Notebook_Semana8_Clasificacion

June 29, 2023

1 Clasificación

1.1 Repaso de regresión

Hasta ahora estuvimos viendo problemas de **regresión**, donde la variable *target* (t; por ejemplo, el precio de las casas en el dataset de Properati) es una varible continua.

En las clases anteriores vimos cómo abordar los problemas de regresión utilizando **modelos lineales**, es decir, aquellos donde la respuesta de la variable target es lineal con el cambio de las *características* de los datos. Por ejemplo, si aumenta la superficie de un vivienda en un metro cuadrado, el modelo predice el **mismo aumento en el precio**, independientemente de si pasamos de un monoambiente de 18 m^2 a 19 m^2 o si pasamos de una cinco ambientes de 180 m^2 a 181 m^2.

¿Poco realista? Puede ser, pero recuerden que esta idea sencilla es mucho más flexible de lo que parece, porque podemos incluir **características no lineales**, como "precio al cuadrado", que nos permiten captar relaciones más complejas conservando toda la simpleza matemática de los modelos lineales ( ).

1.2 Un mundo nuevo: clasificación

En los problemas de **clasificación** la variable a modelar es una *categoría* o *clase*.

Ejemplos:

1. Si la propiedad es una casa o un departamento (Properati)
2. El tipo de corte de un diamante (diamonds)
3. La cercanía al océano de un distrito (California) …

En estos casos, también vamos a poder usar **modelos lineales** pero vamos a tener que darles una vuelta de rosca para convertirlos en **modelos lineales generalizados**.

Al igual que venimos haciendo, el objetivo va a ser modelar la variable *target* en función de una serie de *características* (que llamaremos en general X, pero vamos a intentar ser más explícitos en este notebook).

Además, las tareas de clasificación pueden subdividirse según el número de objetivos que tengamos (etiqueta única o etiqueta múltiple) y el número de valores, o *clases*, que puede tomar el objetivo (binario o multiclase), pero estos son temas más avanzados. Arranquemos desde el principio.

1.2.1 Un detalle: codificar las clases

¿Cómo hacemos para que un algoritmo entienda que tiene que separar los datos que son "departamentos" de "casas"? En principio, necesitamos **codificar** estas clases para poder operar con

ellas.

Ya vimos la importancia de poder codificar variables explicativas (**características**) categóricas usando, por ejemplo, `OneHotEncoder`. Acá vamos a hacer algo parecido. Una práctica común, en los casos binarios que vamos a estudiar hoy (es decir, donde solo hay dos clases, que además son excluyentes), es que la variable target tome valores 0 (o -1) y 1 dependiendo de si la instancia es una casa o un depto, respectivamente. Por supuesto, hay arbitrariedad en cuál es la clase 0 y la clase 1; en general, se elijen según las características del problema.

1.3 Clasificación binaria

Old Faithful es un geiser del parque Yellowstone que exhibe una notable regularidad entre erupciones. Las erupciones pueden largar entre 14,000 y 32,000 litros de agua hirviendo a una altura de más de 50 metros.

La duración de las erupciones muestra una distribución bimodal, es decir, que hay erupciones "largas" y "cortas". Veamos los datos de duración en relación con el intervalo entre erupciones (tiempo de espera de cada erupción).

Invento. Vamos a suponer que las erupciones de Old Faithful pueden separarse en dos clases, según el grado de salinidad de las erupciones. Para eso, hay que extraer una muestra del agua de una erupción, y medir su salinidad con alguna técnica de laboratorio.

Preparamos un dataset de duración y tiempo espera entre las erupciones, y agregamos una varible según si la erupción fue clasificada como de "alta salinidad" (clase 1) o "baja salinidad" (clase 0).

[2]:
```
import os

# Bajamos los datos, preparados para la ocasión
DOWNLOAD_URL = "https://github.com/IAI-UNSAM/datasets/raw/master/"
FAITH_URL = DOWNLOAD_URL + "faithful/faithful1.csv"

FAITH_PATH = './datasets'
!mkdir -p ./datasets

def bajar_fathful_data(save_path=FAITH_PATH):
    os.makedirs(save_path, exist_ok=True)
    !wget {FAITH_URL} -P {save_path}

# Corramos la función
bajar_fathful_data()
```

```
--2023-06-29 07:11:30--  https://github.com/IAI-
UNSAM/datasets/raw/master/faithful/faithful1.csv
Resolving github.com (github.com)… 140.82.112.4
Connecting to github.com (github.com)|140.82.112.4|:443… connected.
HTTP request sent, awaiting response… 302 Found
Location: https://raw.githubusercontent.com/IAI-
UNSAM/datasets/master/faithful/faithful1.csv [following]
--2023-06-29 07:11:30--  https://raw.githubusercontent.com/IAI-
```

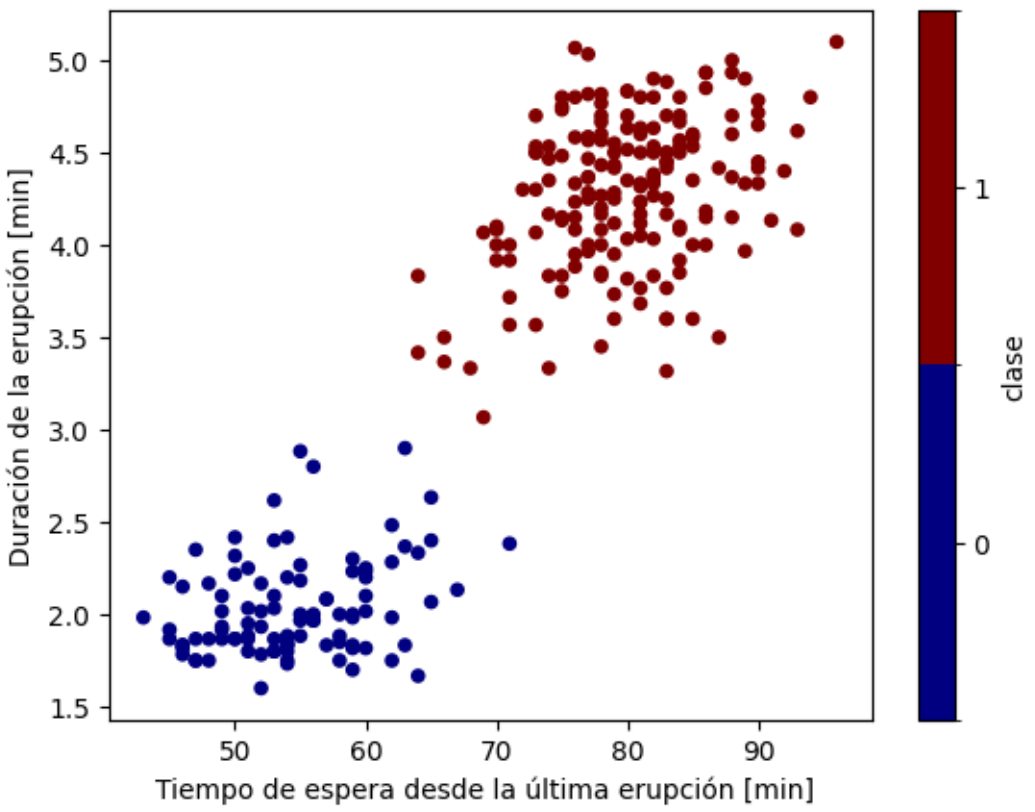
```
UNSAM/datasets/master/faithful/faithful1.csv
Resolving raw.githubusercontent.com (raw.githubusercontent.com)…
185.199.109.133, 185.199.110.133, 185.199.108.133, …
Connecting to raw.githubusercontent.com
(raw.githubusercontent.com)|185.199.109.133|:443… connected.
HTTP request sent, awaiting response… 200 OK
Length: 2821 (2.8K) [text/plain]
Saving to: ‘./datasets/faithful1.csv’

faithful1.csv       100%[===================>]   2.75K  --.-KB/s    in 0s

2023-06-29 07:11:30 (51.9 MB/s) - ‘./datasets/faithful1.csv’ saved [2821/2821]
```

[3]:
```
import pandas as pd

# Leemos los datos que recién bajamos
df = pd.read_csv(os.path.join(FAITH_PATH, 'faithful1.csv'))

# Veamos las primeras líneas
df.head(10)
```

[3]:

	durac_erup	espera	clase
0	3.600	79	1
1	1.800	54	0
2	3.333	74	1
3	2.283	62	0
4	4.533	85	1
5	2.883	55	0
6	4.700	88	1
7	3.600	85	1
8	1.950	51	0
9	4.350	85	1

[4]:
```
# Plot del dataset
df.loc[:, 'clase'] = pd.Categorical(df.clase)

def plot_data():
    _ = df.plot.scatter(x='espera', y='durac_erup', c='clase',
                        xlabel='Tiempo de espera desde la última erupción [min]␣
 ↪',
                        ylabel='Duración de la erupción [min]',
                        cmap='jet')
    return

plot_data()
```



Vemos fácilmente que las erupciones de salinidad diferente se separan en este plano. Podemos entonces pensar en un modelo que a partir del tiempo de espera y la duración de la erupción **clasifique** a los eventos de Old Faithful y evitarnos tener que medir la salinidad.

