Valorar conjuntos de características

Hemos visto como hacer selección de caracteristicas evaluando características individualmente.

Supongamos ahora que tenemos una tabla de datos que responde a un problema que desconocemos. Este problema es la función XOR (o-exclusivo, *eXclusive-OR*) y, por tanto, la tabla de valores tendrá ejemplos de las 4 posibilidades siguientes:

In [166]:
```
import numpy as np
import pandas as pd

tabla_xor = pd.DataFrame(
    {'x1' : [True,  True,  False, False],
     'x2' : [True,  False, True,  False],
     'XOR': [False, True,  True,  False]}
)
tabla_xor
```

Out[166]:

	x1	x2	XOR
0	True	True	False
1	True	False	True
2	False	True	True
3	False	False	False

Supongamos además que, como los datos no son perfectos, resulta que también tenemos los valores de otras dos características 'r1' y 'r2' que sospechamos que podrían significar algo pero realmente son sólo ruido aleatorio. Así que podríamos tener el siguiente conjunto de datos. No se ha hecho aleatorio para evitar resultados distintos y evitar relaciones casuales mientras se mantiene el ejemplo de un tamaño pequeño para ver sus probabilidades a simple vista.

In [167]:
```
tabla_xor = pd.DataFrame(
    {'x1' :  [True,  True,  False, False, True,  True,  False, False],
     'x2' :  [True,  False, True,  False, True,  False, True,  False],
     'r1' :  [False, True,  False, True,  True,  False, True,  False],
     'r2' :  [False, False, False, False, True,  True,  True,  True ],
     'clase':[False, True,  True,  False, False, True,  True,  False]}
)
tabla_xor
```

Out[167]:

	x1	x2	r1	r2	clase
0	True	True	False	False	False
1	True	False	True	False	True
2	False	True	False	False	True
3	False	False	True	False	False
4	True	True	True	True	False
5	True	False	False	True	True
6	False	True	True	True	True
7	False	False	False	True	False

Podemos ver que en todas las características incluida la clase:

$$P(\text{True}) = P(\text{False}) = 0.5$$

In [168]:
```
caracteristicas = [c for c in tabla_xor]
for c in caracteristicas:
    print('P({})={}'.format(c, sum(tabla_xor[c]) / len(tabla_xor)), end='    ')
```

```
P(x1)=0.5    P(x2)=0.5    P(r1)=0.5    P(r2)=0.5    P(clase)=0.5
```

Ademas, la probabilidad de que la clase sea cierta condicionada a cada una de las características es también:

$$P(clase|x_1 = \text{True}) = P(clase|x_2 = \text{True}) = P(clase|r_1 = \text{True}) = P(clase|x_2 = \text{True}) = 0.5$$

y, por extensión de lo anterior, la probabilidad de que la clase sea falsa condicionada a cada una de las características es también 0.5.

In [169]:
```
for c in caracteristicas:
    condicionados = []
    for i in range(len(tabla_xor[c])):
        if tabla_xor[c][i]:
            condicionados.append( (tabla_xor[c][i], tabla_xor['clase'][i]) )
    print('P(clase|{})={}'.format(c,
                                  len([c for c in condicionados if c == (True, True)]) / len(c
ondicionados)),
          end='    ')
```

```
P(clase|x1)=0.5    P(clase|x2)=0.5    P(clase|r1)=0.5    P(clase|r2)=0.5    P(clase|clase)=1.
0
```

Lo anterior quiere decir que la clase es estadísticamente independiente de cada una de las características. Para rizar más el rizo, en este ejemplo hemos hecho que todas las características sean independientes entre sí (como ocurriría si fuesen aleatorias).

