

Aprendizaje supervisado: Clasificación

Machine Learning

SUPPORT VECTOR MACHINE



Flavio E. Spetale

2024

Bayes Ingenuo

El algoritmo de clasificación de Bayes Ingenuo se basa en el concepto de probabilidad condicional y busca maximizar la verosimilitud del modelo, i.e., otorgar mayor importancia a aquellos eventos que son realmente relevantes en el juego de datos.

Se supone que los predictores en un modelo Bayes Ingenuo son condicionalmente independientes o no están relacionados con ninguna de las otras características del modelo.

También supone que todas las características contribuyen por igual al resultado. Si bien estos supuestos a menudo se violan en escenarios del mundo real, simplifica un problema de clasificación al hacerlo más manejable computacionalmente.

Por tanto, sólo se requerirá una probabilidad única para cada variable, facilitando el cálculo del modelo. A pesar de esta suposición poco realista de independencia, el algoritmo de clasificación funciona bien, particularmente con tamaños de muestra pequeños.

Bayes Ingenuo

En general, los métodos estadísticos suelen estimar un conjunto de parámetros probabilísticos, que expresan la probabilidad condicionada de cada clase dadas las propiedades de un ejemplo (descrito en forma de atributos).














Un aspecto central en el algoritmo de Bayes Ingenuo es el concepto de probabilidad condicionada.

Probabilidad condicional

La probabilidad condicional $P(A|B)$ es la probabilidad de que ocurra un evento A, sabiendo que también sucede otro evento B

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

Bayes Ingenuo

	Nombre	Tipo	Ataque	Defensa	Velocidad	HP
	Growlithe	Fuego	Bajo	Débil	Baja	Baja
	Charizard	Fuego	Medio	Media	Alta	Media
	Arcanine	Fuego	Alto	Media	Alta	Alta
	Pyroar	Fuego	Bajo	Débil	Alta	Media
	Entei	Fuego	Alto	Dura	Alta	Alta
	Houndoom	Siniestro	Medio	Débil	Alta	Media
	Mightyena	Siniestro	Medio	Media	Baja	Media
	Hydreigon	Siniestro	Alto	Dura	Alta	Alta
	Thievul	Siniestro	Bajo	Débil	Alta	Baja
	Obstagoon	Siniestro	Medio	Dura	Alta	Alta
	Darkrai	Siniestro	Medio	Dura	Alta	Baja
	Bulbasaur	Planta	Bajo	Débil	Baja	Baja
	Torterra	Planta	Alto	Dura	Baja	Alta

Bayes Ingenuo

Probabilidades:

$$p(\text{tipo} = \text{fuego}) = \frac{5}{13} = 0.385$$

$$p(\text{tipo} = \text{siniestro}) = \frac{6}{13} = 0.462$$

$$p(\text{tipo} = \text{planta}) = \frac{2}{13} = 0.153$$

¿Cuántos tienen un ataque alto dado que sea del tipo siniestro?

Observamos que de entre las 6 clasificadas como siniestro, solo uno tiene un ataque alto.

$$p(\text{alto}|\text{siniestro}) = \frac{p(\text{alto} \cap \text{siniestro})}{p(\text{siniestro})} = \frac{1/13}{6/13} = \frac{1}{6} = 0.166$$

Algoritmo Bayes Ingenuo

El algoritmo Bayes Ingenuo clasifica nuevos ejemplos $d = (d_1, \dots, d_m)$ asignándole la clase c que maximiza la probabilidad condicional de la clase, dada la secuencia observada de atributos del ejemplo.

$$\arg \max_c P(c|d_1, \dots, d_m) = \arg \max_c \frac{P(d_1, \dots, d_m|c)P(c)}{P(d_1, \dots, d_m)} \approx \arg \max_c P(c) \prod_{i=1}^m P(d_i|c)$$

donde $P(c)$ y $P(d_i|c)$ se estiman a partir del conjunto de entrenamiento, utilizando las frecuencias relativas (estimación de la máxima verosimilitud, en inglés maximum likelihood estimation).

En el caso de tener un conjunto de datos que incluya *atributos numéricos continuos* es necesario algún tipo de preproceso antes de poder aplicar el método descrito.

Algoritmo Bayes Ingenuo

Existen dos alternativas principales para lidiar con los atributos continuos:

- La forma más simple consiste en categorizar los atributos continuos, de forma que sean convertidos en intervalos discretos y se puedan tratar de igual forma que los atributos nominales.
- Otra opción consiste en asumir que los valores de cada clase siguen una distribución gaussiana, y aplicar la siguiente fórmula:

$$p(d = v|c) = \frac{1}{\sqrt{2\pi\sigma_c^2}} e^{\left(-\frac{(v-\mu_c)^2}{2\sigma_c^2}\right)}$$

donde d corresponde al atributo, v a su valor, c a la clase, μ_c al promedio de valores de la clase c , y σ_c^2 a su desviación estándar.

En el caso que dentro del conjunto de test aparezca una pareja $\langle \text{atributo}, \text{valor} \rangle$ que no haya aparecido en el conjunto de entrenamiento, no dispondremos del valor $p(d_i|c)$.

Para solucionar este problema se suele aplicar alguna técnica de suavizado, como por ejemplo asignar como probabilidad condicional el valor $\frac{p(c)}{n}$, donde n indica el número de instancias de entrenamiento.

Ejemplo de Bayes Ingenuo

Probabilidades condicionales organizadas por *<atributo, valor>*.

Atributo - Valor	Fuego	Siniestro	Planta
Ataque - Bajo	2/5	4/6	1/2
Ataque - Medio	1/5	1/6	0
Ataque - Alto	2/5	1/6	1/2
Defensa - Débil	2/5	2/6	1/2
Defensa - Media	2/5	1/6	0
Defensa - Dura	1/5	3/6	1/2
Velocidad - Baja	1/5	1/6	0
Velocidad - Alta	4/5	5/6	1
HP - Baja	1/5	2/6	1/2
HP - Media	2/5	2/6	0
HP - Alta	2/5	2/6	1/2

Ejemplo de Bayes Ingenuo

Con esto habremos terminado el proceso de entrenamiento.

Ahora trataremos de clasificar una entrada nueva donde los atributos son Ataque: Alto, Defensa: Media, Velocidad: Alta y HP: Media. Para ello aplicamos el criterio de Bayes Ingenuo:

$$\arg \max_c P(c) \prod_{i=1}^m P(d_i|c)$$

	Tipo	Ataque $p(d_i c)$	Defensa $p(d_i c)$	Velocidad $p(d_i c)$	HP $p(d_i c)$	
$p(c)$?	Alto	Media	Alta	Media	
5/13	Fuego	2/5	2/5	4/5	2/5	$\frac{5}{13} * \left(\frac{2}{5} * \frac{2}{5} * \frac{4}{5} * \frac{2}{5} \right) = \frac{160}{8125} = 0.0196$
6/13	Siniestro	1/6	1/6	5/6	2/6	$\frac{6}{13} * \left(\frac{1}{6} * \frac{1}{6} * \frac{5}{6} * \frac{2}{6} \right) = \frac{60}{16848} = 0.0036$
2/13	Planta	1/2	0	1	0	$\frac{2}{13} * \left(\frac{1}{2} * \frac{0}{2} * \frac{2}{2} * \frac{0}{2} \right) = \frac{0}{208} = 0.0000$

Ejemplo de Bayes Ingenuo

Clasificaremos el nuevo dato con clase «fuego», dado que presenta la mayor verosimilitud de acuerdo con Bayes Ingenuo.



Infernape

Tipos de clasificadores de Bayes Ingenuo

- Gaussian Naïve Bayes (GaussianNB): Es una variante del clasificador Bayes Ingenuo, que se utiliza con distribuciones gaussianas, i.e., distribuciones normales y variables continuas. Este modelo se ajusta encontrando la media y la desviación estándar de cada clase.
- Multinomial Naïve Bayes (MultinomialNB): Supone que las características provienen de distribuciones multinomiales. Esta variante es útil cuando se utilizan datos discretos, como recuentos de frecuencia, y normalmente se aplica en casos de uso de procesamiento de lenguaje natural.
- Bernoulli Naïve Bayes (BernoulliNB): Es otra variante del clasificador Bayes Ingenuo, que se usa con variables booleanas, i.e., variables con dos valores (1 y 0).

Todo esto se puede implementar a través de la biblioteca Python de Scikit Learn.

Bayes Ingenuo

Ventajas:

1. Menos complejo: en comparación con otros clasificadores, Bayes Ingenuo se considera un clasificador más simple ya que los parámetros son más fáciles de estimar.
2. Se escala bien: en comparación con la regresión logística, Bayes Ingenuo se considera un clasificador rápido y eficiente que es bastante preciso cuando se cumple el supuesto de independencia condicional. También tiene bajos requisitos de almacenamiento.
3. Puede manejar datos de alta dimensión: la clasificación de documentos puede tener una gran cantidad de dimensiones, lo que puede resultar difícil de gestionar para otros clasificadores.

Desventajas:

1. Supuesto central poco realista: si bien el supuesto de independencia condicional en general funciona bien, no siempre se cumple, lo que lleva a clasificaciones incorrectas.
2. Sujeto a frecuencia cero: La frecuencia cero ocurre cuando una variable categórica no existe dentro del conjunto de entrenamiento. La probabilidad en este caso sería cero, y dado que este clasificador multiplica todas las probabilidades condicionales juntas, esto también significa que la probabilidad posterior será cero. Para evitar este problema, se puede aprovechar el suavizado de Laplace.

Ejemplo en python

Conjunto de datos: *Clasificación de frutas*

Información del conjunto de datos:

El conjunto de datos esta formado por xxx frutas, 70 elementos cada una.
Todos estos parámetros fueron de valor real continuo.

Información de atributos:

Para construir los datos, se midieron 34 parámetros fenotípicos y geométricos de las frutas.

Ejemplo en python

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.naive_bayes import GaussianNB
from sklearn import metrics

dataFruit = pd.read_csv("FruitBD.csv")
xFruit = dataFruit.iloc[:, [2,8,15,23]].values
yFruit = dataFruit['Clase']

xFruit = StandardScaler().fit_transform(xFruit)
xFruitTrain, xFruitTest, yFruitTrain, yFruitReal = train_test_split(xFruit, yFruit, test_size=0.2)
```

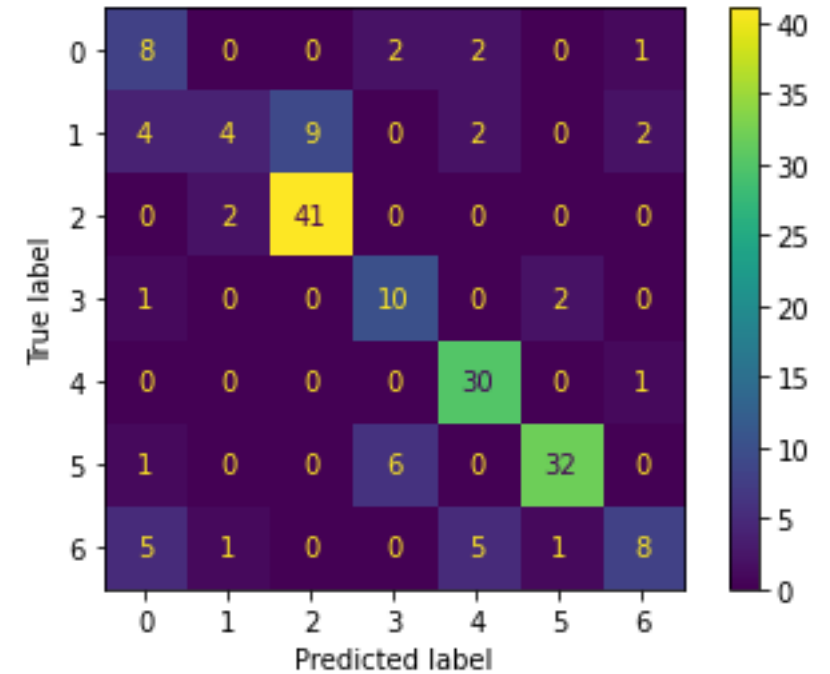
Ejemplo en python

""" Naive Bayes - Classification """

```
modelNB = GaussianNB()  
modelNB.fit(xFruitTrain, yFruitTrain)
```

```
yFruitPred = modelNB.predict(xFruitTest)
```

```
cm = metrics.confusion_matrix(yFruitReal, yFruitPred)  
metrics.ConfusionMatrixDisplay(confusion_matrix=cm).plot()
```



metrics.precision_score(yFruitReal, yFruitPred, average='macro') → 0.674

metrics.recall_score(yFruitReal, yFruitPred, average='macro') → 0.673

metrics.accuracy_score(yFruitReal, yFruitPred) → 0.738

Ejemplo en python

""" Naive Bayes - All features """