¿Cómo lo hacemos?

1. Gráficamente.
2. Con una receta escrita explícitamente.

Buscamos los valores concretos de la pendiente (m) y la ordenada al origen (b):

$$\text{duración} = m \times \text{tiempo de espera} + b$$

3. Con un método automático.

Para esto, conviene expresar la recta de arriba como

$$\omega_0 + \omega_1 \times \text{tiempo de espera} + \omega_2 \times \text{duración} = 0$$

Entonces, la **frontera de decisión** es el lugar del espacio de los datos en el que la **función de decisión**

$$y(\text{datos}, \omega) = \omega_0 + \omega_1 \times \text{tiempo de espera} + \omega_2 \times \text{duración}$$

vale 0.

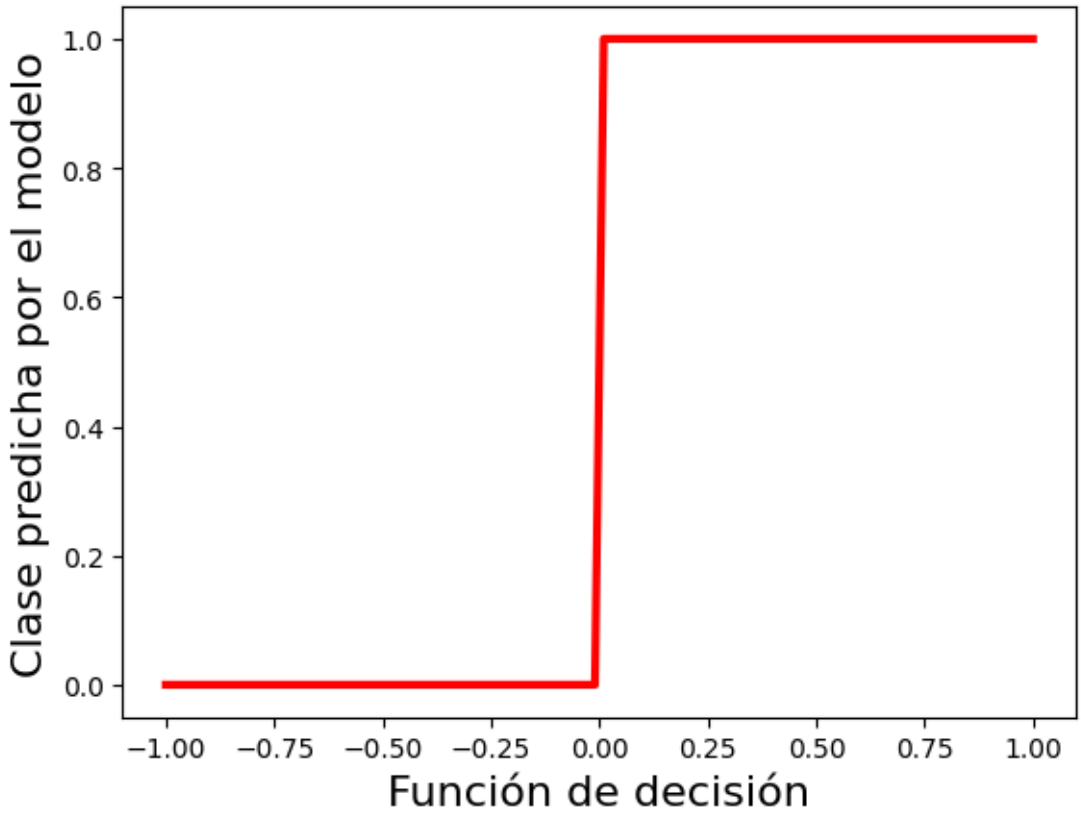
Así, cada punto tiene un valor de y. Hay dos propiedades de la función de decisión que son importantes:

1. de un lado de la frontera los valores de la función de decisión, y son positivos y del otro son negativos.
2. cuanto más lejos está un punto de la frontera de decisión, mayor es el valor absoluto de y.

Entonces, para armar un algoritmo que clasifica las instancias de un set de datos, podemos aplicar una **función escalón** a la **función de decisión**.

[5]:
```
import numpy as np
import matplotlib.pyplot as plt

# Vector para el gráfico
x = np.linspace(-1, 1, 100)

# Función escalón, también conocida como theta de Heaviside
# ver https://es.wikipedia.org/wiki/Funci%C3%B3n_escal%C3%B3n_de_Heaviside
y = np.heaviside(x, 0.5)

plt.plot(x, y, lw=3, color='r')
plt.xlabel('Función de decisión', size=16)
plt.ylabel('Clase predicha por el modelo', size=16)
```

[5]:
```
Text(0, 0.5, 'Clase predicha por el modelo')
```



En pocas palabras, el método es:

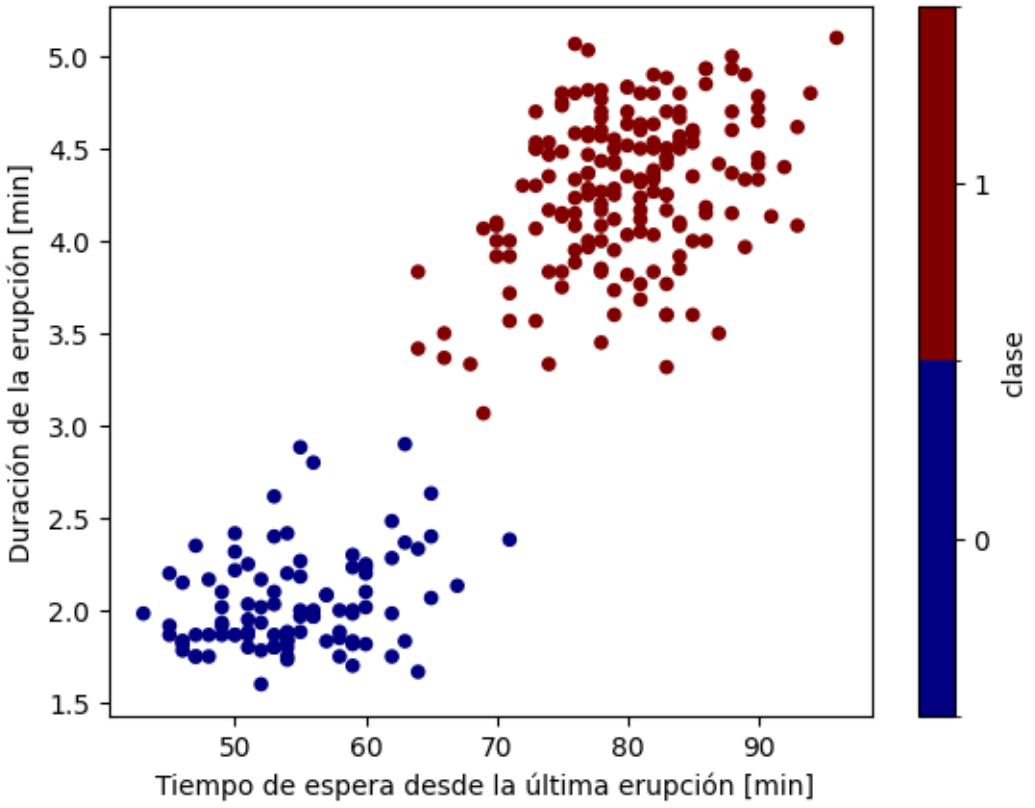
1. Encontrar la mejor frontera de decisión (ver más abajo cómo hacer esto) con el conjunto de entrenamiento.
2. Con los parámetros ajustados, calcular el valor de y para un datapoint que se quiera clasificar.
3. Si $y > 0$, clasificar como clase 1; de lo contrario, clasificar como clase 0.

