Resumen del capítulo: Distancia entre vectores

Producto escalar

Si multiplicamos todas las componentes y luego sumamos los valores obtenidos, tendremos el **producto escalar** o **producto punto**. Como resultado de esta operación con dos vectores del mismo tamaño obtenemos un nuevo número. Ese número se llama **escalar**.

Aquí está la fórmula para un producto escalar de dos vectores a=[x1, x2… xn] y b=[y1, y2… yn]:

$$(a, b) = x_1 \times y_1 + x_2 \times y_2 + \ldots + x_n \times y_n$$

El producto escalar de los vectores *a* y *b* generalmente se denota entre paréntesis (a, b) o a un punto $a \cdot b$.

En NumPy, podemos encontrar el producto escalar usando la función `numpy.dot()`:

```
import numpy as np

dot_value = np.dot(vector1, vector2)
```

El **operador de multiplicación de matrices** nos permite calcular el producto escalar aún más fácilmente. El operador se marca con `@`:

```
import numpy as np

volume = np.array([0.1, 0.3, 0.1])
content = np.array([0.4, 0.0, 0.1])

dot_value = vector1 @ vector2
```

También existe la multiplicación elemento por elemento. A diferencia del producto escalar, el resultado será un vector:

```
import numpy as np

vector3 = vector1 * vector2
```

Distancia planar

La longitud del vector o módulo es igual a la raíz cuadrada del producto escalar del vector y este mismo. Por ejemplo, para el vector $a=(x, y)$, se calcula así:

$$|a| = \sqrt{(a, a)} = \sqrt{x^2 + y^2}$$

Para medir la distancia entre dos puntos, es decir, para obtener la raíz cuadrada de las diferencias de los vectores, vamos a buscar la **distancia euclidiana**. Esta calcula la distancia más corta usando el teorema de Pitágoras: el cuadrado de la hipotenusa es igual a la suma de los cuadrados de los otros lados.

La distancia euclidiana se puede escribir así: $d_2(a, b)$. La d lleva el subíndice 2 para indicar que las coordenadas del vector están elevadas a la segunda potencia.

La distancia entre los puntos $a(x1, y1)$ y $b(x2, y2)$ se calcula mediante la fórmula:

$$d_2(a, b) = \sqrt{(a - b, a - b)} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Encontremos la distancia euclidiana entre los puntos *a*=(5, 6) y *b*=(1, 3):

```
import numpy as np

a = np.array([5, 6])
b = np.array([1, 3])
d = np.dot(a-b, a-b)**0.5
print('La distancia entre a y b es igual a', d)
```

SciPy tiene una librería dedicada al cálculo de distancias, que se llama **distance**. Llama a la función `distance.euclidean()` para encontrar la distancia euclidiana:

```
import numpy as np
from scipy.spatial import distance

a = np.array([5, 6])
b = np.array([1, 3])
d = distance.euclidean(a, b)
print('La distancia entre a y b es igual a', d)
```

Los resultados del cálculo son los mismos.

Distancia Manhattan

Distancia Manhattan o **distancia entre manzanas** es la suma de módulos de diferencias de coordenadas. El nombre se debe a que el trazado de las calles de Manhattan hace imposible utilizar la distancia euclidiana (que se calcula mediante la línea recta).

Vamos a calcular la distancia Manhattan entre los puntos *a=(x1, y1)* y *b=(x2, y2)* con la siguiente fórmula:

$$d_1(a, b) = |x_1 - x_2| + |y_1 - y_2|$$

La distancia Manhattan se formula como $d_1(a, b)$. La d lleva el subíndice 1 para indicar que las coordenadas del vector están elevadas a la primera potencia (el número no cambia).

La función para calcular la distancia Manhattan en SciPy se llama `distance.cityblock()`:

```
import numpy as np
from scipy.spatial import distance

a = np.array([5, 6])
b = np.array([1, 3])
d = distance.cityblock(a, b)
print('La distancia entre a y b es igual a', d)
```

Para encontrar el índice mínimo en la matriz NumPy, llama a la función `argmin()`.

```
index = np.array(distances).argmin() # índice de elemento mínimo
```

Distancias en el espacio multidimensional

En machine learning, los vectores son características de las observaciones. Por lo general, los vectores son multidimensionales en lugar de bidimensionales.