```
xFruit = dataFruit.drop('Clase', axis=1)
xFruit = StandardScaler().fit_transform(xFruit)
xFruitTrain, xFruitTest, yFruitTrain, yFruitReal = train_test_split(xFruit, yFruit,
test_size=0.2)
```

```
modelNB = GaussianNB()
modelNB.fit(xFruitTrain, yFruitTrain)
```

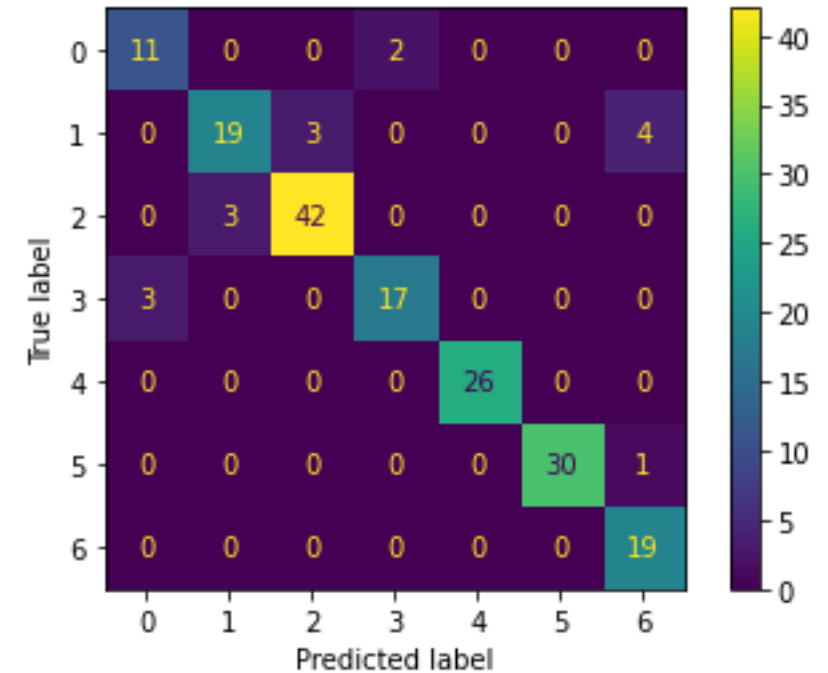
```
yFruitPred = modelNB.predict(xFruitTest)
```

