor 1 al parámetro n_clusters_per_class. Da un valor también al parámetro random_state para que los experimentos sean reproducibles. El problema que generará será de clasificación binaria.

Como se puede ver en el Notebook asociado a este mismo documento, el código que se ha usado para este enunciado ha sido el siguiente.

```
from sklearn.datasets import make_classification

b_sample = make_classification( n_samples=100, n_features=2,
                                n_informative=2, n_redundant=0,
                                n_clusters_per_class=1,
                                random_state=RANDOM_STATE)
b_sample_prob = b_sample[0]
b_sample_label = b_sample[1]
b_sample_prob[:5], b_sample_label[:5]
```

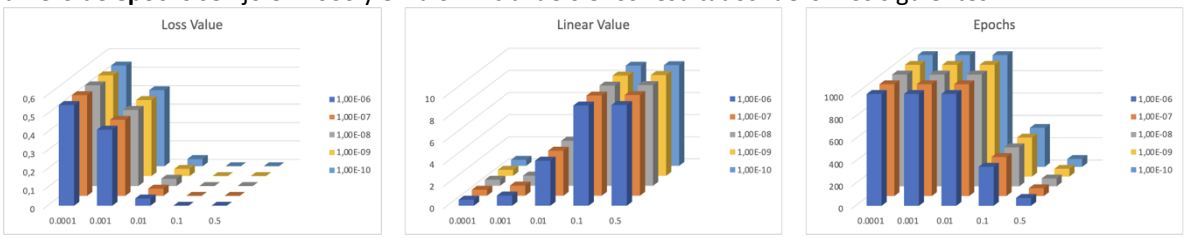
Si ejecutamos este fragmento de código con un valor de 1 en la variable RANDOM_STATE y creamos la gráfica de dispersión asociada, veremos el siguiente resultado.



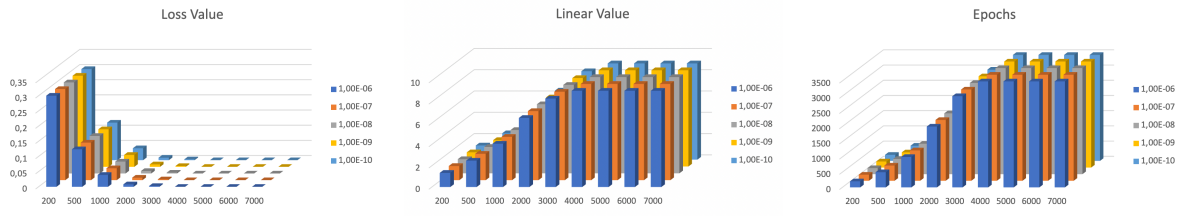
c) Podemos crear un modelo probabilístico con una función linear $f(w,x) = w \cdot x$. Para obtener probabilidades simplemente tenemos que aplicar sobre el resultado una función que de un valor entre 0 y 1. Por ejemplo la función sigmoide σ: $\sigma(x) = \frac{1}{1+e^x}$ A partir de la divergencia de Kullback-Leibler simplificando para problemas binarios podemos llegar a la función de pérdida de entropía cruzada

binaria (binary cross entropy): $BCE(p(x), y) = y \cdot log\big(p(x)\big) + (1 - y) \cdot log(1 - p(x))$ Donde $p(x)$ es la probabilidad que le asigna el modelo a un ejemplo, e y es la etiqueta que le corresponde a los datos. Implementa un algoritmo de descenso de gradiente usando JAX con la función de entropía cruzada binaria. Explora diferentes tasas de aprendizaje. Comenta lo que observes en el comportamiento del error y los parámetros durante la optimización. Escoge un número de iteraciones y un valor para decidir el final de la optimización que te parezcan adecuados.

Se ha decidido probar ejecutar el descenso del gradiente con distintos valores de **learning rate** y **epsilon**. El número de **epochs** se fijó en 1000 y el valor inicial de 0.5 los resultados fueron los siguientes:



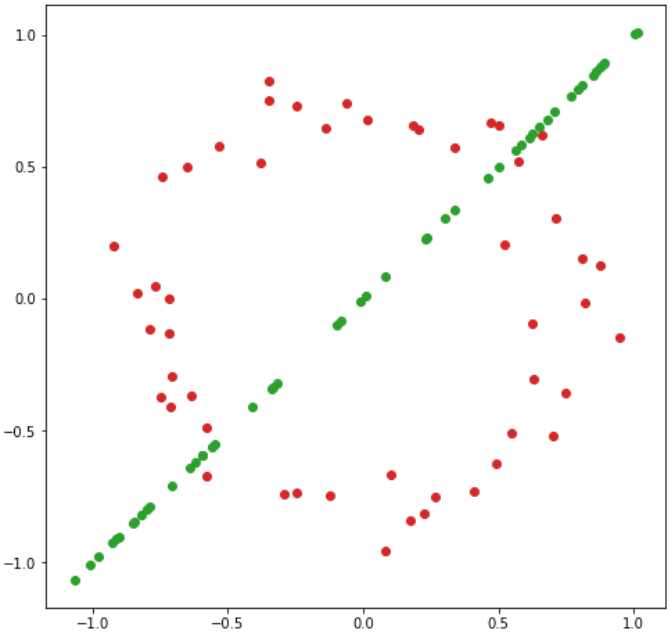
Como se puede ver, si ejecutamos el descenso del gradiente con un **learning rate** superior a 0.01, el valor de la lambda crece muy rápido, y el algoritmo no es capaz de estimar el valor. Entonces, vemos que un ejecutar el descenso del gradiente con un **learning rate** de 0.01 es lo óptimo y veremos como evoluciona el resultado en función del valor de **epsilon** y el número de **epochs**.



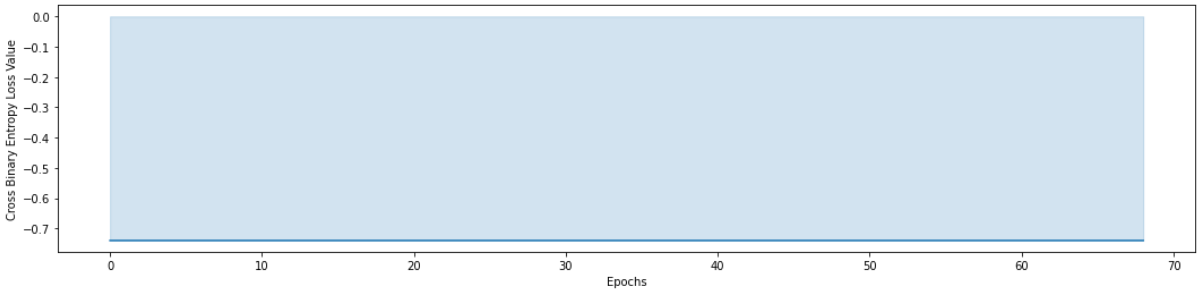
Como se puede ver, nuestro modelo de clasificación binaria no mejora con muchas epochs. Se ha podido ver, como al pasar de las 3400 **epochs**, nuestra función de perdida empieza a dar como valor NaN, por problemas de precisión. Por lo tanto, una posible solución sería operar con tipos de valores que nos permitan más expresión decimal o, por otra parte, tener estos factores en cuenta, y no permitir que nuestro modelo pase de las 3400 **epochs** siempre y cuando tengamos un **learning rate** de 0.01.

d) Genera un conjunto de datos con la función make_circles de scikit-learn usando el valor 0,1 para el parámetro noise. Optimiza el modelo para varios parámetros iniciales diferentes del modelo. Cuenta que esta que sucediendo e intenta explicar el porqué.

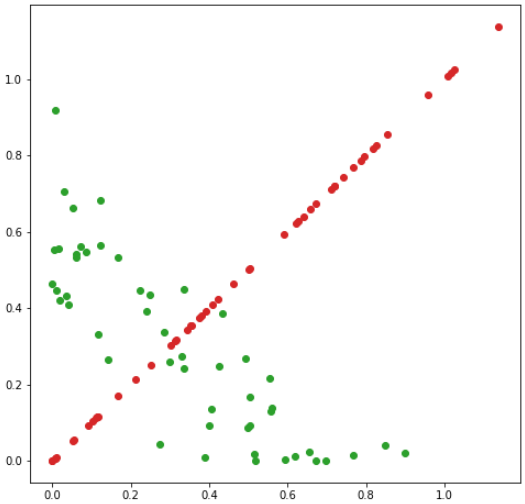
```python
from sklearn.datasets import make_circles

d_sample = make_circles(n_samples=100, noise=0.1,
                        random_state=RANDOM_STATE)
d_sample_prob = d_sample[0]
d_sample_label = d_sample[1]
d_sample_prob[:5], d_sample_label[:5]
```