1.4 Perceptrón

La pregunta que sigue es: ¿cómo hacemos para encontrar de forma automática los valores de los parámetros $\omega 0$, $\omega 1$ y $\omega 2$ que crean la mejor frontera de decisión?

Esta pregunta está asociada a la elección de una **función de error**. ¿Qué creen que pasaría si eligiéramos como función de error al MSE o MAE?

[6]:
```
plot_data()
```



1.4.1 La función de error

Los problemas de clasificación tienen funciones de error propias. Cuando usamos una función escalón como describimos arriba, lo más común es usar la **función de error del perceptrón**, que básicamente es igual a la suma de las distancias a la frontera de decisión de los puntos mal clasificados (conjunto que llamamos $\mathcal{M}$).

$$E_p = \sum_{i \in \mathcal{M}} |y(\text{datos}_i, \omega)| \ .$$

El módulo es complicado si uno quiere derivar e igualar a cero para encontrar el mínimo de esta función, pero hay formas astutas de escribir lo mismo, sin usar el módulo.

1.4.2 Implementación en `scikit-learn`

[7]:
```
from sklearn.linear_model import Perceptron

# Instancio un perceptrón con los parámetros por defecto
perce = Perceptron()
```

[8]:
```
# Preparo la data en formato que entienda el bicho

# Separo el target
clase = df.clase.to_numpy()

# Tiro el target de la data
datos = df.drop(['clase'], axis=1, inplace=False)
# Reordeno los datos
datos = datos[['espera', 'durac_erup']]

# Los normalizo, porque los valores son grandes
from sklearn.preprocessing import StandardScaler
stsc = StandardScaler()
datosn = stsc.fit_transform(datos)

# Chequeo
# datosn.columns
```

[9]:
```
# Ajusto el perceptron
perce.fit(datosn, clase)
```

[9]:
```
Perceptron()
```

Al igual que con los otros algoritmos, podemos ver los coeficientes y el "intercept"

[10]:
```
print('Omega_0 = {:.2f}'.format(*perce.intercept_))
print('Omega_1 = {:.2f}'.format(perce.coef_[0][0]))
print('Omega_2 = {:.2f}'.format(perce.coef_[0][1]))
```

```
Omega_0 = 1.00
Omega_1 = 1.19
Omega_2 = 1.44
```

1.4.3 Funciones útiles

Definamos algunas funciones útiles para graficar fronteras de decisión en dos dimensiones y para obtener el vector de los pesos del modelo lineal.

[11]:
```
import numpy as np
import matplotlib.pyplot as plt

def plot_clasi(x, t, ws, labels=[], xp=[-1,1], spines='zero',
               equal=True, join_centers=False, margin=None, **kwargs):
    """
    Plot results of linear classification problems.
    :param np.array x: Data matrix
    :param np.array t: Label vector.
```

```
    :param list or tuple ws: list with fitted paramter vector of models, one␣
↪element per model
    :param tuple xp: start and end x-coordinates of decision boundaries and
                     margins.
    :param str or None spines: whether the spines go through zero. If None,
                               the default behaviour is used.
    :param bool equal: whether to use equal axis aspect (default=True;
                       recomended to see the parameter vector normal to
                       boundary)
    :param bool join_centers: whether to draw lines between classes centres.
    :param None or tuple margin: tupler of booleans that define whether
                                 to plot margin for each model being plotted.
                                 If None, False for all models.
    """
    assert len(labels) == len(ws) or len(labels) == 0

    if margin is None:
        margin = [False] * len(ws)
    else:
        margin = np.atleast_1d(margin)
    assert len(margin) == len(ws)

    if len(labels) == 0:
        labels = np.arange(len(ws)).astype('str')

    # Agregemos el vector al plot
    fig = plt.figure(figsize=(9, 7))
    ax = fig.add_subplot(111)

    xc1 = x[t == np.unique(t).max()]
    xc2 = x[t == np.unique(t).min()]

    ax.plot(*xc1.T, 'or', mfc='None', label='C1')
    ax.plot(*xc2.T, 'ob', mfc='None', label='C2')

    for i, w in enumerate(ws):
        # Separa el sesgo del resto de los pesos
        thr = -w[0]
        w = w[1:]
        # Calcula la norma del vector
        wnorm = np.sqrt(np.sum(w**2))

        # Ploteo vector de pesos
        ax.quiver(0, thr/w[1], w[0]/wnorm, w[1]/wnorm,
                  color='C{}'.format(i+2), scale=10, label=labels[i],
                  zorder=10)
```

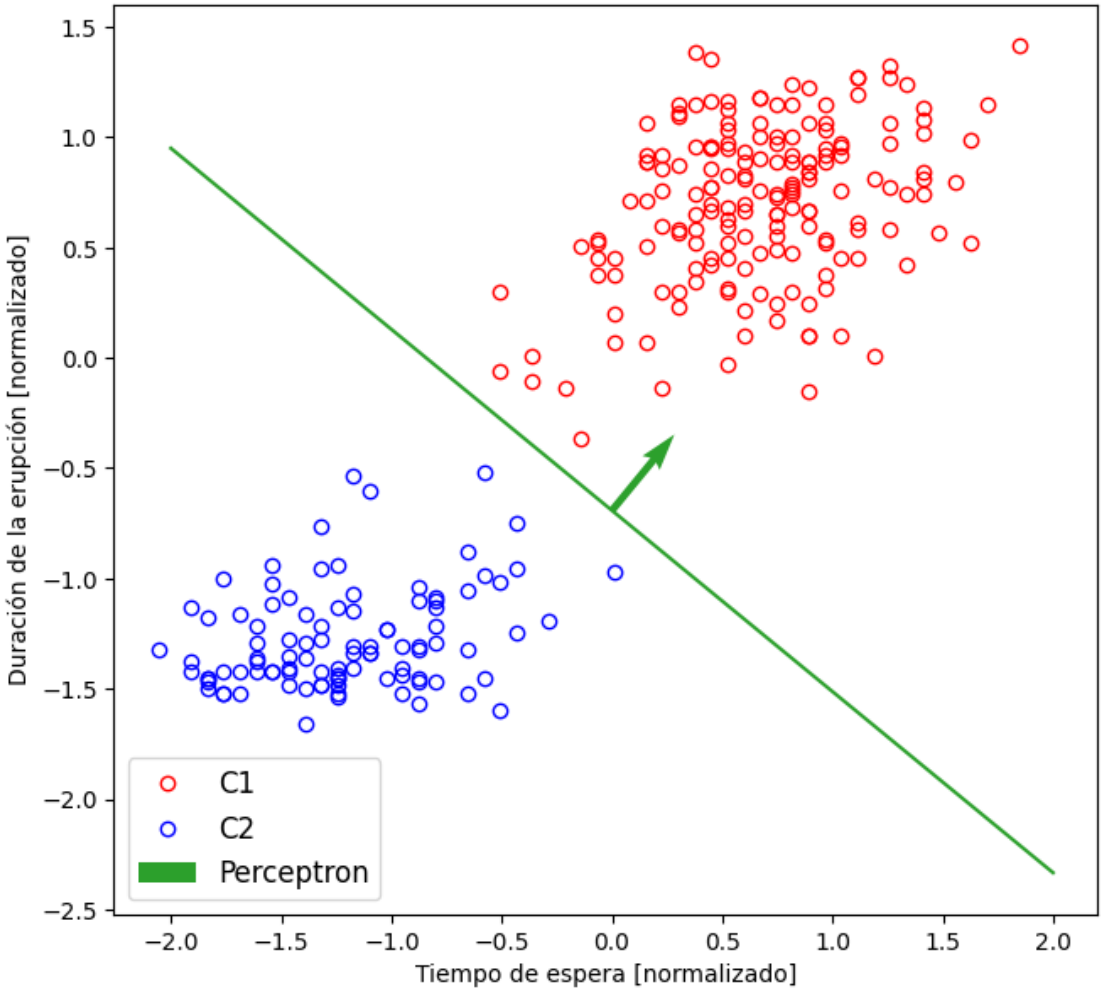
```
        # ploteo plano perpendicular
        xp = np.array(xp)
        yp = (thr - w[0]*xp)/w[1]

        plt.plot(xp, yp, '-', color='C{}'.format(i+2))

        # ploteo el margen (para SVC)
        if margin[i]:
            for marg in [-1, 1]:
                ym = yp + marg/w[1]
                plt.plot(xp, ym, ':', color='C{}'.format(i+2))

    if join_centers:
        # Ploteo línea que une centros de los conjuntos
        mu1 = xc1.mean(axis=1)
        mu2 = xc2.mean(axis=1)
        ax.plot([mu1[0], mu2[0]], [mu1[1], mu2[1]], 'o:k', mfc='None', ms=10)

    ax.legend(loc=0, fontsize=12)
    if equal:
        ax.set_aspect('equal')

    if spines is not None:
        for a in ['left', 'bottom']:
            ax.spines[a].set_position('zero')
        for a in ['top', 'right']:
            ax.spines[a].set_visible(False)

    for k in kwargs:
        print(k, kwargs[k])
        getattr(ax, 'set_'+k)(kwargs[k])

    return


def makew(fitter, norm=False):
    """
    Prepare parameter vector for an sklearn.liner_model predictor.

    :param sklearn.LinearModel fitter: the model used to classify the data
    :param bool norm: default: False; whether to normalize the parameter vector
    """
    # # Obtengamos los pesos
    w = fitter.coef_.copy()

    # # Incluye intercept
    if fitter.fit_intercept:
```

```
        w = np.hstack([fitter.intercept_.reshape(1, 1), w])

    # # Normalizon
    if norm:
        w /= np.linalg.norm(w)
    return w.T


def plot_fundec(fitter, x, t):
    from matplotlib import colors

    plt.figure(figsize=(9, 7))

    xx, yy = np.meshgrid(np.linspace(x[:, 0].min()-1, x[:, 0].max()+1, 200),
                         np.linspace(x[:, 1].min()-1, x[:, 1].max()+1, 200))

    # evaluate decision function
    Z = fitter.decision_function(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    # veamos la función de decisión y la frontera de decisión
    mynorm = colors.TwoSlopeNorm(vmin=Z.min(), vmax=Z.max(), vcenter=0.0)
    pme = plt.pcolormesh(xx, yy, Z, cmap=plt.cm.RdBu_r, norm=mynorm,␣
 ↪shading='auto')
    plt.colorbar(label='Función de decisión')
    plt.contour(xx, yy, -Z, 0, colors='0.5', zorder=1)
    # plt.contour(xx, yy, -Z, [-1, 1], colors='0.25', linestyles='dashed',␣
 ↪zorder=1)

    xc1 = x[t == np.unique(t.flatten()).max()]
    xc2 = x[t == np.unique(t.flatten()).min()]

    plt.plot(*xc1.T, 'or', mfc='None', label='C1')
    plt.plot(*xc2.T, 'ob', mfc='None', label='C2')

    plt.xticks(())
    plt.yticks(())
    plt.axis('tight')

    return pme
```