```
cm = metrics.confusion_matrix(yFruitReal, yFruitPred)
metrics.ConfusionMatrixDisplay(confusion_matrix=cm).plot()
```

metrics.precision_score(yFruitReal, yFruitPred, average='macro') → 0.895

metrics.recall_score(yFruitReal, yFruitPred, average='macro') → 0.903

metrics.accuracy_score(yFruitReal, yFruitPred) → 0.911



k vecinos más cercanos (k-NN)

El algoritmo k-NN, es un clasificador de aprendizaje supervisado no paramétrico (no hace suposiciones sobre los datos subyacentes), ***no genera un modelo*** fruto del aprendizaje con datos de entrenamiento, sino que el aprendizaje sucede en el mismo momento en el que se pide clasificar una nueva instancia.

A este tipo de algoritmos se los denomina métodos de aprendizaje perezoso (lazy learning methods).

Todo el cálculo ocurre cuando se realiza una clasificación o predicción. Dado que depende en gran medida de la memoria para almacenar todos sus datos de entrenamiento, también se lo denomina método de aprendizaje basado en instancias o basado en la memoria.

Si bien no es tan popular hoy en día, sigue siendo uno de los primeros algoritmos que uno aprende en la ciencia de datos debido a su simplicidad y precisión.

A medida que crece un conjunto de datos, k-NN se vuelve cada vez más ineficiente, lo que compromete el rendimiento general del modelo.

El funcionamiento del algoritmo es muy simple. Para cada nueva instancia a clasificar, se calcula la distancia con todas las instancias de entrenamiento, se seleccionan las k instancias más cercanas y su clase se determinará como la clase mayoritaria de sus k instancias más cercanas.

k-NN: métricas de distancia

La métrica de similitud utilizada debería tener en cuenta la importancia relativa de cada atributo, puesto que esta influirá fuertemente en la relaciones de cercanía que se irán estableciendo en el proceso de construcción del algoritmo.

La métrica de distancia puede llegar a contener pesos que nos ayudarán a calibrar el algoritmo de clasificación, convirtiéndola de hecho en una métrica personalizada.

Debería ser eficiente computacionalmente, ya que deberemos ejecutar el cálculo de similitud muchas veces durante el proceso de clasificación de nuevas instancias.

Distancia euclidiana (p=2): Esta es la medida de distancia más utilizada y está limitada a vectores de valor real.

$$d(x, y) = \sqrt{\sum_{i=1}^n (y_i - x_i)^2}$$

k-NN: métricas de distancia

Distancia Manhattan (p=1): Esta es también otra métrica de distancia popular, que mide el valor absoluto entre dos puntos.

$$d(x, y) = \sum_{i=1}^n |y_i - x_i|$$

Distancia Minkowski: Esta distancia puede considerarse una generalización de las distancias euclideas y Manhattan

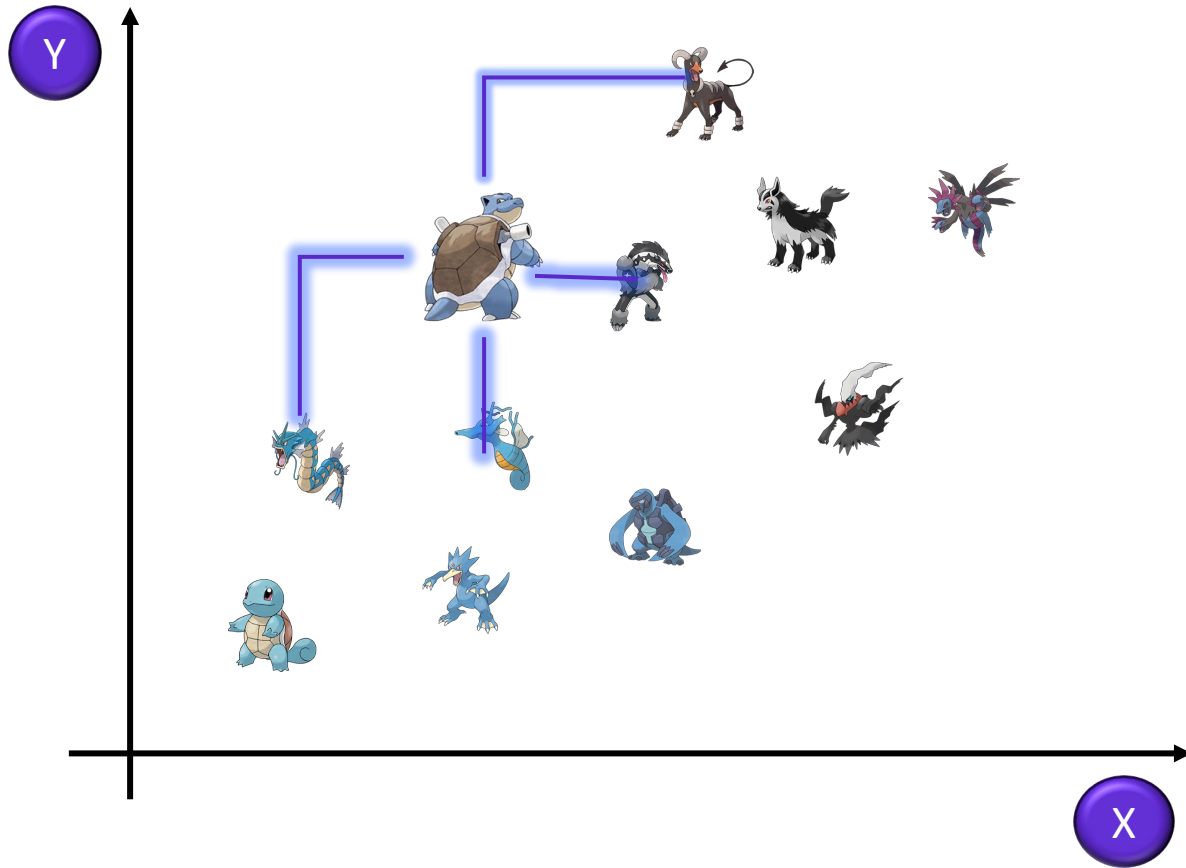
$$d(x, y) = \left(\sum_{i=1}^n |y_i - x_i|^p \right)^{1/p}$$

Distancia de Hamming: Esta técnica se usa típicamente con vectores booleanos o de cadena, identificando los puntos donde los vectores no coinciden.

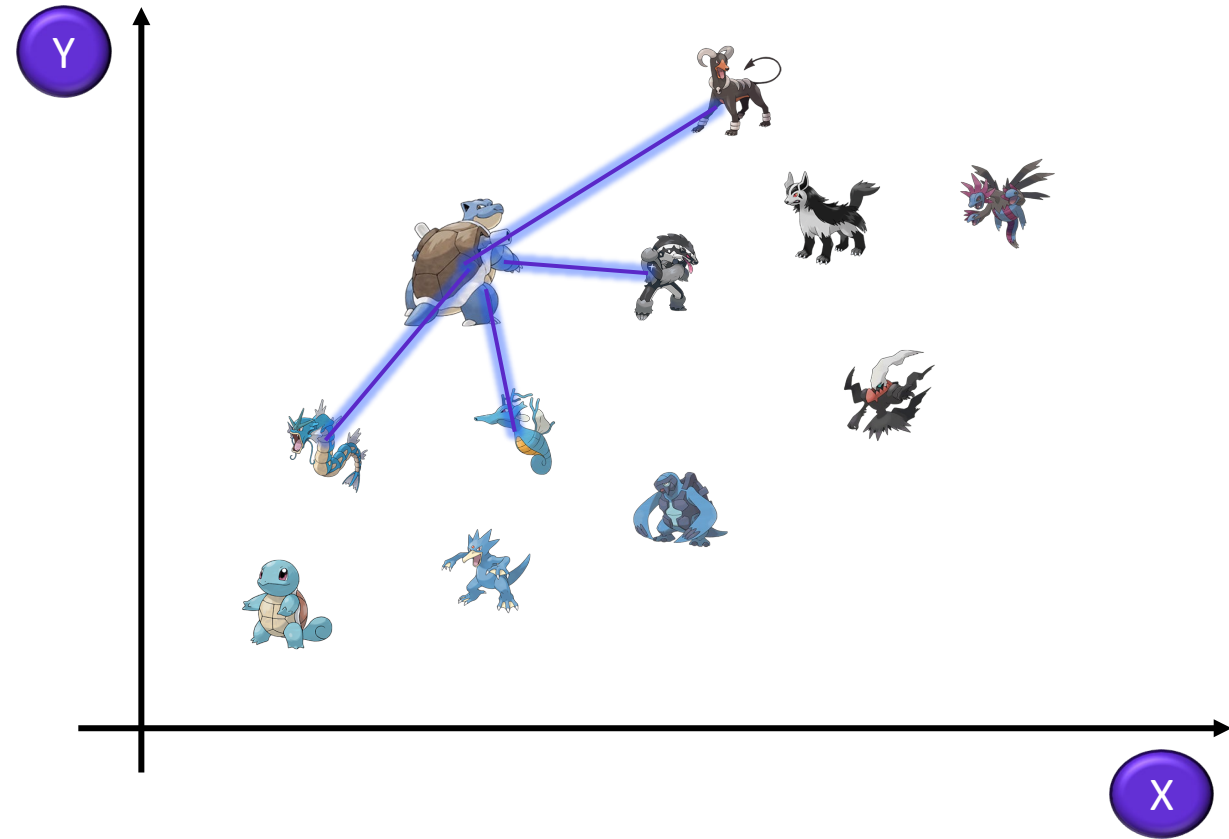
$$d(x, y) = \sum_{i=1}^n |y_i - x_i|$$

$$\text{si } x_i = y_i \rightarrow 0 \text{ sino } x_i \neq y_i \rightarrow 1$$

Métricas de distancia



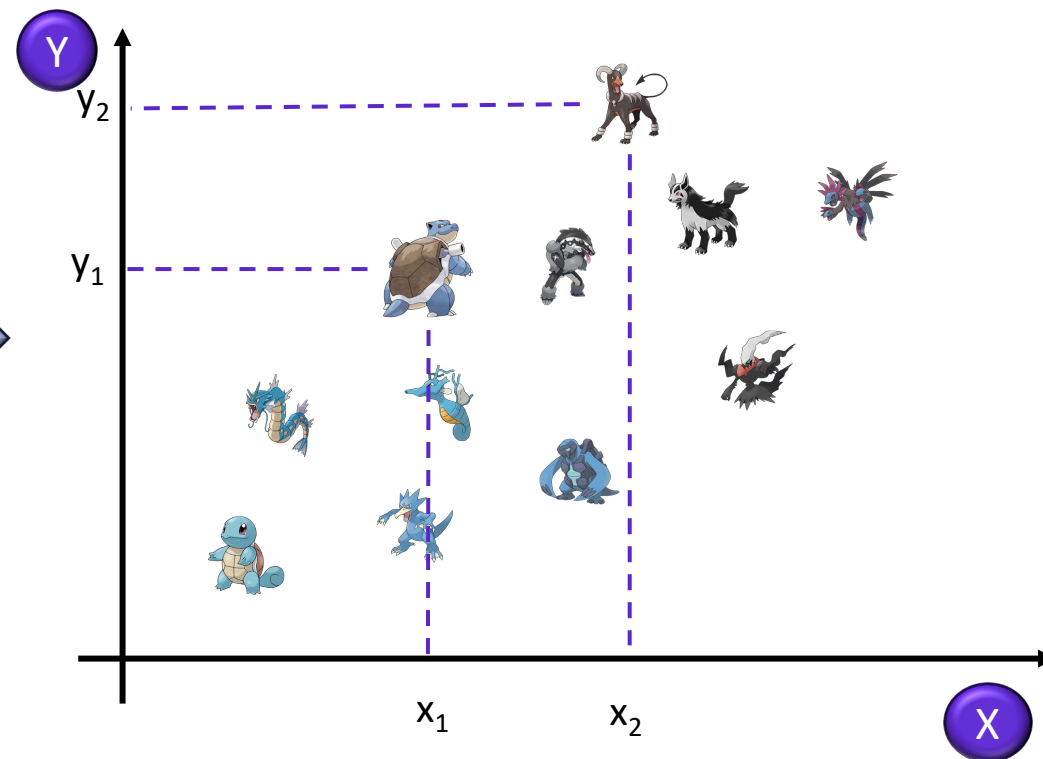
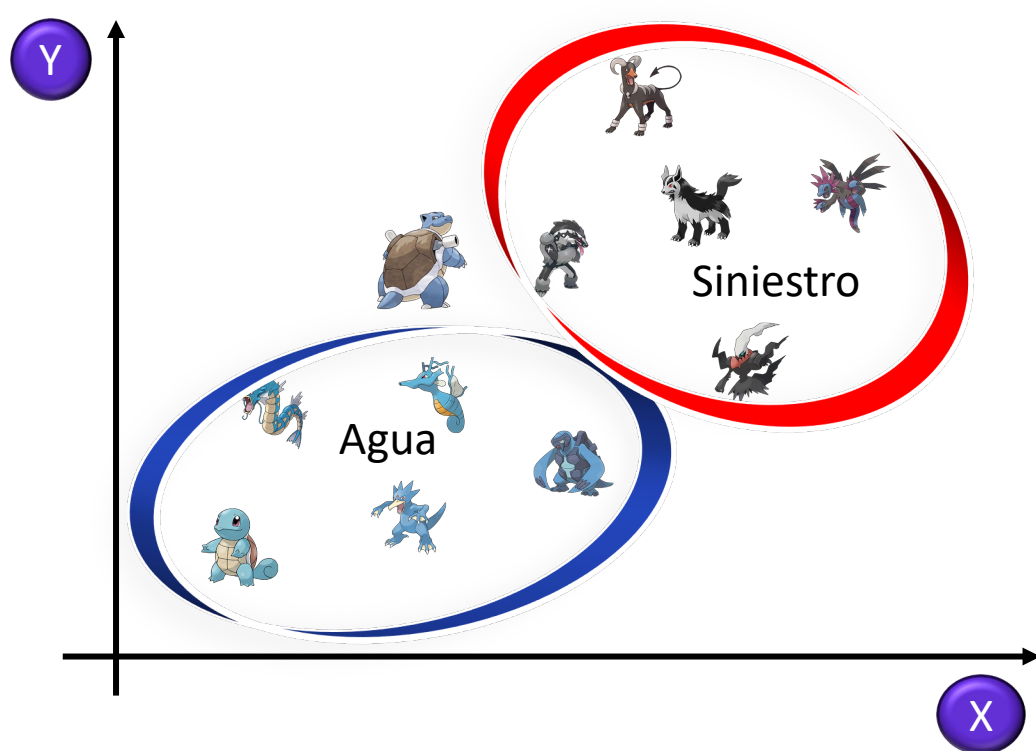
Distancia Manhattan



Distancia euclidiana

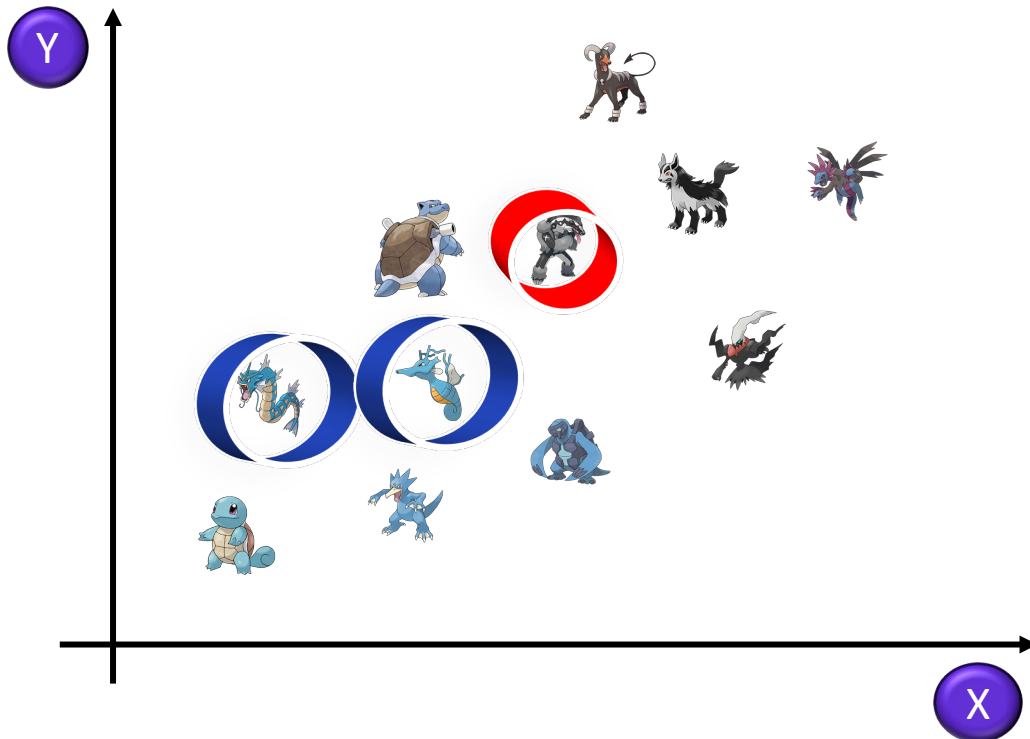
Algoritmo k-NN

1. Calcular la distancia de este nuevo punto hacia todos los otros puntos que ya se encuentran etiquetados con su clase correspondiente para después ordenar de menor a mayor.
2. Con las distancias ordenadas de menor a mayor basta con tomar los mas cercanos al punto a clasificar, lo cual esta determinado con el valor K seleccionado. Aquí consideramos $k=3$.



Algoritmo k-NN

3. Utilizamos un método de agregación, voto mayoritaria, para estimar a la categoría que pertenece el nuevo dato.



Clasificamos el nuevo dato
con clase «Agua»



Blastoise

k-NN: Selección del valor k

Optimizar el valor de k es de suma importancia, ya que al elegir un valor de k muy pequeño es posible que el ruido de los datos tome importancia, por otro lado si el valor de k es alto la clase que sea mayoría tendrá un peso importante.

Este valor suele fijarse tras un proceso de pruebas con varias instancias.

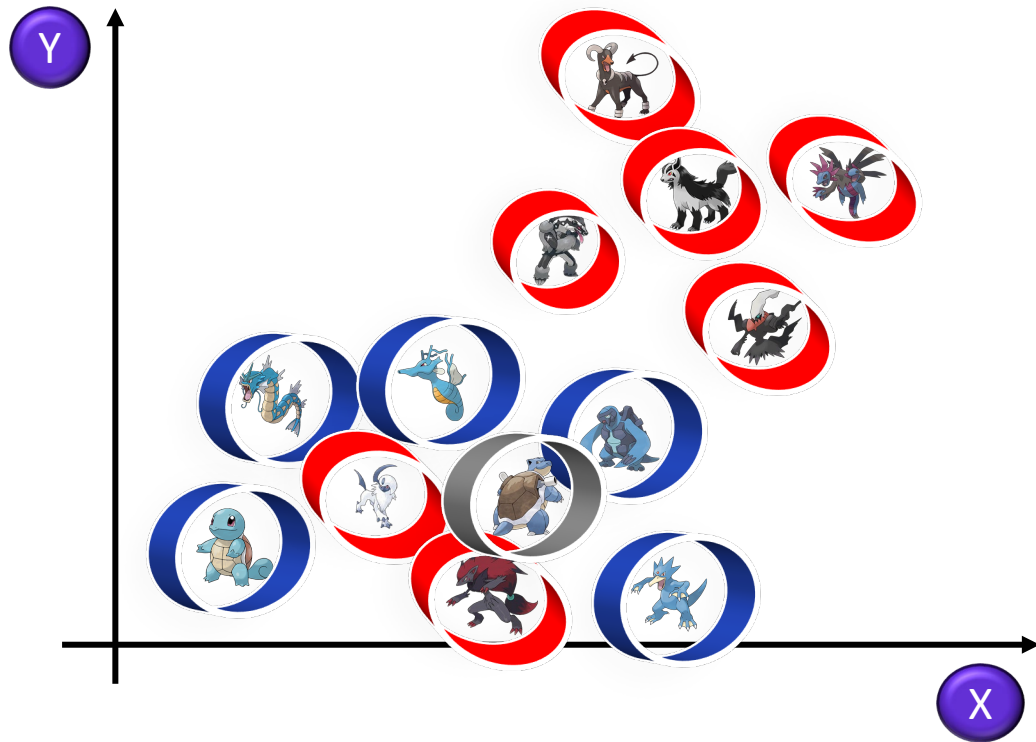
Se suele escoger un número impar o primo para minimizar la posibilidad de empates en el momento de decidir la clase de una nueva instancia.

Aun así, cuando se produce un empate se debe decidir cómo clasificar la instancia. Algunas alternativas pueden ser, por ejemplo, no dar predicción o dar la clase más frecuente en el conjunto de aprendizaje de las clases que han generado el empate.

Otro factor importante que se debe considerar antes de aplicar el algoritmo k-NN es el rango u orden de los datos. Es importante expresar los distintos atributos en valores que sean «comparables», i.e., normalizarlos.

k-NN: Selección del valor k

k pequeño (*overfitting*)



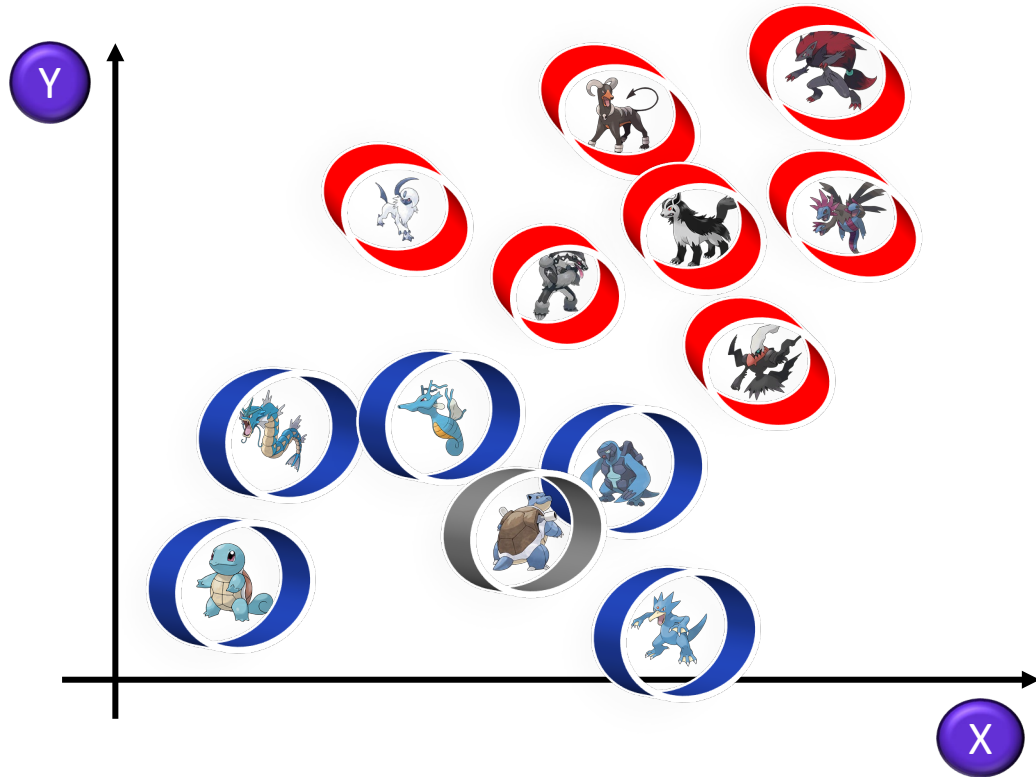
Clasificamos el nuevo dato
con clase «Siniestro»



Tomando un valor de $k = 3$ el nuevo dato será etiquetado como Siniestro aunque visualmente es evidente que pertenece a Agua, sin embargo aquí es donde el **ruido toma importancia**, y se dice que el modelo está sobreajustado.

k-NN: Selección del valor k

k alto (underfitting)



Clasificamos el nuevo dato
con clase «Siniestro»



Tomando un valor de $k = 12$ el nuevo dato será etiquetado como Siniestro, subajuste del algoritmo, ya que visualmente el nuevo dato pertenece a la categoría Agua, sin embargo al ser la clase Siniestro mayoría, es aquí donde será clasificado.

k-NN: Selección del valor k

Al analizar los extremos (k muy pequeño y k muy alto) es clara la importancia de un k-valor adecuado, normalmente se escoge como un *buen valor de inicio la raíz cuadrada del número de datos para el entrenamiento*, sin embargo es buena practica tomar varios valores de k, y observar cual de estos es el que nos da una mejor clasificación de los datos de entrenamiento.

Ejemplo en python