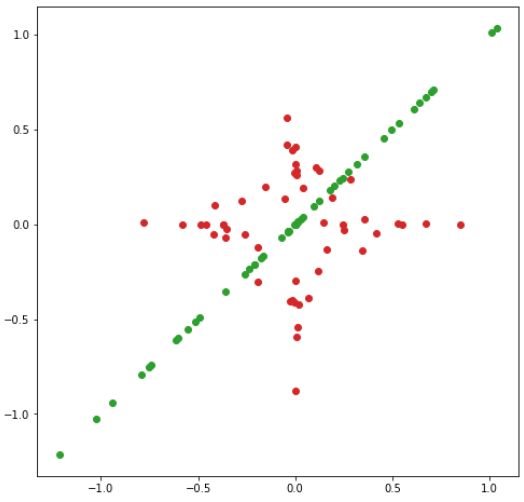


Si ejecutamos el descenso del gradiente con una ecuación lineal, veremos que el modelo no es capaz de mejorar y, en pocas iteraciones, vemos como el valor de la función de perdida no se modifica.

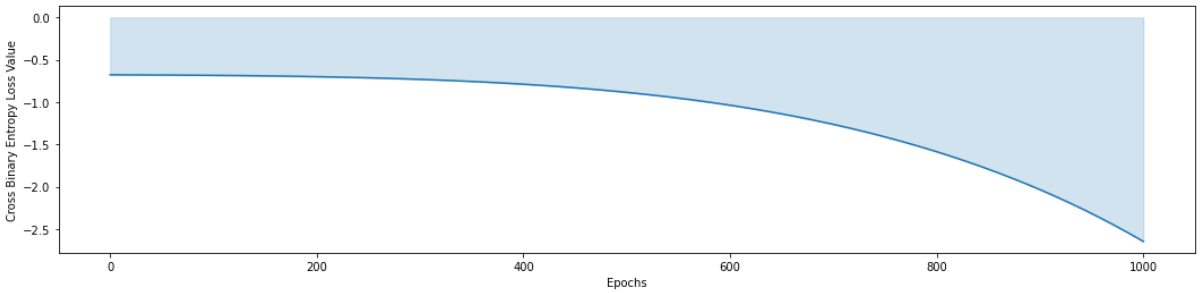


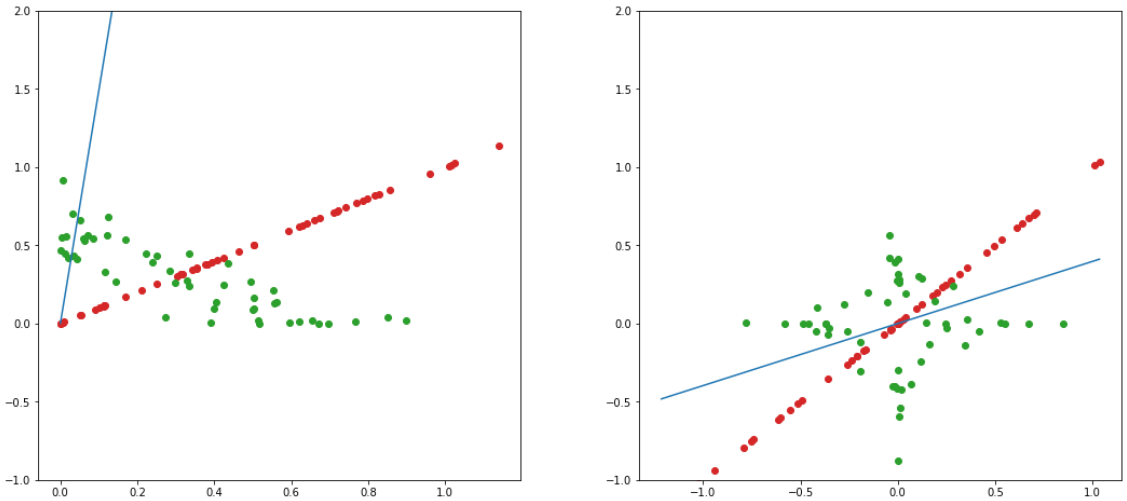
Como posible solución al problema, se prueba elevar las probabilidades al cuadrado y al cubo. Si lo hacemos obtenemos los siguientes gráficos de dispersión.





En el caso de los valores elevados al cuadrado, se ha podido ver como el descenso del gradiente varía el valor de la función lineal y la perdida aumenta. Esto se podría solucionar dotando al modelo de una variable más, y, por lo tanto, se le permitiría ajustar mejor la recta.