[12]:
```
# Creo el vector de pesos
w_perce = makew(perce)

plot_clasi(datosn, t=clase, ws=[w_perce,], xp=[-2, 2], labels=['Perceptron', ],
           spines=None,
           xlabel='Tiempo de espera [normalizado]',
```

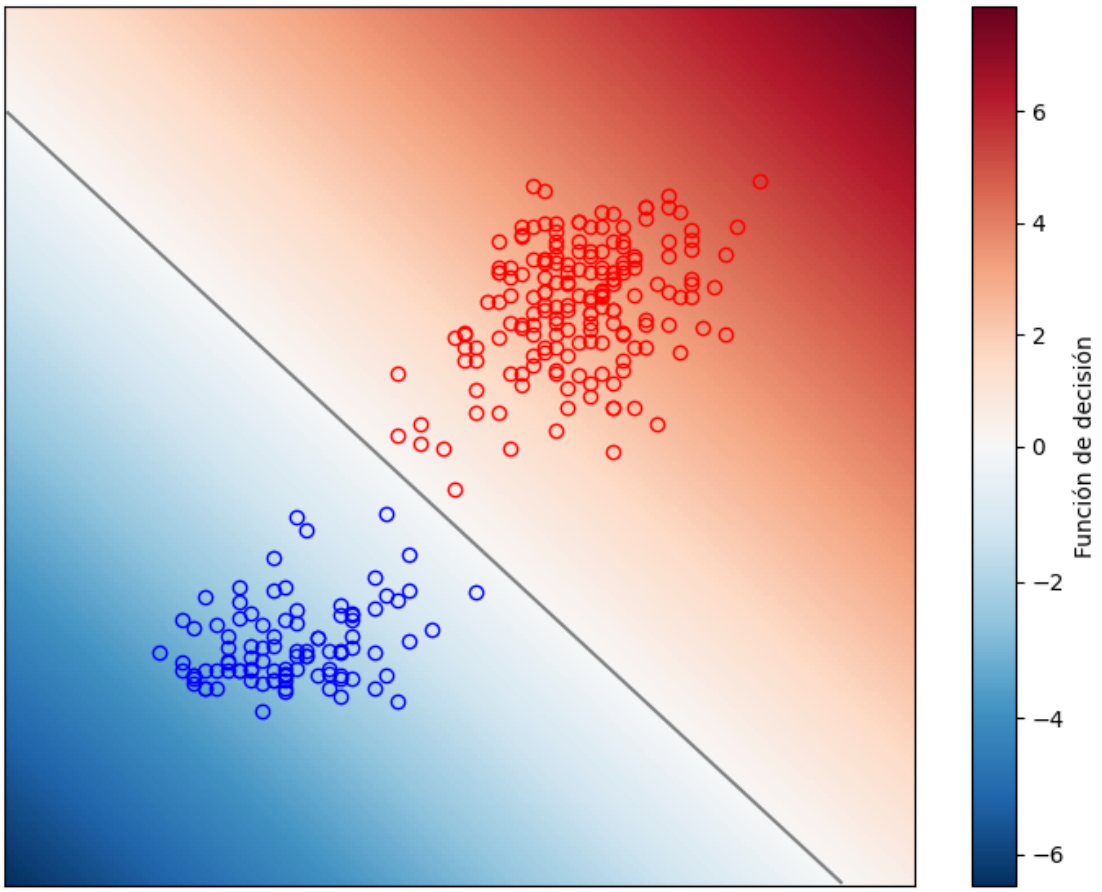
```
            ylabel='Duración de la erupción [normalizado]')
```

```
xlabel Tiempo de espera [normalizado]
ylabel Duración de la erupción [normalizado]
```



[13]:
```
plot_fundec(perce, datosn, clase)
```

[13]:
```
<matplotlib.collections.QuadMesh at 0x7f2dfa50d9f0>
```



Al perceptron también se le puede pedir que prediga a qué clase pertenece cada instancia