```
KNeighborsClassifier(  
    n_neighbors=5,      # Número de vecinos a considerar  
    weights='uniform',  # Cómo ponderar distancias  
    algorithm='auto',   # Algoritmo para calcular los vecinos  
    leaf_size=30,       # El tamaño de la hoja para agilizar las búsquedas  
    p=2,                # El parámetro de potencia para la métrica de Minkowski  
    metric='minkowski', # El tipo de distancia a utilizar  
    n_jobs=None         # Trabajos en paralelos a ejecutar  
)
```

Ejemplo en python

Codificación one-hot de variables categóricas en Sklearn

```
from sklearn.preprocessing import OneHotEncoder  
from sklearn.compose import make_column_transformer
```

```
X = df.drop(columns = ['clases'])  
y = df ['clases']
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state = 100)
```

```
column_transformer = make_column_transformer( (OneHotEncoder(), ['x1', 'x2', 'x3']),  
remainder='passthrough')
```

```
X_train = column_transformer.fit_transform(X_train)  
X_train = pd.DataFrame(data=X_train, columns=column_transformer.get_feature_names())
```

Ejemplo en python

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn import metrics

dataFruit = pd.read_csv("FruitBD.csv")

yFruit = dataFruit['Clase']

xFruit = dataFruit.drop('Clase', axis=1)
xFruit = StandardScaler().fit_transform(xFruit)
xFruitTrain, xFruitTest, yFruitTrain, yFruitReal = train_test_split(xFruit, yFruit, test_size=0.2)

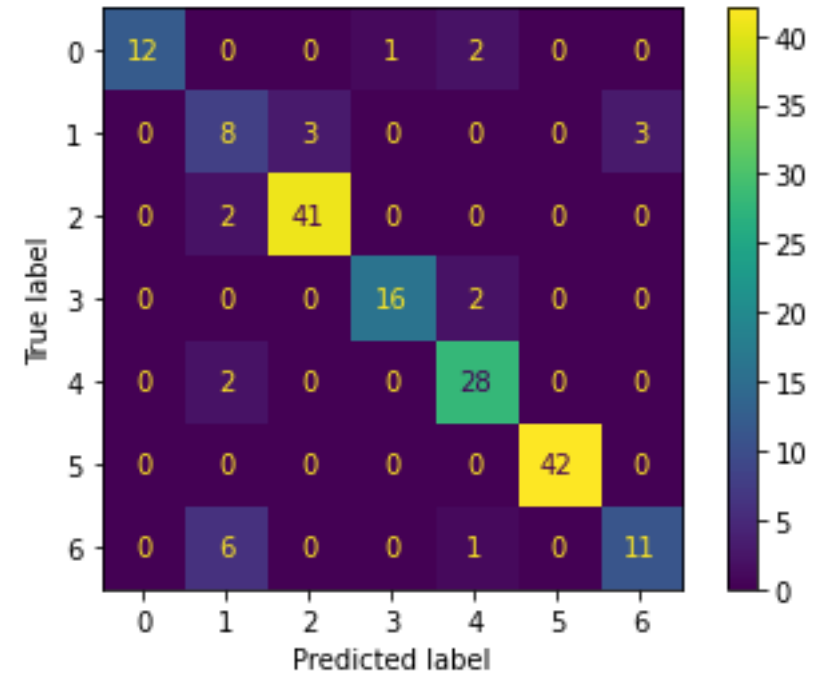
modelkNN = KNeighborsClassifier(n_neighbors=3, metric='minkowski', p=2)

modelkNN.fit(xFruitTrain, yFruitTrain)

yFruitPred = modelkNN.predict(xFruitTest)
```

Ejemplo en python

```
cm = metrics.confusion_matrix(yFruitReal, yFruitPred)
metrics.ConfusionMatrixDisplay(confusion_matrix=cm).plot()
```



`metrics.precision_score(yFruitReal, yFruitPred, average='macro')` → 0.850

`metrics.recall_score(yFruitReal, yFruitPred, average='macro')` → 0.822

`metrics.accuracy_score(yFruitReal, yFruitPred)` → 0.877

Ejemplo en python

```
from sklearn.model_selection import GridSearchCV
```

```
params = {  
    'n_neighbors': range(1, 15, 2),  
    'p': [1,2],  
    'weights': ['uniform', 'distance']  
}
```

```
clf = GridSearchCV( estimator=KNeighborsClassifier(), param_grid=params, cv=5, n_jobs=5, verbose=1)
```

```
clf.fit(xFruitTrain, yFruitTrain)
```

```
print(clf.best_params_)      →      {'n_neighbors': 5, 'p': 1, 'weights': 'distance'}
```

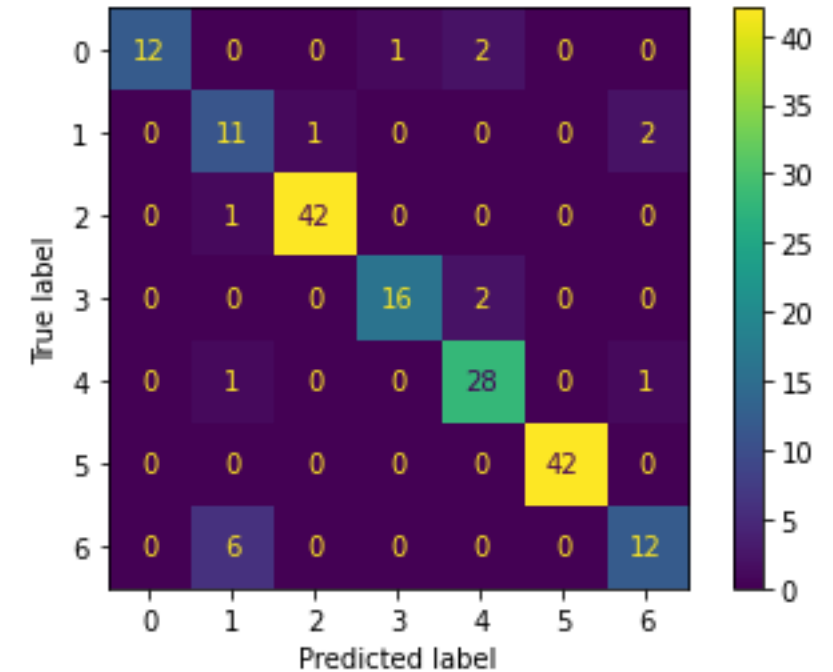
```
modelkNN = KNeighborsClassifier(n_neighbors=5, metric='minkowski', p=1, weights='distance')
```

```
modelkNN.fit(xFruitTrain, yFruitTrain)
```

```
yFruitPred = modelkNN.predict(xFruitTest)
```

Ejemplo en python

```
cm = metrics.confusion_matrix(yFruitReal, yFruitPred)
metrics.ConfusionMatrixDisplay(confusion_matrix=cm).plot()
```



`metrics.precision_score(yFruitReal, yFruitPred, average='macro')` → 0.881

`metrics.recall_score(yFruitReal, yFruitPred, average='macro')` → 0.864

`metrics.accuracy_score(yFruitReal, yFruitPred)` → 0.905

Máquinas de soporte vectorial (SVM)

La gran aportación de Vapnik radica en que construye un método que tiene por objetivo producir predicciones en las que se puede tener mucha confianza, en lugar de lo que se ha hecho tradicionalmente, que consiste en construir hipótesis que cometan pocos errores.

La hipótesis tradicional se basa en lo que se conoce como minimización del riesgo empírico (empirical risk minimization) mientras que el enfoque de las SVM se basa en la minimización del riesgo estructural (structural risk minimization), de modo que lo que se busca es construir modelos que estructuralmente tengan poco riesgo de cometer errores ante clasificaciones futuras.

El concepto de minimización del riesgo estructural fue introducido en 1974 por Vapnik y Chervonenkis.

La idea detrás de las máquinas de soporte vectorial es muy intuitiva. Si la frontera definida entre dos regiones con elementos de clases diferentes es compleja, en lugar de construir un clasificador complejo que reproduzca dicha frontera, lo que se intenta es «doblar» el espacio de datos en un espacio de mayor dimensionalidad de forma que con un único corte se puedan separar fácilmente ambas regiones.

SVM: Definición de margen e hiperplano separador

Uno de los factores diferenciadores de este algoritmo respecto de otros clasificadores es su gestión del concepto de margen.

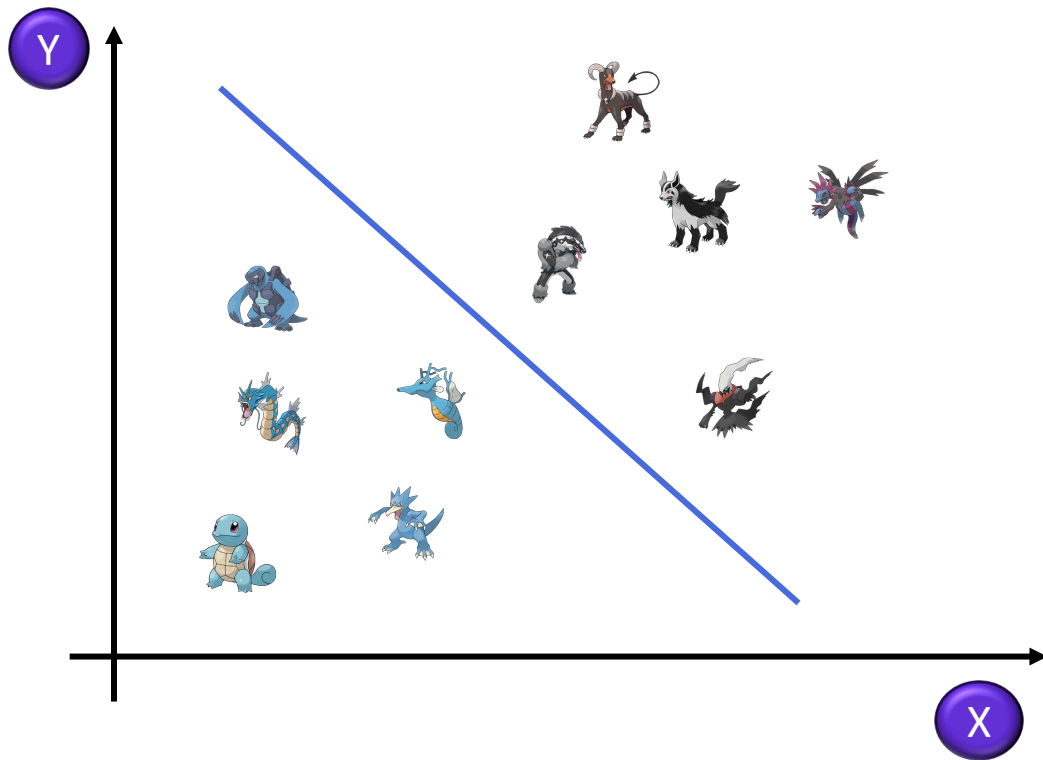
Las SVM buscan maximizar el margen entre los puntos pertenecientes a los distintos grupos a clasificar.

Maximizar el margen implica que el hiperplano de decisión o separación esté diseñado de tal forma que el máximo número de futuros puntos queden bien clasificados.

Las SVM utilizan las técnicas de optimización cuadrática propuestas por el matemático italiano Lagrange.