La distancia euclidiana entre los vectores $a=(x1, x2\ldots xn)$ y $b=(y1, y2\ldots yn)$ es la suma de los cuadrados de las diferencias de coordenadas:

$$d_2(a, b) = \sqrt{(y_1 - x_1)^2 + \ldots + (y_n - x_n)^2} = \sqrt{\sum_{i=1}^{n} (y_i - x_i)^2}$$

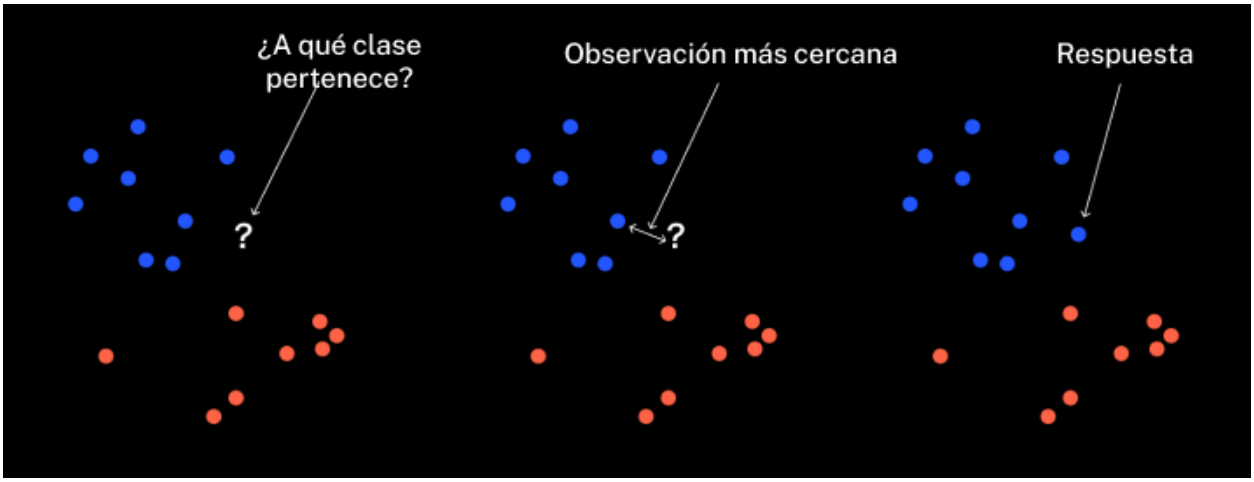
La distancia Manhattan es la suma de módulos de diferencias de coordenadas:

$$d_1(a, b) = |x_1 - y_1| + |x_2 - y_2| + \ldots + |x_n - y_n| = \sum_{i=1}^{n} |x_i - y_i|$$

Incluso cuando el número de coordenadas es superior a dos, utilizamos las conocidas funciones `distance.euclidean()` y `distance.cityblock()` para calcular distancias en el espacio multidimensional.

Algoritmo de vecinos más cercanos

Observa la imagen de abajo. ¿Cómo podemos predecir la clase de observación? Podemos encontrar el objeto más cercano en la muestra y obtener la respuesta de él. Así es como funciona el **algoritmo de vecinos más cercanos**. Por lo general, buscamos la observación más cercana en el conjunto de entrenamiento.



El algoritmo funciona tanto en el plano como en el espacio multidimensional, en cuyo caso las distancias se calculan mediante fórmulas multidimensionales.

Creación de clase modelo

Clase (class) es un nuevo tipo de datos con sus propios métodos y atributos. Echemos un vistazo al modelo constante para una tarea de regresión. Este predice respuestas basadas en el valor medio del objetivo en el conjunto de entrenamiento.

Para crear una nueva clase, especifica la palabra clave `class` seguida del nombre de la clase.

```
class ConstantRegression:
    # contenido de clase con un offset (desplazamiento) de cuatro espacios
    # ...
```

Para entrenar el modelo, vamos a utilizar el método `fit()`. Es una función dentro de la clase y el primer parámetro siempre es `self`. `self` es una variable que almacena el modelo. Es necesaria para trabajar con los atributos. Otros dos parámetros son las características y el objetivo del conjunto de entrenamiento, al igual que en Sklearn.

```
class ConstantRegression:
    def fit(self, features_train, target_train):
        # contenido de la función con offset 4+4
        # ...
```

Al realizar el entrenamiento, debemos guardar el valor medio del objetivo. Para crear el nuevo atributo `value`, agrega `self` con un punto al principio del nombre de la variable. De esta forma, indicamos que la variable está dentro de la clase:

```
class ConstantRegression:
    def fit(self, features_train, target_train):
        self.value = target_train.mean()
```

Vamos a usar el método `predict()` para predecir la respuesta, que es la media guardada:

```
class ConstantRegression:
    def fit(self, features_train, target_train):
        self.value = target_train.mean()
```

```
    def predict(self, new_features):
        answer = pd.Series(self.value, index=new_features.index)
        return answer
```