[14]:
```
# predicciones del perceptron
preds = perce.predict(datosn)
```

1.4.4 Dependencia con las condiciones iniciales

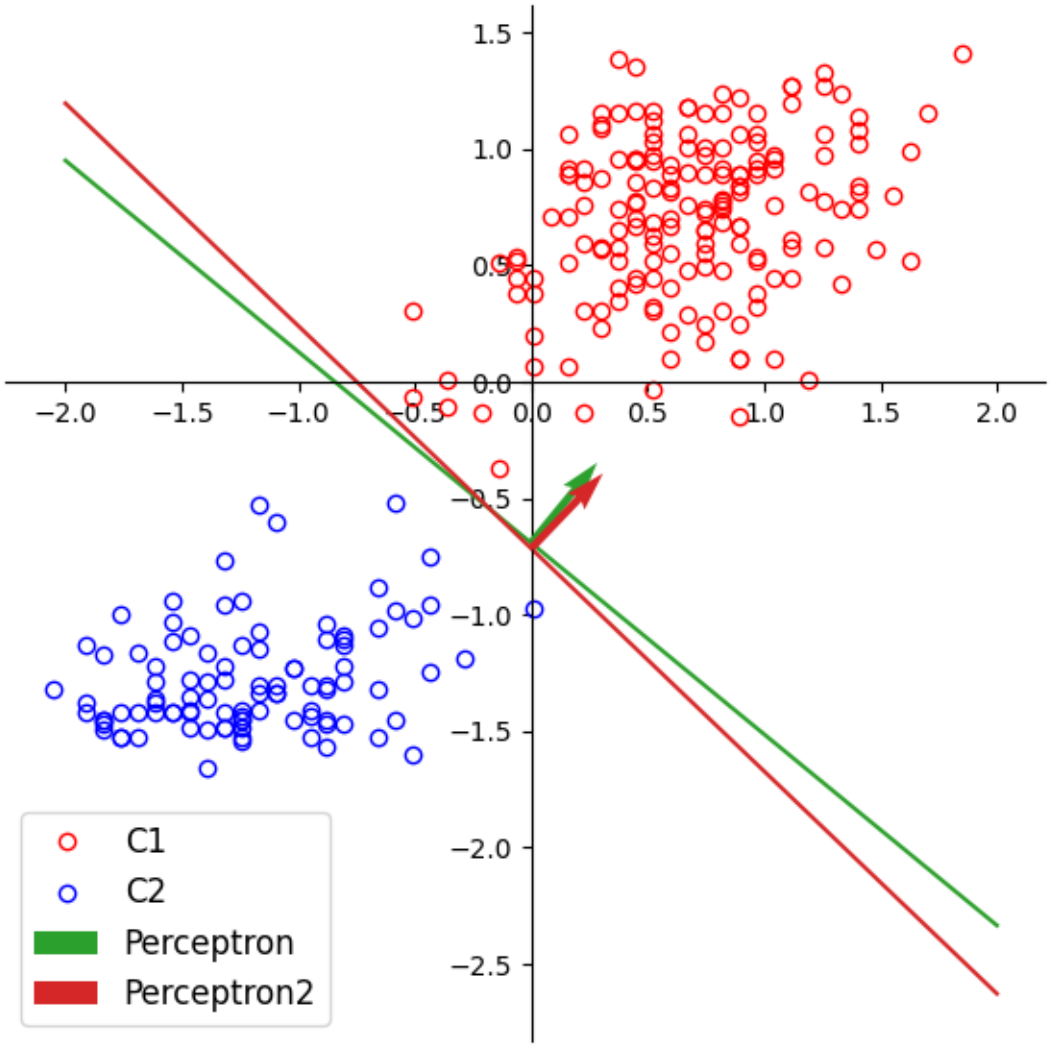
Una característica del algoritmo planteado más arriba es que podemos converger a distintas soluciones, según el punto inicial del que partamos.

El punto inicial puede elegirse de forma aleatoria o podemos fijarlo usando los argumentos `coef_init` e `intercept_init` del método `fit`:

[15]:
```
# Definimos un punto inicial
w0 = np.array([5.0, 5.0]).reshape(1, 2)
intercept = np.array([-2,])

# Ajustamos arrancando de esa posición.
perce.fit(datosn, clase, coef_init=w0, intercept_init=intercept)
w_perce2 = makew(perce)
```

Y podemos graficar ambas soluciones:

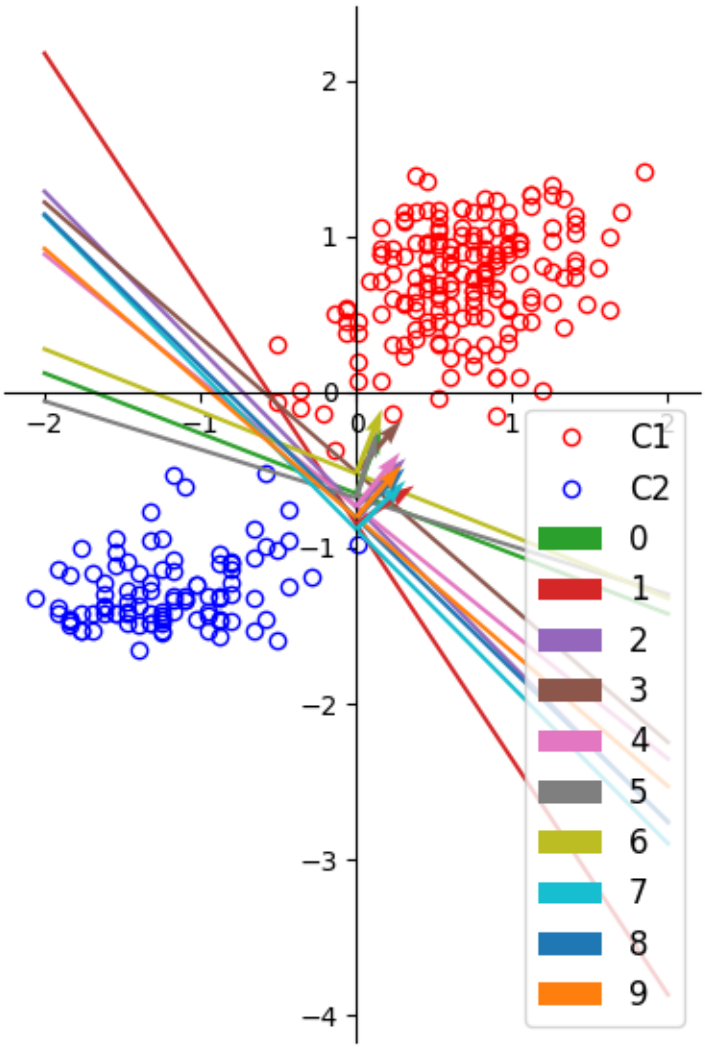
[16]:
```
plot_clasi(datosn, clase, [w_perce, w_perce2], ['Perceptron', 'Perceptron2'],␣
↪xp=[-2, 2])
```



O podemos ver soluciones múltiples

[17]:
```
w0s = np.random.rand(10, 2) * 5.0
intercepts = np.random.rand(10) * 5.0

wlist = []
thlist = []
for i in range(len(w0s)):
    perce.fit(datosn, clase, coef_init=w0s[i], intercept_init=intercepts[i])
    w_p = makew(perce)
```

```
    wlist.append(w_p)

plot_clasi(datosn, clase, wlist, xp=[-2, 2])
```



1.5 Métricas

Para decidir cuán bien funciona nuestro clasificador, necesitamos definir alguna métrica. Las siguientes son opciones comunes:

- Exactitud (accuracy): ¿Qué fracción de los puntos están clasificados correctamente?
- Precisión: ¿Cuántos puntos de los que predecimos que pertenecen a la Clase 1, realmente son de esa clase? Por supuesto, podemos también definir la precisión para la Clase 2 de manera análoga.
- Exhaustividad (recall): ¿Qué fracción de los puntos de la Clase 1 son clasificados como Clase 1?

Dependiendo de qué nos interese más, y cuán balanceado esté nuestro dataset (balance entre una clase y otra) eligiremos una u otra métrica. En el caso en el que estamos interesados en la precisión **Y** la exhaustividad, podemos considerar la media armónica de ambas:

- socre F1: media armónica de la precisión y la exhaustividad.

Esta forma de calcular la media le da más peso al menor valor, de manera que un score F1 alto significa que tanto la precisión como la exhaustividad son altas.

Todas estas métricas se pueden calcular a partir de la *matriz de confusión*:

	Predicción C2 (-)	Predicción C1 (+)
Etiqueta C2 (-)	Verdaderos Negativos (VN)	Falsos Positivos (FP)
Etiqueta C1 (+)	Falsos Negativos (FN)	Verdaderos Positvos (VP)

$$\text{Exactitud} = \frac{VP + VN}{VP + VN + FP + FN}$$

$$\text{Precisión} = \frac{VP}{VP + FP}$$

$$\text{Exhaustividad} = \frac{VP}{VP + FN}$$

Por supuesto, todo esto está implementado en `sklearn`, en el paquete `metrics`.

[18]:
```
from sklearn.metrics import confusion_matrix, accuracy_score, precision_score,␣
↪recall_score

print(f"Exactitud: {round(accuracy_score(clase, preds), 2)}")
print(f"Precisión: {round(precision_score(clase, preds), 2)}")
print(f"Exhaustividad: {round(recall_score(clase, preds), 2)}")
print("Matriz de confusión:")
print(confusion_matrix(clase, clase))
```

```
Exactitud: 1.0
Precisión: 1.0
Exhaustividad: 1.0
Matriz de confusión:
[[ 97   0]
 [  0 175]]
```

Como es de esperar viendo los gráficos de arriba, en estos casos el funcionamiento del algoritmo es perfecto.

1.6 *Descenso de gradiente estocástico (SGD)

Un algoritmo de optimización frecuentemente utilizado es el *descenso de gradiente estocástico* (*stochastic gradient descent* o SGD). Se utiliza para encontrar los valores de las componentes del vector ω que minimiza una determinada función de pérdida $E(\omega)$.