El objetivo de las SVM es encontrar el hiperplano óptimo que maximiza el margen entre clases (variable objetivo) del juego de datos de entrenamiento.

SVM: Definición de margen e hiperplano separador

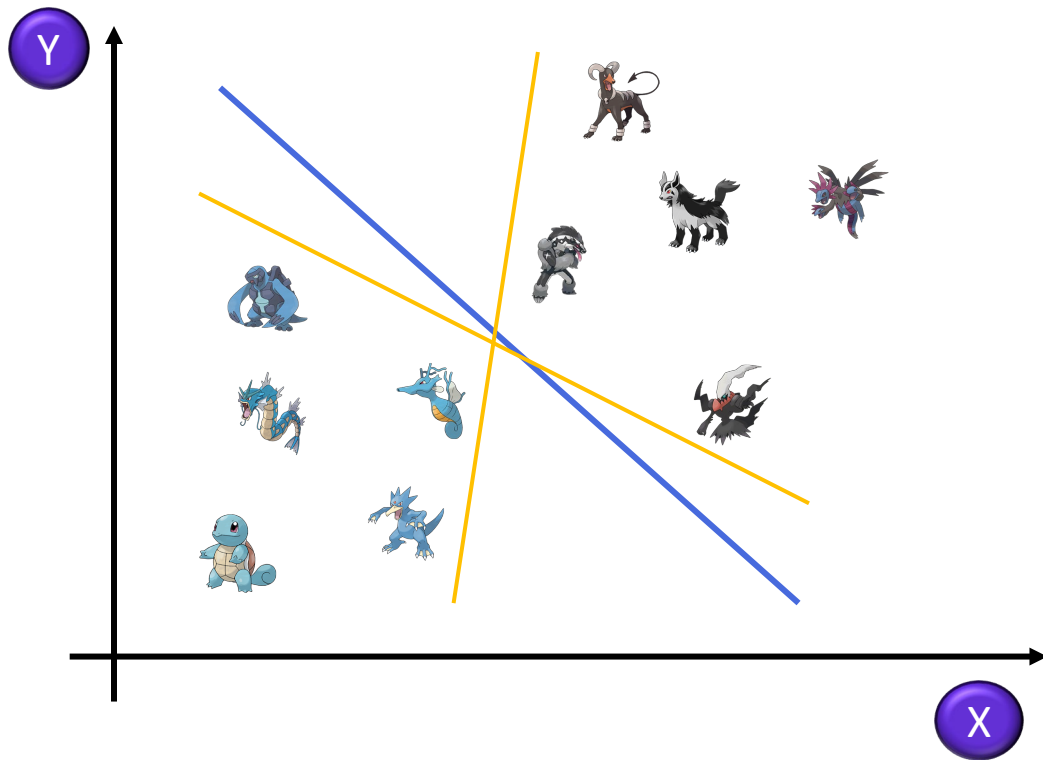


Aquí podemos observar que a partir de un hiperplano podríamos separar las dos clases de datos de forma clara.

Diremos que esta recta de separación es en realidad un hiperplano de separación.

Debemos tener en cuenta que el hecho de disponer de un hiperplano separador no nos garantiza, en absoluto, que este sea el mejor de todos los posibles hiperplanos separadores.

SVM: Definición de margen e hiperplano separador

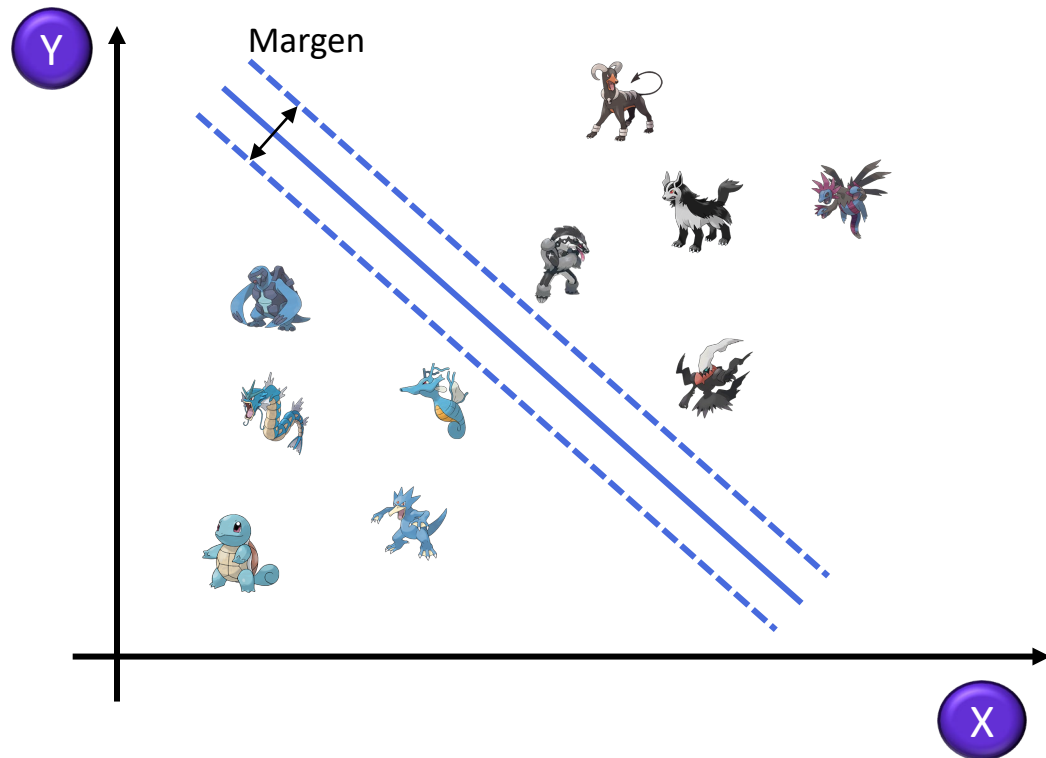


Aquí podemos ver que existen infinitos hiperplanos separadores, todos ellos perfectamente válidos para separar las dos clases propuestas.

Ante la pregunta de si hay o no un hiperplano separador mejor que otro, pensemos, por ejemplo, que aquellos hiperplanos que estén demasiado cerca de los puntos del gráfico, muy probablemente no serán demasiado buenos para clasificar nuevos puntos, ya que correremos mucho riesgo de acabar teniendo puntos en el lado no deseado del hiperplano.

Por lo tanto, nuestra intuición nos dice que sería conveniente fijar un margen de seguridad capaz de establecer una distancia entre los puntos de cada clase de puntos a clasificar.

SVM: Definición de margen e hiperplano separador



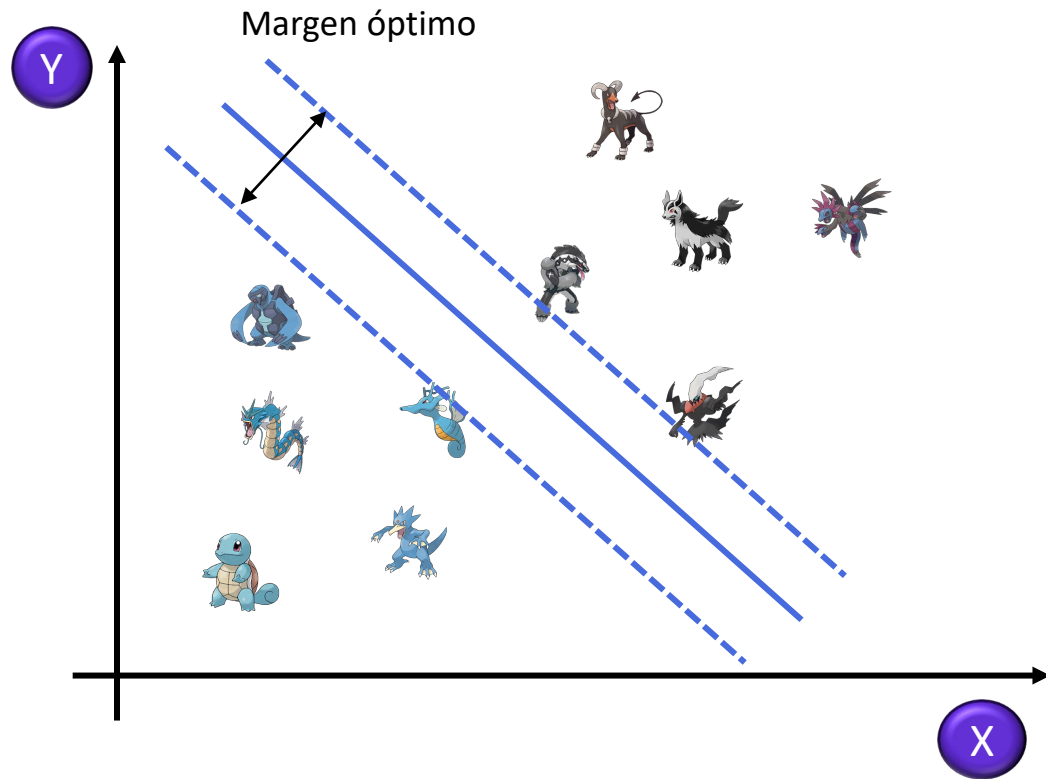
El margen es un área donde no deberíamos encontrar ningún punto del conjunto de datos de entrenamiento.

Para construir el margen asociado a un hiperplano, procederemos a calcular la distancia entre el hiperplano y el punto más cercano a este.

Una vez que tenemos esta distancia, la doblamos y este será nuestro margen. De este modo, cuanto más cerca esté un hiperplano de los puntos, menor será su margen.

Por el contrario, cuanto más alejado este un hiperplano de los puntos, mayor será su margen y, en consecuencia, menor será su riesgo de clasificar incorrectamente nuevos puntos de datos.

SVM: Definición de margen e hiperplano separador



Así, el hiperplano separador óptimo se define como el hiperplano que tenga mayor margen posible.

El objetivo de las SVM es encontrar el hiperplano separador óptimo que maximice el margen de los datos de entrenamiento.

SVM: Cálculo del margen y del hiperplano separador óptimo

La expresión más habitual para representar una recta en el plano es:

$$y = ax + b$$

Si generalizamos este caso a un espacio de más dimensiones, por cuestiones de practicidad se suele usar la expresión:

$$w^T \cdot x = 0$$

donde la equivalencia con la expresión anterior se muestra a continuación:

$$w = \begin{pmatrix} b \\ a \\ -1 \end{pmatrix}, \quad x = \begin{pmatrix} 1 \\ x \\ y \end{pmatrix}$$

A partir de la notación, es fácil operar y ver que estas dos expresiones son equivalentes:

$$w^T \cdot x = b + a \cdot x - y$$

SVM: Cálculo del margen y del hiperplano separador óptimo

Aunque ambas ecuaciones son equivalentes, generalmente es más cómodo trabajar con la expresión $w^T \cdot x$ por los siguientes motivos:

- Es más adecuada para más de dos dimensiones.
- El vector w siempre será perpendicular al hiperplano, por definición.

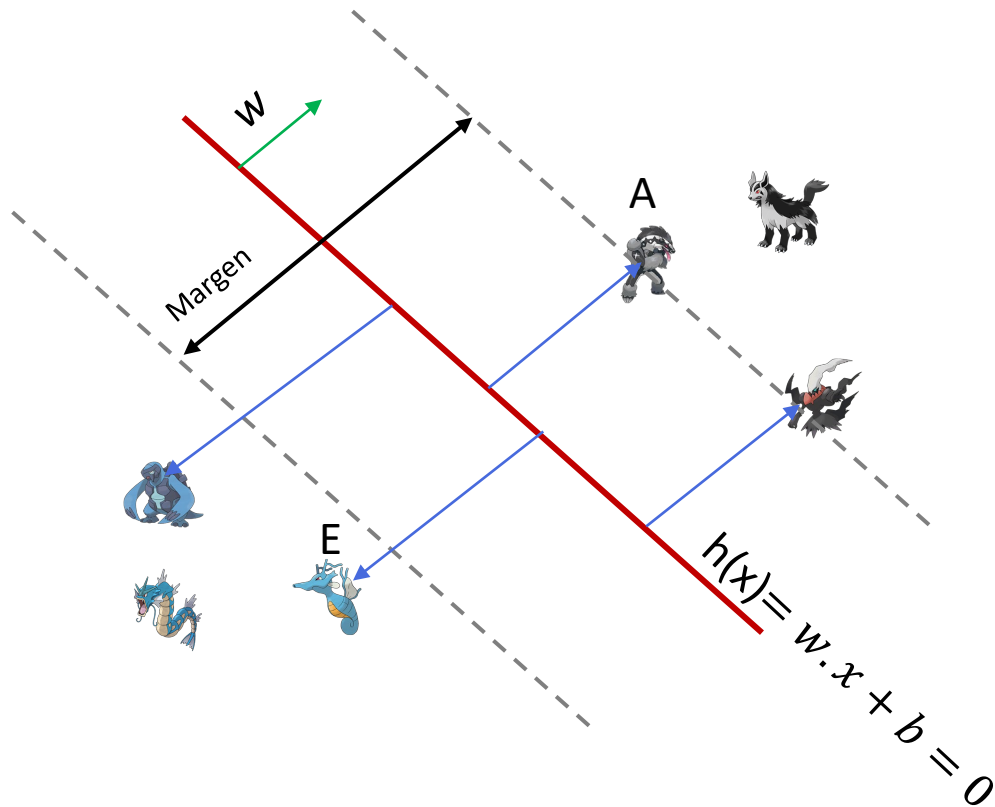
Un cálculo que se repite constantemente en las SVM es la distancia entre un punto x_0 y un hiperplano definido por su vector normal w .