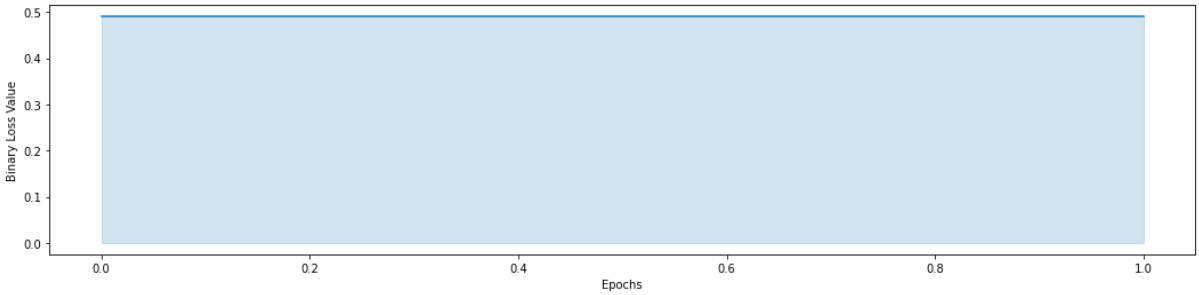


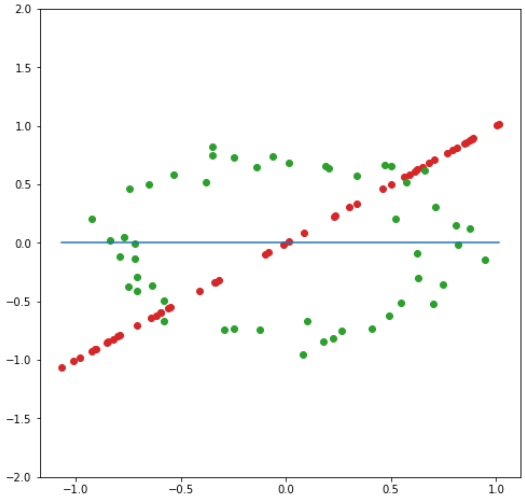


Resumiendo, en este apartado se ha podido ver como nuestro modelo de clasificación binaria solo funciona cuando los grupos que se han de clasificar están muy bien diferenciados, y una recta puede hacer una división limpia entre ellos y, en este caso, al seguir una distribución circular no existe una recta capaz de

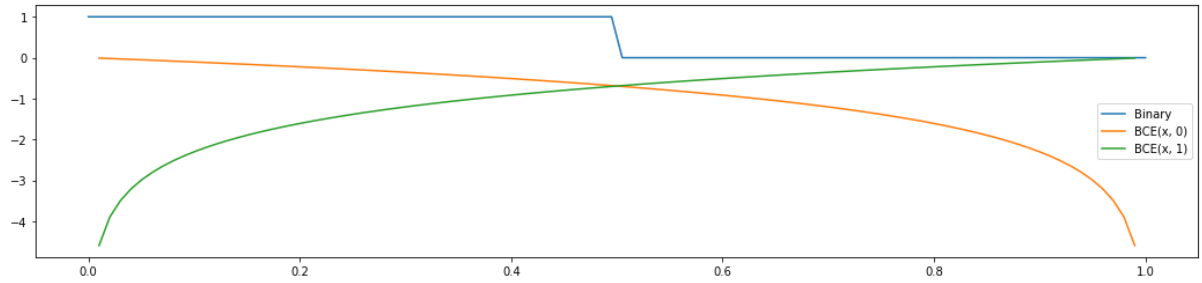
e) La función de entropía cruzada parece una función extraña para optimizar cuando lo que nos interesa es un modelo que tenga el mínimo número de ejemplos mal clasificados. En este caso se correspondería a la función de pérdida 0/1, que en el caso de probabilidades asignaría una pérdida de 0 a valores menores que 0.5 y 1 en caso contrario ¿Porqué no es una buena idea optimizar directamente esta función? Representa las dos funciones.

En este apartado, se desarrolla la función de perdida 0/1, cuya aplicación no mejora los resultados ya obtenidos. Seguimos con una perdida constante, o, en otras palabras, el valor lineal no se modifica en las distintas iteraciones del descenso del gradiente. El resultado al que llegamos es el siguiente:





Si representamos las dos funciones obtenemos las siguientes gráficas:



No es buena idea optimizar con la función de perdida 0/1 porque no le da al modelo información relevante sobre la perdida, ya que es un está bien o está mal, no como de bien o como de mal está esa aproximación.