En general, el algoritmo de descenso de gradiente consiste en modificar los pesos ω en la dirección que reduce más rápidamente $E(\omega)$. Localmente, esta dirección viene dada por $\nabla_\omega E(\omega)$, por lo que el proceso iterativo es

$$\omega \to \omega - \eta \nabla_\omega E(\omega),$$

donde η establece la escala de *cuánto* debemos movernos en esa dirección, y se llama la *tasa de aprendizaje*. El valor de esta tasa de aprendizaje es un hiperparámetro que debe ser optimizado (normalmente menos de 1). Para valores pequeños, se necesitarán más iteraciones para converger, pero para valores grandes el proceso iterativo podría oscilar alrededor de un mínimo o incluso empezar a diverger.

El primer valor de ω se elige generalmente al azar, y el algoritmo procede hasta la convergencia, que se define fijando un umbral de tolerancia, o hasta que se alcanza un número máximo de iteraciones (en cuyo caso decimos que el algoritmo no ha convergido).

En el descenso de gradiente estocástico, necesitamos evaluar el gradiente de la función de pérdida en la posición del vector de pesos ω *en cada paso* (ya que los pesos se están modificando), y la función de pérdida suele ser una media de todo un conjunto de datos. Por esta razón, necesitamos una función de pérdida que podamos diferenciar analíticamente, ya que, de lo contrario, el cálculo numérico del gradiente sería demasiado caro.

Para conjuntos de datos grandes, sólo podemos iterar una vez que hemos calculado la función de pérdida en todo el conjunto de datos ($E(\omega) = \frac{1}{N}\sum_i e(x^{(i)}, y^{(i)}; \omega)$), lo que puede llevar algún tiempo.

Para acelerar la convergencia, se puede estimar la función de pérdida en un solo punto de datos, y mover los pesos en la dirección de $-\nabla_\omega e(x^{(i)}, y^{(i)}; \omega)$ para cada punto del conjunto de datos. De esta manera, en el coste computacional de calcular este gradiente en todo el conjunto de datos, nuestros pesos se han iterado N veces. La contrapartida es que, en cada punto de datos, uno está sesgado hacia la aleatoriedad de ese punto específico, y por lo tanto hace que la convergencia sea estocástica (similar a un paseo aleatorio). Este método se conoce como *Descenso Gradiente Estocástico* (SGD). La aleatoriedad de la convergencia puede ayudar al algoritmo a escapar de los mínimos locales en problemas no convexos.

En la práctica, se utiliza un algoritmo intermedio llamado *Descenso Gradiente Estocástico por Lotes*, que utiliza subconjuntos (*lotes*) de todo el conjunto de datos para estimar la función de error, manteniendo parte de la estocasticidad de SGD pero haciendo uso de la paralelización para acelerar el cálculo (calculando cada punto del lote en paralelo, ya que estos cálculos son independientes).

1.7 *Linealmente separable. ¿Qué significa?

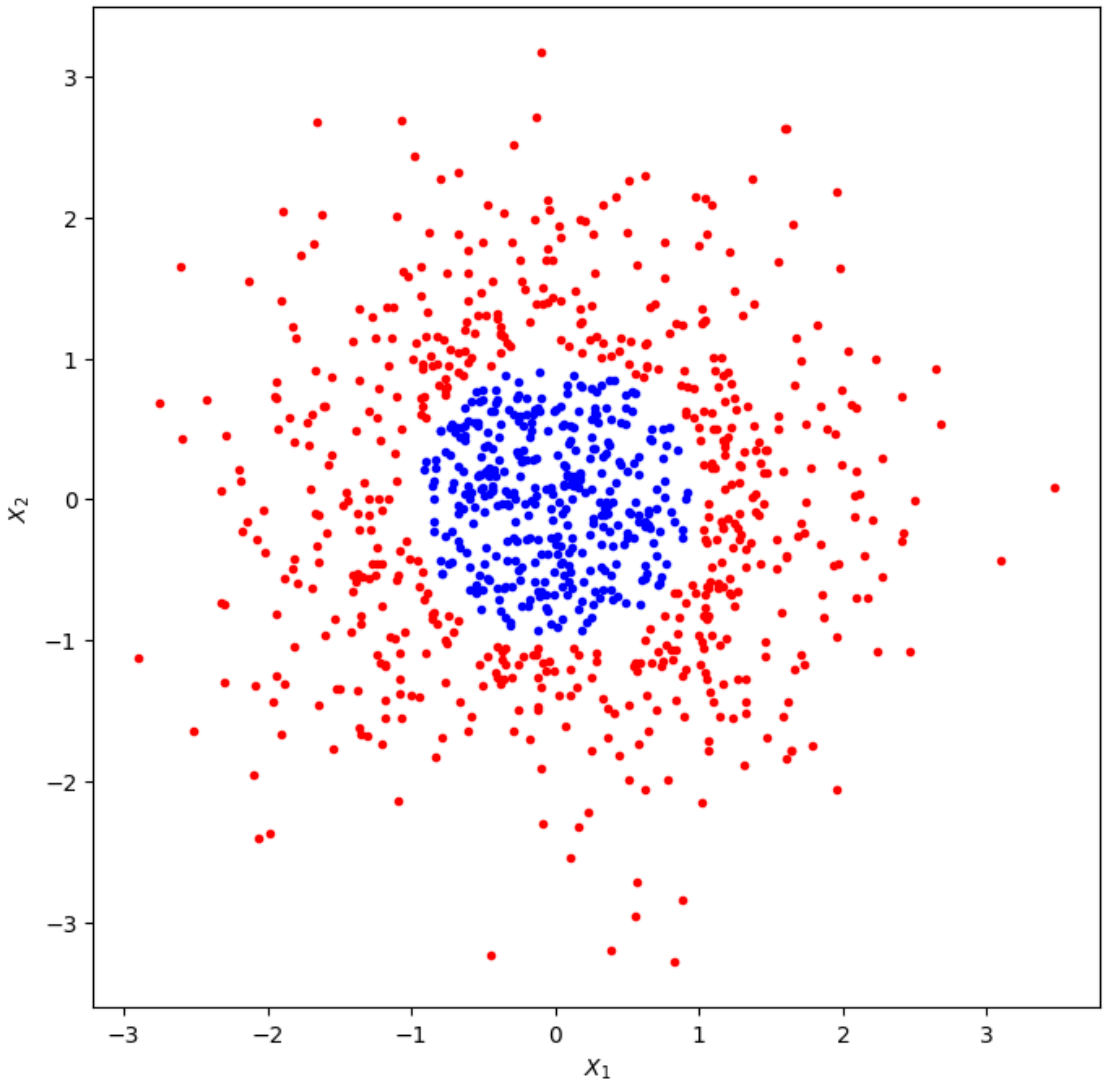
Empecemos con un dataset bastante particular. Lo generamos con la celda a continuación y lo graficamos después.

[19]:
```
def make_circle_data():

    np.random.seed(2022)
```

```
    x = np.random.multivariate_normal(mean=[0,0], cov=[[1, 0],[0, 1]],␣
 ↪size=1000).T
    r = (x[0]**2 + x[1]**2)

    xc1 = x[:, r < 0.9]
    xc2 = x[:, r > 1.1]

    # Uso coordenadas polares
    rc1 = np.sqrt(xc1[0]**2 + xc1[1]**2)
    thetac1 = np.arctan2(xc1[1], xc1[0])
    rc2 = np.sqrt(xc2[0]**2 + xc2[1]**2)
    thetac2 = np.arctan2(xc2[1], xc2[0])

    phi1 = np.vstack([rc1[np.newaxis, :], thetac1[np.newaxis, :]])
    phi2 = np.vstack([rc2[np.newaxis, :], thetac2[np.newaxis, :]])

#     phi1 = rc1[np.newaxis, :]
#     phi2 = rc2[np.newaxis, :]

    # Creamos la matrix de diseño completa y los labels
    t1 = np.ones_like(rc1)
    t2 = np.zeros_like(rc2)

    return xc1, xc2, phi1, phi2, t1, t2

xc1, xc2, phi1, phi2, t1, t2 = make_circle_data()
```

[20]:
```
plt.figure(figsize=(8, 8))
plt.plot(*xc1, '.b')
plt.plot(*xc2, '.r')
plt.xlabel('$X_1$')
plt.ylabel('$X_2$')
```