Este cálculo se efectúa mediante una proyección del vector definido entre el punto x_0 y el hiperplano en cuestión sobre el vector normal al hiperplano.

$$d = \frac{|x_0 \cdot w|}{\|w\|}$$

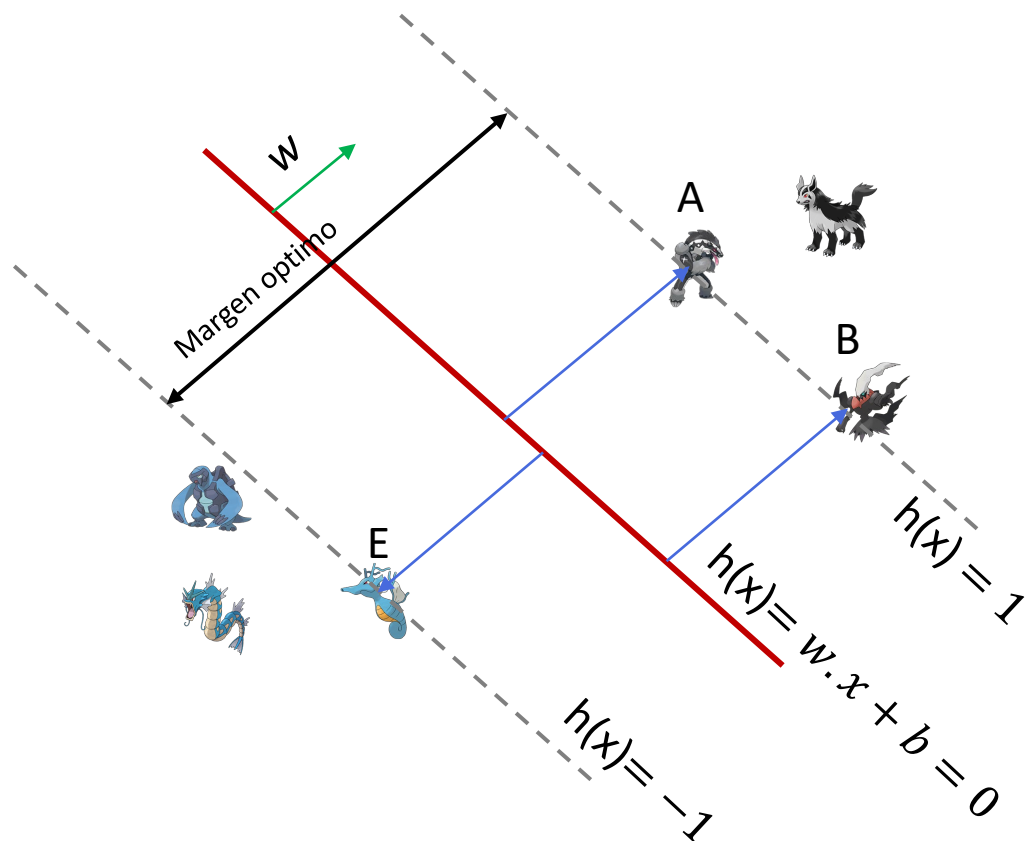
Si x_0 es el punto más cercano al hiperplano, podemos definir el margen como el doble de la distancia calculada (extendiendo dicha distancia a cada lado del hiperplano), dado que en ese margen no existe ningún otro punto.

SVM: Cálculo del margen y del hiperplano separador óptimo



Podemos apreciar cómo el margen calculado para el punto A no es el óptimo, ya que podríamos conseguir un margen mayor hasta alcanzar el punto E, haciendo retroceder el hiperplano en cuestión.

SVM: Cálculo del margen y del hiperplano separador óptimo



Vemos cómo el margen alcanza los puntos A, B y E que marcan la frontera entre los dos grupos de puntos.

Diremos entonces que el margen e hiperplano generados son los óptimos.

Los conceptos hiperplano y margen están intrínsecamente relacionados, puesto que fijado un margen podemos obtener el hiperplano que pasa justo por su centro y al revés, fijado un hiperplano podemos obtener el margen a partir de la distancia de los puntos frontera con el hiperplano.

Por lo tanto, podemos concluir que determinar el mayor margen posible es equivalente a encontrar el hiperplano óptimo.

SVM: Cálculo del margen y del hiperplano separador óptimo

El proceso de optimización se completa con el método de los multiplicadores de Lagrange, adecuado para encontrar máximos y mínimos de funciones continuas de múltiples variables y sujetas a restricciones.

En dos dimensiones, los multiplicadores de Lagrange permiten resolver el problema de maximizar una función $f(x, y)$ sujeta a las restricciones $g(x, y) = 0$, con la única condición de que ambas tengan derivadas parciales continuas, es decir, que su forma sea «suave» en todas sus variables.

De este modo, Lagrange optimizará la función $f(x, y) - \lambda \cdot g(x, y)$, donde λ es el coste de la restricción.

Esto es generalizable a cualquier número de dimensiones.

En el caso de las SVM, el objetivo es encontrar aquel hiperplano que maximiza el margen (la distancia del punto más cercano a dicho hiperplano) y que satisface que todos los puntos de una clase estén a un lado del hiperplano y todos los de la otra clase estén al otro lado. Cuando esto es posible se habla de *hard-margin*.

Si esto no es posible (el conjunto de datos no es linealmente separable), se habla de *soft-margin* y es necesario añadir una condición más de forma que el hiperplano óptimo minimice el número de elementos de cada clase que se encuentran en el lado equivocado, teniendo en cuenta también su distancia.

Algoritmo SVM

Partimos de una clasificación binaria, i.e., donde cada punto de nuestro espacio de características, $x_i \in X$, estará asociado a una clase binaria $\{-1, 1\}$ para entender el concepto.

En este caso, nuestra función de clasificación podría consistir en calcular la suma de distancias alteradas por un peso w .

$$h(z) = \text{signo} \left(\sum_{i=1}^n w_i \cdot c_i \cdot k(x_i, z) \right)$$

- n es el número de entradas del conjunto de datos de entrenamiento.
- z es el nuevo punto a clasificar.
- $h(z) \in \{-1, 1\}$ es la función de clasificación.
- $k: X \cdot X \rightarrow R$ es la función kernel que mide la similitud entre puntos (producto escalar, distancia, etc.).
- El conjunto de pares $\{(x_i, c_i)\}_{i=1}^n$ son los puntos etiquetados del juego de datos, i.e., constituyen el conjunto de datos de entrenamiento, donde $c_i \in \{-1, 1\}$ es la etiqueta o clase del punto x_i .
- $w_i \in R$ son los pesos que el algoritmo ha determinado para el conjunto de datos de entrenamiento.
- $\text{signo}()$ es la función que nos devuelve el signo de un número $\{+,-\}$.

SVM: Funciones kernel

El trabajo del algoritmo SVM será, precisamente, determinar los valores w y b óptimos en términos de maximización del margen, con el objetivo de encontrar el mejor hiperplano de separación.

Las SVM se ocupan tan solo de resolver problemas de clasificación lineal y se apoyan en las funciones kernel para transformar un problema no lineal en el espacio original X en un problema lineal en el espacio transformado F .

Las funciones kernel son las que «doblan» el espacio de entrada para poder realizar cortes complejos con un simple hiperplano.

Una función kernel es aquella que a cada par de vectores de un espacio origen $X \subseteq R^n$, asigna un número real, cumpliendo con las propiedades del producto escalar.

$$k: X \cdot X \rightarrow R$$

Las propiedades que debe cumplir una función kernel son:

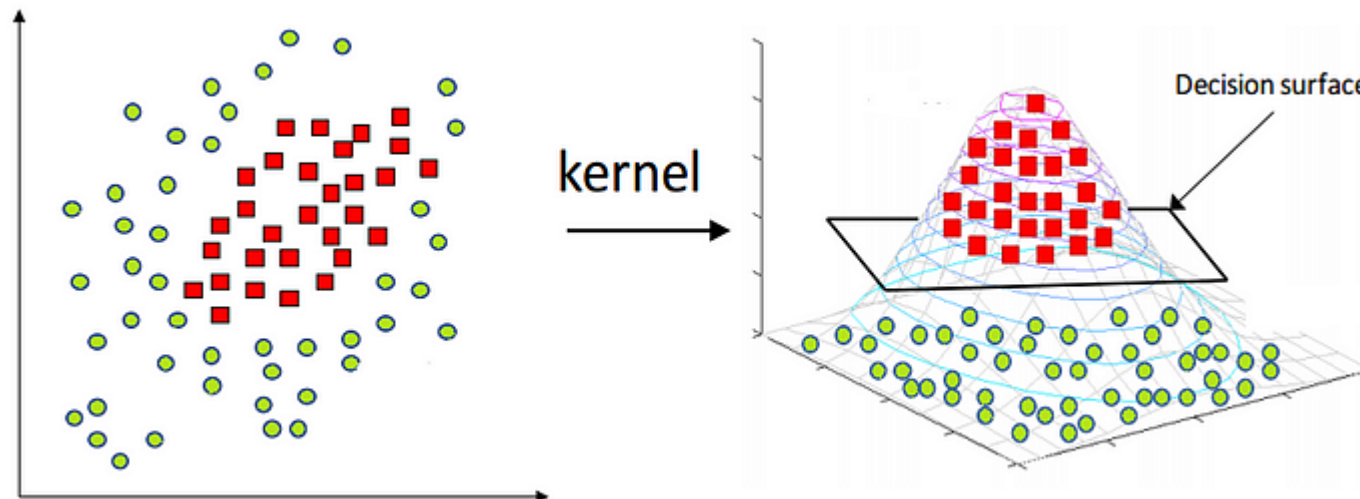
- La propiedad distributiva.
- La propiedad conmutativa.
- La propiedad semidefinida positiva.

SVM: Funciones kernel

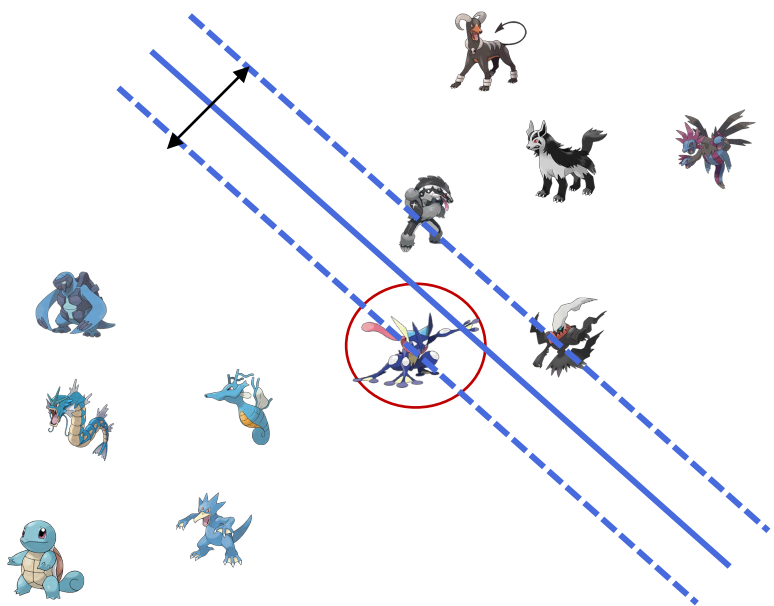
En términos algebraicos el producto escalar es la suma de los productos de los correspondientes componentes de cada vector.

En términos geométricos podemos pensar el producto escalar como la proyección de un vector sobre otro.

Una función kernel no es más que una versión «modificada» de la definición clásica de producto escalar. Este tipo de funciones reciben el nombre de funciones kernel porque se construyen a partir de una asignación de pesos a los distintos componentes de cada vector.