[20]:
```
Text(0, 0.5, '$X_2$')
```



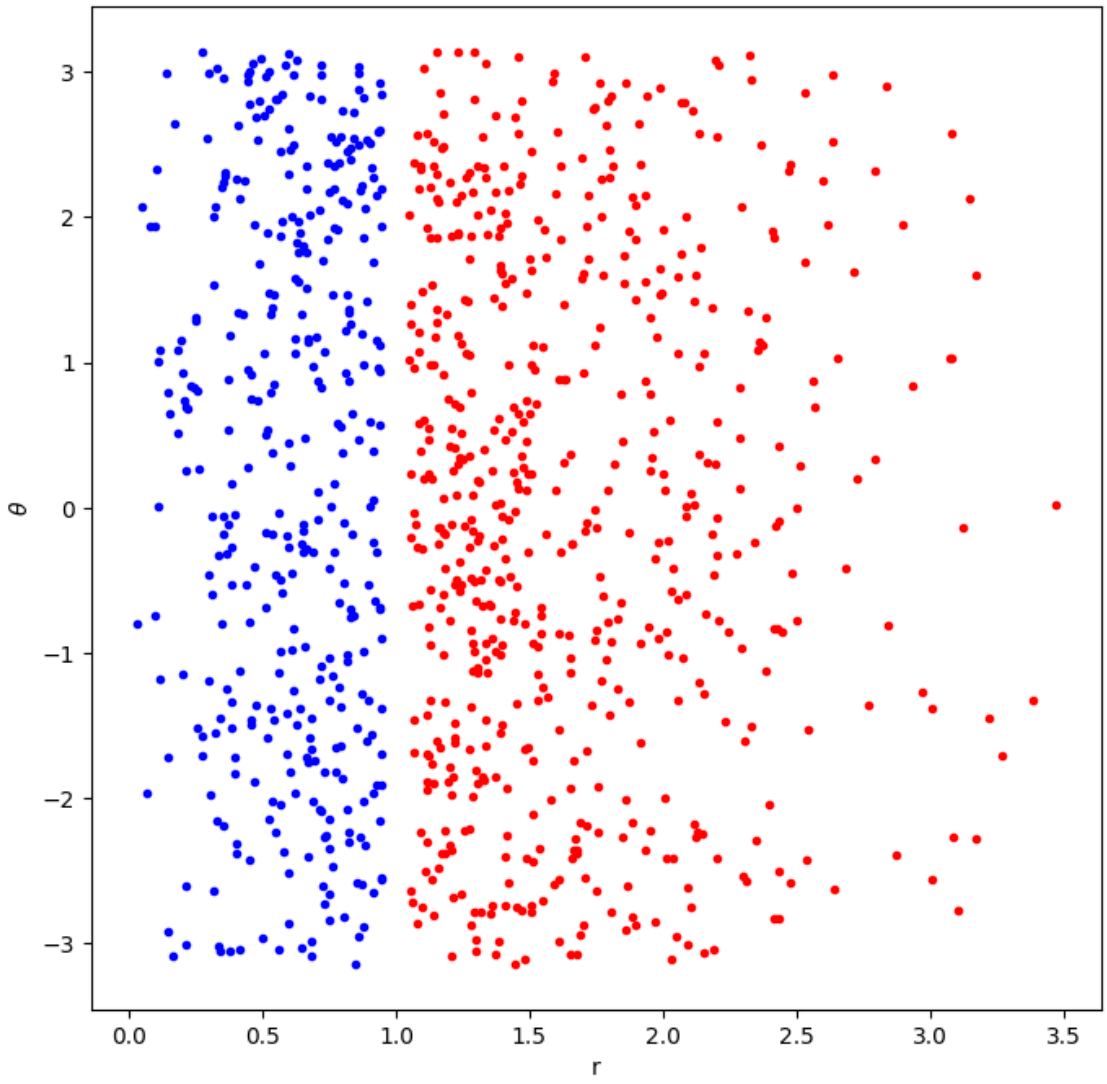
Preguntas * ¿Podemos trazar una recta que separe ambas clases? * ¿Podemos trazar una **curva** que lo haga? * ¿Dirían que este conjunto es *linealmente separable*?

El mismo conjunto puede representarse usando otras coordenadas (*features*), las coordenadas polares, donde cada punto está identificado por su distancia al origen (*radio*) y por su ángulo con respecto a la dirección positiva del eje x (*ángulo polar*)

Veamos a este dataset en esas coordenadas.

[21]:
```
plt.figure(figsize=(8, 8))
plt.plot(phi1[0], phi1[1], '.b')
plt.plot(phi2[0], phi2[1], '.r')
plt.xlabel('r')
plt.ylabel(r'$\theta$')
```

[21]: Text(0, 0.5, '$\\theta$')

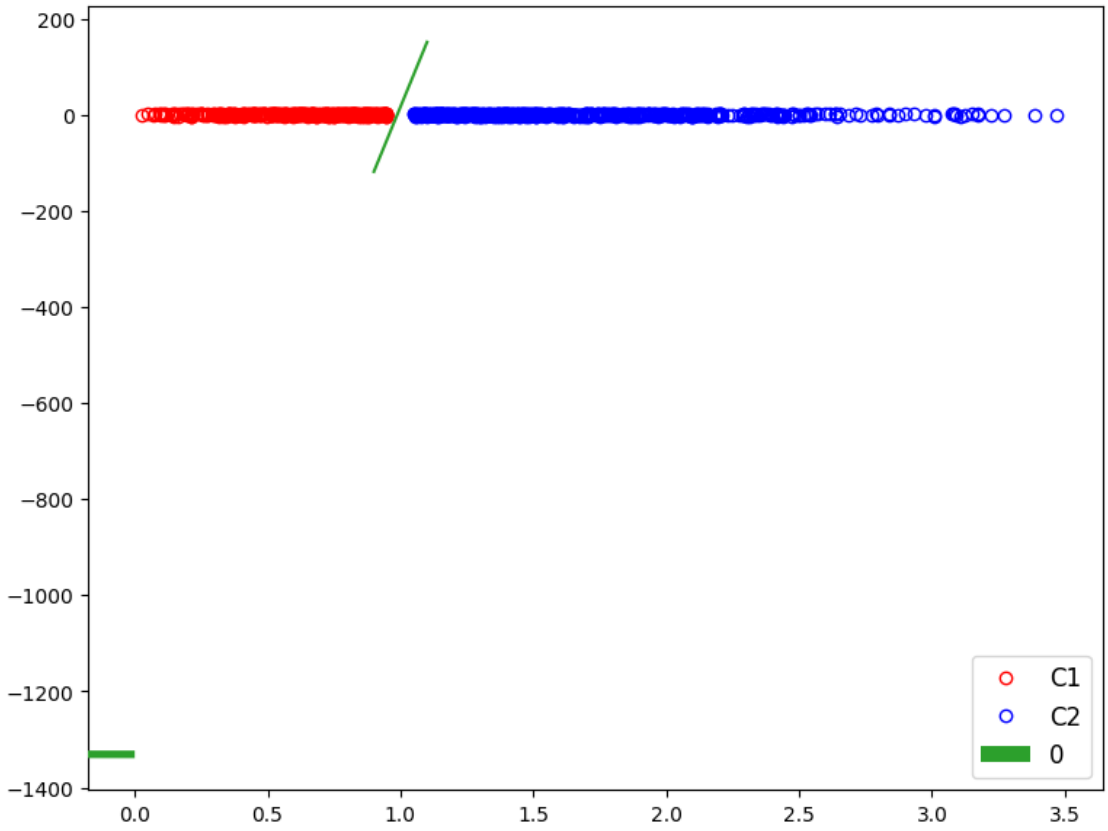


- ¿Cambiarían las respuestas a las preguntas de arriba? ¡Este es *el mismo* conjunto de datos!

Ahora veamos cómo le va al perceptron con este dataset.