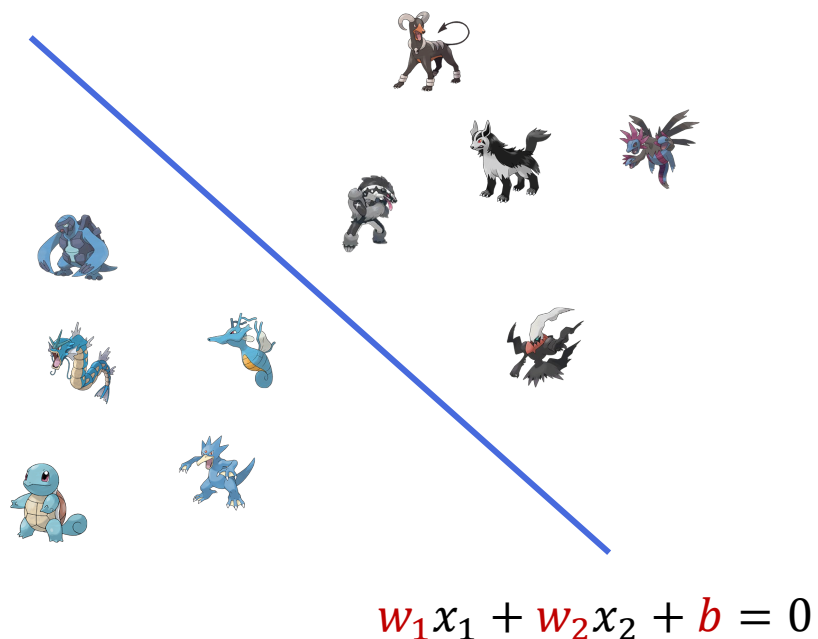
SVM: Presencia de outliers y el parámetro C



Si aplicamos el algoritmo veremos que se obtendrá un hiperplano óptimo pero que el margen *será muy pequeño*.

Esto se debe a que los vectores de soporte están más cerca y puede llevar al *overfitting*, i.e., que si introducimos un dato nuevo, que no haya sido nunca visto por el algoritmo, muy probablemente será clasificado incorrectamente porque el margen es muy reducido y no hay una adecuada separación entre las clases.

SVM: Presencia de outliers y el parámetro C

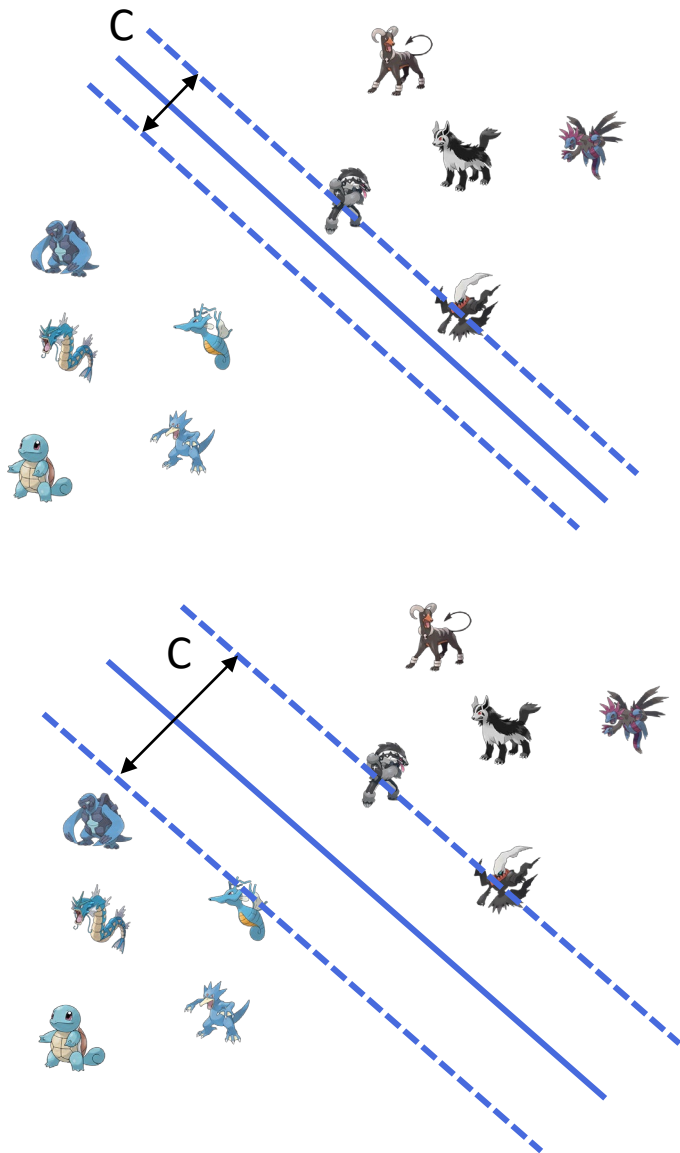


Así que el algoritmo SVM, conocido como hard margin (margen duro) no resulta muy flexible y no es ideal usarlo si hay outliers como el de nuestro corredor excepcional.

La manera de resolver esto es ensanchando el margen, permitiendo que la mayoría de los datos estén clasificados correctamente, y aceptando la posibilidad de que existan unos cuantos errores al momento de la clasificación.

Esto se logra modificando el algoritmo para obtener el hiperplano óptimo: originalmente en el clasificador hard margin se buscaba maximizar el margen, y para esto se modificaban únicamente los coeficientes del hiperplano.

SVM: Presencia de outliers y el parámetro C



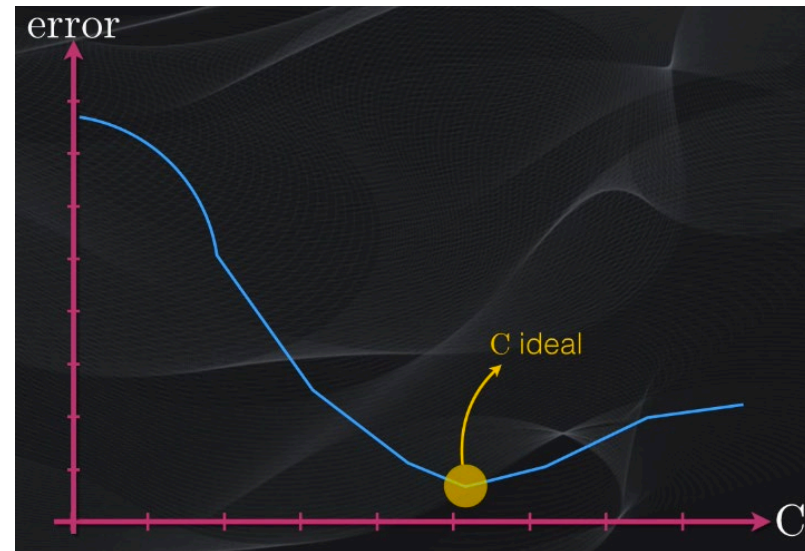
Ahora, para contrarrestar el efecto del outlier, se incluye un término adicional a esta función, lo que permite flexibilizar el margen.

Este término depende de un parámetro C , un hiperparámetro, que yo como diseñador elijo durante el entrenamiento.

La idea es que un C relativamente pequeño permite generar márgenes amplios, y a medida que su tamaño aumenta el margen se va reduciendo poco a poco

SVM: Costo Computacional y el parámetro C

Este parámetro se escoge de manera empírica analizando el error que se obtiene en la clasificación vs. diferentes valores de C, y a este algoritmo de máquina de vectores de soporte se le conoce como “soft margin”.



Costo computacional

Debido a la utilización de kernels, en el ajuste de un SVM participa una matriz $n \times n$, donde n es el número de observaciones de entrenamiento. Por esta razón, lo que más influye en el tiempo de computación necesario para entrenar un SVM es el número de observaciones, no el de predictores.

SVM: Tipos de kernels

Una de las tareas que realiza el algoritmo SVM es identificar los vectores de soporte, i.e., aquellos vectores que marcaran la trayectoria del hiperplano separador. Los vectores de soporte marcan la frontera.

Kernel lineal

$$K(u, v) = u^T \cdot v$$

Kernel polinomial

$$K(u, v) = (\gamma u^T \cdot v + b)^p$$

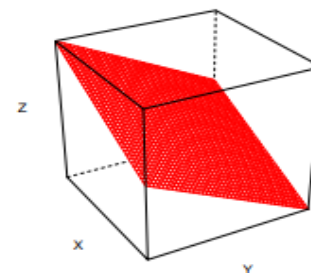
Kernel radial

$$K(u, v) = e^{-\gamma \|u - v\|^2}$$

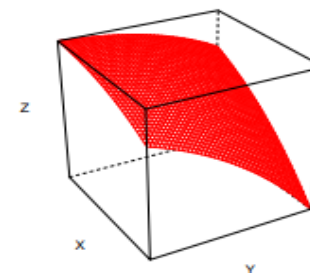
Kernel sigmoidal

$$K(u, v) = \tanh(\gamma u^T \cdot v + b)$$

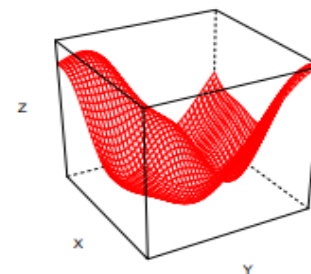
donde $u, v \in X$ y $\gamma > 0$, $b \geq 0$ y $p > 0$ son parámetros.



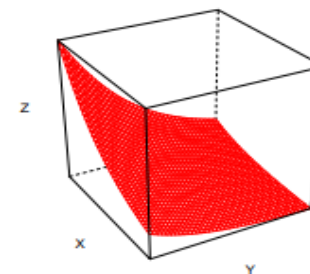
(a) Kernel lineal



(b) Kernel polinomial



(c) Kernel radial



(d) Kernel sigmoidal

SVM: Clasificación multi-clase

El concepto de hiperplano de separación en el que se basan los SVMs no se generaliza de forma natural para más de dos clases. Se han desarrollado numerosas estrategias con el fin de aplicar este algoritmo a problemas multiclase, de entre ellos, los más empleados son: one-versus-one, one-versus-all y DAGSVM

One-versus-one

Supóngase un escenario en el que hay $K > 2$ clases y que se quiere aplicar el método de clasificación basado en SVMs. La estrategia de one-versus-one consiste en generar un total de $K(K-1)/2$ SVMs, comparando todos los posibles pares de clases.

Para generar una predicción, se emplean cada uno de los $K(K-1)/2$ clasificadores, registrando el número de veces que la observación es asignada a cada una de las clases.

Finalmente, se considera que la observación pertenece a la clase a la que ha sido asignada con más frecuencia.

La principal desventaja de esta estrategia es que el número de modelos necesarios se dispara a medida que aumenta el número de clases, por lo que no es aplicable en todos los escenarios.

SVM: Clasificación multi-clase

One-versus-all

Esta estrategia consiste en ajustar K SVMs distintos, cada uno comparando una de las K clases frente a las restantes $K-1$ clases. Como resultado, se obtiene un hiperplano de clasificación para cada clase.

Para obtener una predicción, se emplean cada uno de los K clasificadores y se asigna la observación a la clase para la que la predicción resulte positiva.

Esta aproximación, aunque sencilla, puede causar inconsistencias, ya que puede ocurrir que más de un clasificador resulte positivo, asignando así una misma observación a diferentes clases.

Otro inconveniente adicional es que cada clasificador se entrena de forma no balanceada.

Por ejemplo, si el set de datos contiene 100 clases con 10 observaciones por clase, cada clasificador se ajusta con 10 observaciones positivas y 990 negativas.

SVM: Clasificación multi-clase

DAGSVM

DAGSVM (Directed Acyclic Graph SVM) es una mejora del método one-versus-one.

La estrategia seguida es la misma, pero consiguen reducir su tiempo de ejecución eliminando comparaciones innecesarias gracias al empleo de una directed acyclic graph (DAG).

Supóngase un set de datos con cuatro clases (A, B, C, D) y 6 clasificadores entrenados con cada posible par de clases (A-B, A-C, A-D, B-C, B-D, C-D).

Se inician las comparaciones con el clasificador (A-D) y se obtiene como resultado que la observación pertenece a la clase A, o lo que es equivalente, que no pertenece a la clase D.

Con esta información se pueden excluir todas las comparaciones que contengan la clase D, puesto que se sabe que no pertenece a este grupo.

En la siguiente comparación se emplea el clasificador (A-C) y se predice que es A. Con esta nueva información se excluyen todas las comparaciones que contengan C.

Finalmente solo queda emplear el clasificador (A-B) y asignar la observación al resultado devuelto. Siguiendo esta estrategia, en lugar de emplear los 6 clasificadores, solo ha sido necesario emplear 3.

DAGSVM tiene las mismas ventajas que el método one-versus-one pero mejorando mucho el rendimiento.

Ejemplo en python

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import style
import seaborn as sns
from mlxtend.plotting import plot_decision_regions
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn import metrics
```

```
plt.rcParams['image.cmap'] = "bwr"
plt.rcParams['savefig.bbox'] = "tight"
style.use('ggplot') or plt.style.use('ggplot')
```

```
import warnings
warnings.filterwarnings('ignore')
```


Ejemplo en python

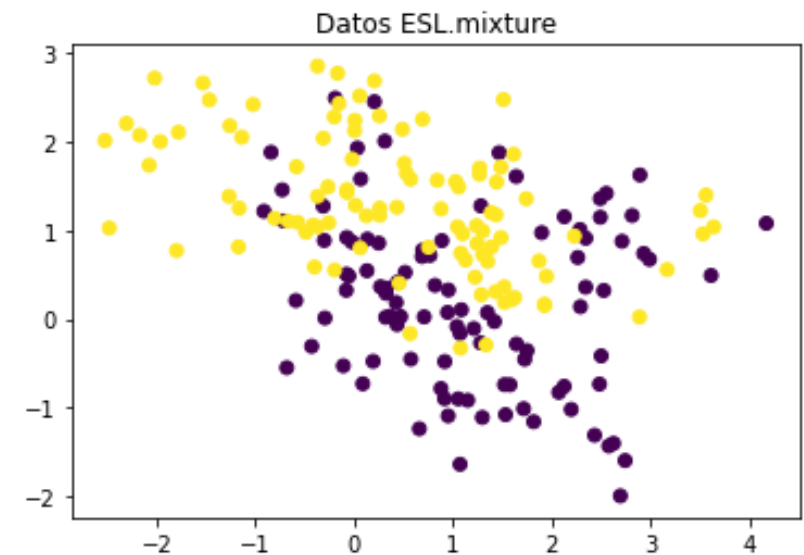
```
url = 'https://raw.githubusercontent.com/JoaquinAmatRodrigo/' \
      + 'Estadistica-machine-learning-python/master/data/ESL.mixture.csv'
datos = pd.read_csv(url)
datos.head(3)
```

```
fig, ax = plt.subplots(figsize=(6,4))
ax.scatter(datos.X1, datos.X2, c=datos.y);
ax.set_title("Datos ESL.mixture");
```

```
X = datos.drop(columns = 'y')
y = datos['y']
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y.values.reshape(-1,1),
                                                    train_size = 0.8, random_state = 1234, shuffle = True)
```

```
modelo = SVC(C = 100, kernel = 'linear', random_state=123)
modelo.fit(X_train, y_train)
```



Ejemplo en python

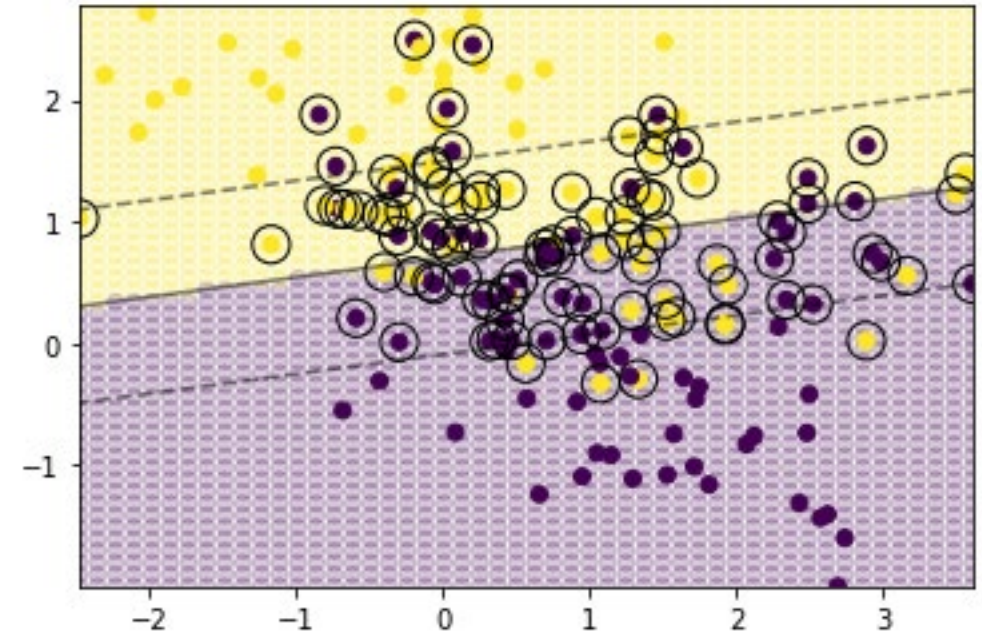
```
x = np.linspace(np.min(X_train.X1), np.max(X_train.X1), 50)
y = np.linspace(np.min(X_train.X2), np.max(X_train.X2), 50)
Y, X = np.meshgrid(y, x)
grid = np.vstack([X.ravel(), Y.ravel()]).T
```

```
pred_grid = modelo.predict(grid)
```

```
fig, ax = plt.subplots(figsize=(6,4))
ax.scatter(grid[:,0], grid[:,1], c=pred_grid, alpha = 0.2)
ax.scatter(X_train.X1, X_train.X2, c=y_train, alpha = 1)
```

```
ax.scatter(
    modelo.support_vectors_[:, 0],
    modelo.support_vectors_[:, 1],
    s=200, linewidth=1, facecolors='none', edgecolors='black')
```

```
ax.contour(X, Y, modelo.decision_function(grid).reshape(X.shape),
    colors = 'k', levels = [-1, 0, 1], alpha = 0.5, linestyles = ['--', '-', '--'])
```



Ejemplo en python

```
predicciones = modelo.predict(X_test)
predicciones

array([1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0,
       1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0], dtype=int64)

accuracy = accuracy_score(y_true = y_test, y_pred = predicciones,
                           normalize = True)
print("")
print(f"El accuracy de test es: {100*accuracy}%")

El accuracy de test es: 70.0%
```

Ejemplo en python

```
# SVM radial
param_grid = {'C': np.logspace(-5, 7, 20)}

grid = GridSearchCV(
    estimator = SVC(kernel= "rbf", gamma='scale'),
    param_grid = param_grid,
    scoring = 'accuracy', n_jobs = -1,
    cv = 3, verbose = 0,
    return_train_score = True
)

# Se asigna el resultado a _ para que no se imprima por pantalla
_ = grid.fit(X = X_train, y = y_train)

resultados = pd.DataFrame(grid.cv_results_)
resultados.filter(regex = '(param.*|mean_t|std_t)')\
    .drop(columns = 'params')\
    .sort_values('mean_test_score', ascending = False) \
    .head(5)
```

Ejemplo en python

	param_C	mean_test_score	...	mean_train_score	std_train_score
8	1.128838	0.762520	...	0.790778	0.035372
12	379.269019	0.750641	...	0.868777	0.007168
7	0.263665	0.750175	...	0.778228	0.026049
9	4.83293	0.744118	...	0.815729	0.026199
11	88.586679	0.738062	...	0.859431	0.019840

[5 rows x 5 columns]

```
print("Mejores hiperparámetros encontrados (cv)")  
print(grid.best_params_, ":", grid.best_score_, grid.scoring)
```

```
modelo = grid.best_estimator_
```

```
{'C': 1.1288378916846884} : 0.7625203820172374 accuracy
```

Ejemplo en python

```
x = np.linspace(np.min(X_train.X1), np.max(X_train.X1), 50)
y = np.linspace(np.min(X_train.X2), np.max(X_train.X2), 50)
Y, X = np.meshgrid(y, x)
grid = np.vstack([X.ravel(), Y.ravel()]).T
```

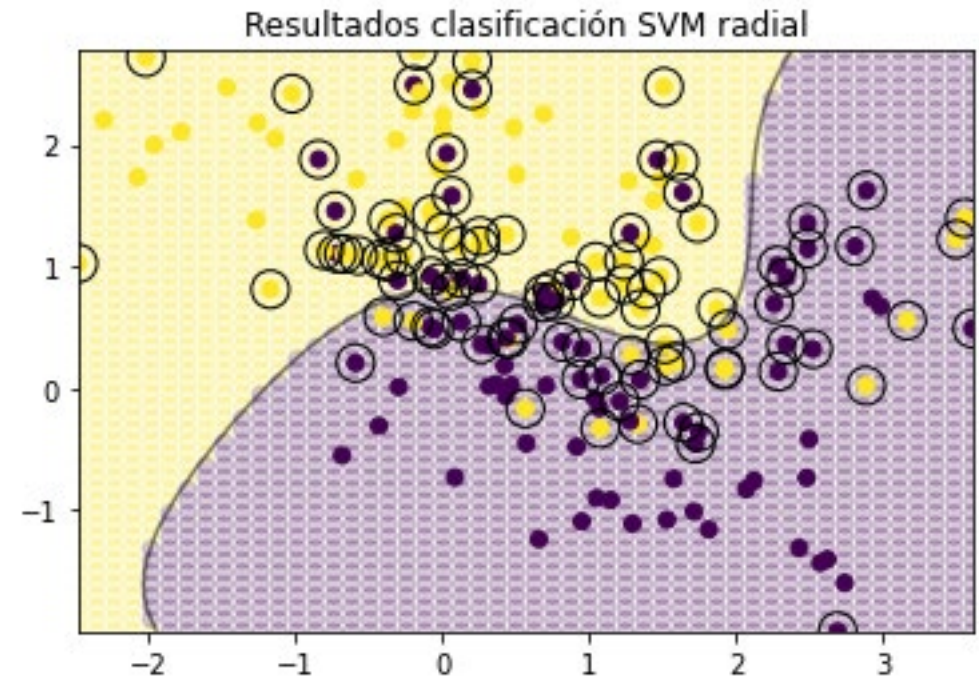
```
pred_grid = modelo.predict(grid)
```

```
fig, ax = plt.subplots(figsize=(6,4))
ax.scatter(grid[:,0], grid[:,1], c=pred_grid, alpha = 0.2)
ax.scatter(X_train.X1, X_train.X2, c=y_train, alpha = 1)
```

```
ax.scatter(
    modelo.support_vectors_[:, 0],
    modelo.support_vectors_[:, 1],
    s=200, linewidth=1, facecolors='none', edgecolors='black')
```

```
ax.contour(X, Y,
    modelo.decision_function(grid).reshape(X.shape),
    colors='k', levels=[0], alpha=0.5, linestyles='-')
```

```
ax.set_title("Resultados clasificación SVM radial");
```



Ejemplo en python

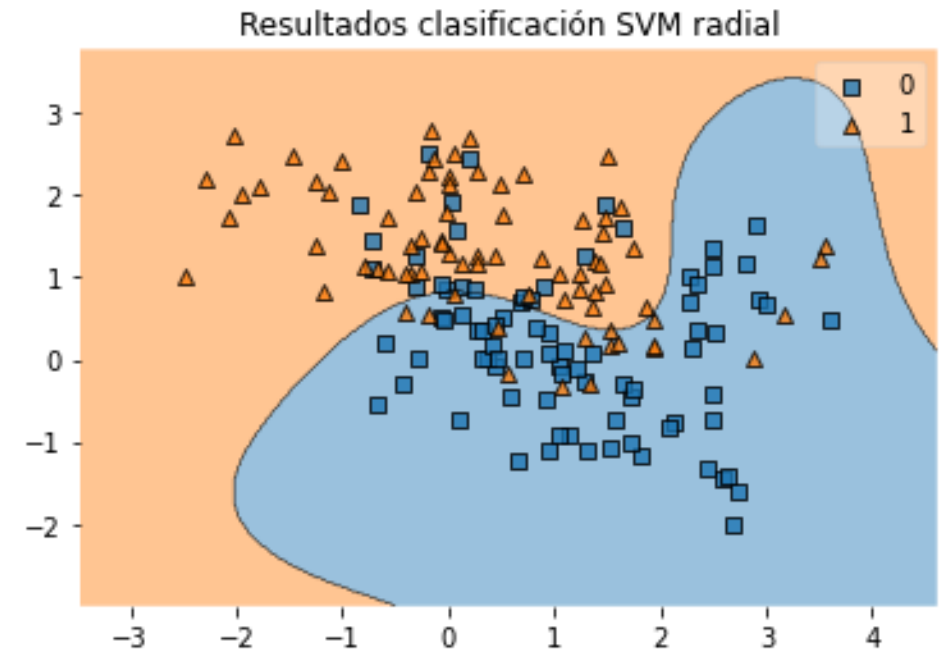
```
fig, ax = plt.subplots(figsize=(6,4))
plot_decision_regions(
    X = X_train.to_numpy(),
    y = y_train.flatten(), clf = modelo, ax = ax)

ax.set_title("Resultados clasificación SVM radial");

predicciones = modelo.predict(X_test)

accuracy = accuracy_score(
    y_true = y_test,
    y_pred = predicciones,
    normalize = True
)
print("")
print(f"El accuracy de test es: {100*accuracy}%")
```

El accuracy de test es: 80.0%



Ejemplo en python

```
confusion_matrix = pd.crosstab(  
    y_test.ravel(),  
    predicciones,  
    rownames=['Real'],  
    colnames=['Predicción']  
)
```

confusion_matrix

	Predicción	0	1
Real			
0		14	3
1		5	18