[22]:
```
t = np.concatenate([t1, t2])
x = np.vstack([phi1.T, phi2.T])

perce = Perceptron(max_iter=1000, tol=1e-10, random_state=2020202)

# Ajusto
perce.fit(x, t)
w = makew(perce)
```

```
plot_clasi(x, t, [w,], xp=[0.9, 1.1],  spines=None, equal=False)
```

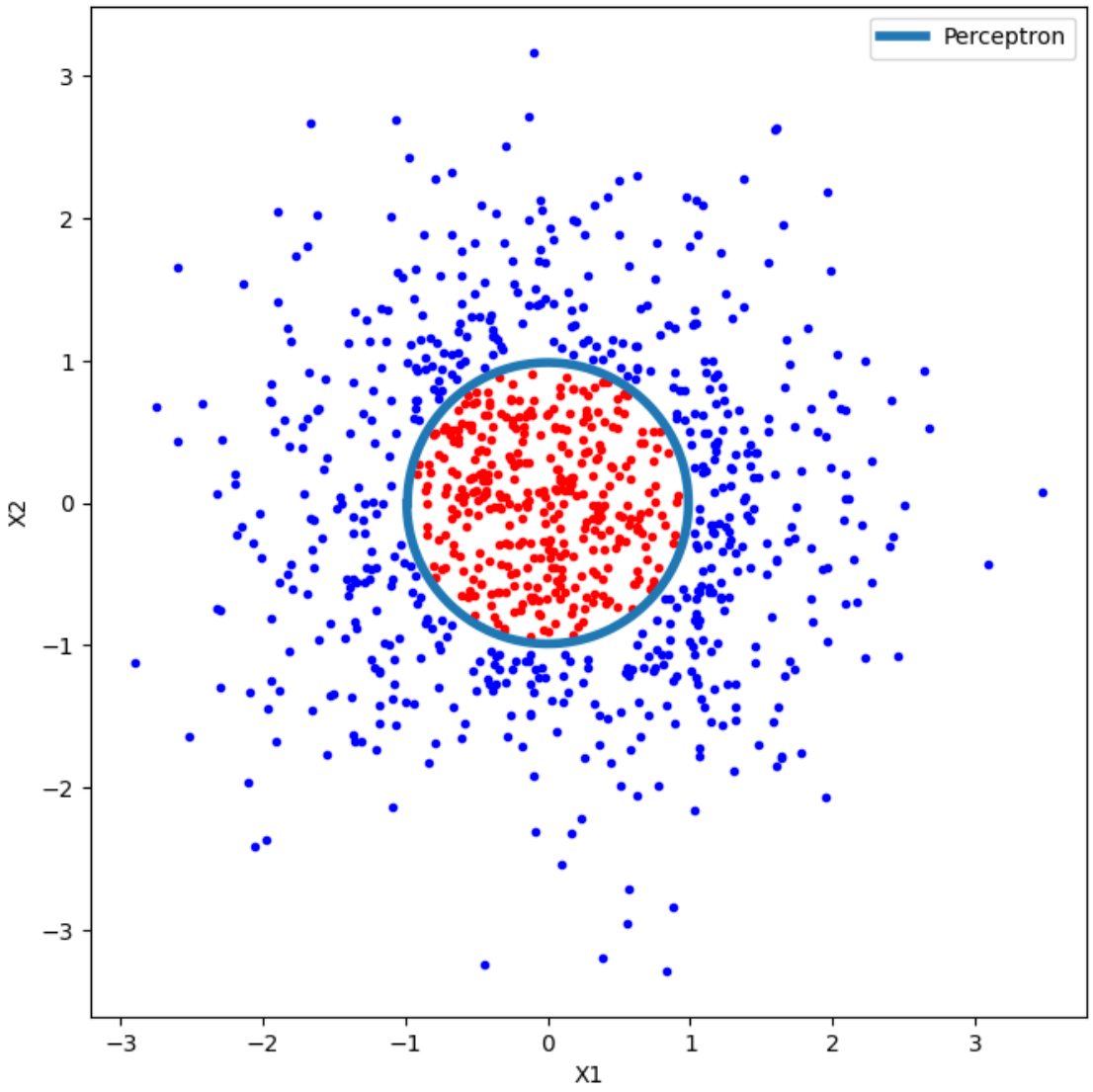


Y volvemos al espacio original

[23]:
```
fig = plt.figure(figsize=(8, 8))
plt.plot(*xc1, '.r')
plt.plot(*xc2, '.b')
plt.gca().set_aspect('equal')
plt.xlabel('X1')
plt.ylabel('X2')

# Points in the plane
yp = x[:, 1]
xp = (w[0] - w[2]*yp)/w[1]

# Convert back to original input space, keeping the sign
signx1 = np.where((yp > -np.pi/2) * (yp <= np.pi/2), 1, -1)
signx2 = np.where((yp > 0) * (yp <= np.pi), 1, -1)

x1p = signx1 * np.sqrt(xp**2/(1 + np.tan(yp)**2))
x2p = signx2 * np.sqrt(xp**2 - x1p**2)
```

```
# Plot
i = np.argsort(yp)
plt.plot(x1p[i], x2p[i], '-', lw=4, label='Perceptron')

plt.legend()
```

[23]: <matplotlib.legend.Legend at 0x7f2dfa102d40>



Conclusión. Si logramos encontrar el cambio de coordenadas adecuado, algunos problemas pueden *volverse* linealmente separables.

2 *MNIST

En esta sección, usamos el perceptrón para clasificar los datos de MNIST. Como vimos, este conjunto tiene diez clases, pero el perceptrón solo funciona para clasificaciones binarias.

Por lo tanto, vamos a transformarlo articifialmente para clasificar las imágenes de los dígitos en 5 y no-5.

2.1 Datos

Bajamos los datos usando directamente `sklearn`. Según la versión, esto cambia un poco:

[24]:
```
def sort_by_target(mnist):
    reorder_train = np.array(sorted([(target, i) for i, target in␣
 ↪enumerate(mnist.target.loc[:60000-1])]))[:, 1]
    reorder_test = np.array(sorted([(target, i) for i, target in␣
 ↪enumerate(mnist.target.loc[60000:])]))[:, 1]
    mnist.data[:60000] = mnist.data.loc[reorder_train]
    mnist.target[:60000] = mnist.target.loc[reorder_train]
    mnist.data[60000:] = mnist.data.loc[reorder_test + 60000]
    mnist.target[60000:] = mnist.target.loc[reorder_test + 60000]

try:
    from sklearn.datasets import fetch_openml
    mnist = fetch_openml('mnist_784', version=1, cache=True)
    mnist.target = mnist.target.astype(np.int8) # fetch_openml() returns␣
 ↪targets as strings
    sort_by_target(mnist) # fetch_openml() returns an unsorted dataset
except ImportError:
    from sklearn.datasets import fetch_mldata
    mnist = fetch_mldata('MNIST original')
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/datasets/_openml.py:968:
FutureWarning: The default value of `parser` will change from `'liac-arff'` to
`'auto'` in 1.4. You can set `parser='auto'` to silence this warning. Therefore,
an `ImportError` will be raised from 1.4 if the dataset is dense and pandas is
not installed. Note that the pandas parser may return different data types. See
the Notes Section in fetch_openml's API doc for details.
  warn(
<ipython-input-24-34ce319bd2c0>:4: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  mnist.data[:60000] = mnist.data.loc[reorder_train]
<ipython-input-24-34ce319bd2c0>:4: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame
```

```
See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  mnist.data[:60000] = mnist.data.loc[reorder_train]
<ipython-input-24-34ce319bd2c0>:6: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  mnist.data[60000:] = mnist.data.loc[reorder_test + 60000]
<ipython-input-24-34ce319bd2c0>:6: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  mnist.data[60000:] = mnist.data.loc[reorder_test + 60000]
```

Ahora redefinimos algunas variables por comodidad

[25]:
```
X, t = np.array(mnist["data"]), np.array(mnist["target"])
```

Para separar en *conjunto de entrenamiento* y de *test* usamos el hecho de que MNIST ya tiene una separación hecha que asegura el correcto equilibrio de cada clase en cada conjunto.

[26]:
```
X_train, X_test, t_train, t_test = X[:60000], X[60000:], t[:60000], t[60000:]
```

Y finalmente, barajamos los datos para asegurar que tenemos una buena distribución de las clases para hacer CV.

[27]:
```
shuffle_index = np.random.permutation(60000)
X_train, t_train = X_train[shuffle_index], t_train[shuffle_index]
```

Ahora cambiamos los targets de forma artificial para clasificar los datos en 5 y no-5. Solo necesitamos cambiar las etiquestas; los datos tenemos que dejarlos igual.

[28]:
```
t_train5 = t_train == 5
t_test5 = t_test == 5
```

Ejercicio

- Ahora instancien un `Perceptron` y ajusten con los datos que generamos.
- Calculen las métricas que vimos en el notebook.