In [170]:
```
for dependiente in caracteristicas[:-1]:
    for c in caracteristicas[:-1]:
        condicionados = []
        for i in range(len(tabla_xor[c])):
            if tabla_xor[c][i]:
                condicionados.append( (tabla_xor[c][i], tabla_xor[dependiente][i]) )
        print('P({}|{})={}'.format(dependiente,
                                   c,
                                   len([c for c in condicionados if c == (True, True)]) / len
(condicionados)),
              end='    ')
    print()
```

```
P(x1|x1)=1.0    P(x1|x2)=0.5    P(x1|r1)=0.5    P(x1|r2)=0.5
P(x2|x1)=0.5    P(x2|x2)=1.0    P(x2|r1)=0.5    P(x2|r2)=0.5
P(r1|x1)=0.5    P(r1|x2)=0.5    P(r1|r1)=1.0    P(r1|r2)=0.5
P(r2|x1)=0.5    P(r2|x2)=0.5    P(r2|r1)=0.5    P(r2|r2)=1.0
```

Por tanto, este problema parece una pesadilla para la selección de características. De hecho, cualquier medida de características que sólo use los valores de una característica fallará estrepitosamente. Sin embargo, nosotros sabemos que el problema tiene solución y que con las características x_1 y x_2 se puede conseguir un 100% de acierto.

Entonces, podemos concluir que necesitamos medidas que consideren varias características juntas. Estas serán las medidas sobre conjuntos de características que veremos a continuación.

Medidas basadas en consistencia

Miden la utilidad de un conjunto de características según lo cerca es este conjunto está de conseguir clasificar acertadamente todos los ejemplos. Esto es: cómo de lejos estamos de poder distinguir las clases con esas caracteristicas. Para ello pueden contarse los ejemplos que no se pueden diferenciar. Por ejemplo, si tenemos:

x1	r1	clase
T	F	T
T	F	F

Vemos que hay dos ejemplos iguales en las caracteríticas seleccionadas (x1 y r1 en este ejemplo) pero con distinto valor para la clase. Aquí, la de los pares de ejemplos inconsistentes (IEP) contaría un par de ejemplos inconsistentes. La medida de Liu (o EI, de los Ejemplos Inconsistentes) consideraría que hay 1 ejemplo mal clasificado (el otro estará bien). La basada en la teoría de conjuntos rugosos contaría 2 ejemplos mal, ya que no sabemos realmente nada de ninguno de los dos.

Lectura: Lee el siguiente artículo para ver la definición completa de las medidas de consistencia y la justificación de la implementación que haremos a continuación.

Arauzo-Azofra, A., Benitez, J. M., & Castro, J. L. (2008). Consistency measures for feature selection. Journal of Intelligent Information Systems, 30(3), 273-292. (https://sci2s.ugr.es/sites/default/files/ficherosPublicaciones/0824_2008-arauzo-JIIS.pdf)

In [171]:
```
import collections

def inconsistent_examples(seleccionadas, objetivo, tabla):
    """
    Ratio de inconsistencia (Liu et al., 1998),
     valorado en el rango [0,1[ (0 = completamente consistente)
    """
    x, y = tabla[seleccionadas], tabla[[objetivo]]

    # Contar ejemplos de cada patron
    tabla_hash = collections.defaultdict(collections.Counter)
    for ejemplo, obj in zip(x.itertuples(), y.itertuples()):
        tabla_hash[tuple(ejemplo[1:])][obj[1]] += 1

    # Calcular ejemplos inconsistentes (clases minoritarias en cada patrón)
    contador_inc = 0
    for ej in tabla_hash:
        total = sum(tabla_hash[ej].values())
        clase_mayoritaria = tabla_hash[ej].most_common(1)[0][1]
        contador_inc += total - clase_mayoritaria

    return contador_inc / len(tabla.index)
```

Vemos que al pasarle las dos características que determinan bien la clase la consistencia es total:

In [172]:
```
inconsistent_examples(['x1', 'x2'], 'clase', tabla_xor)
```

Out[172]: 0.0

En cambio si le pasamos un conjunto de las aleatorias, detecta inconsistencia:

In [173]:
```
inconsistent_examples(['r1', 'r2'], 'clase', tabla_xor)
```

Out[173]: 0.5

Y cualquier conjunto que incluya las características que determinan la clase también será consistente. Esto se cumple por la propiedad de la monotonicidad que menciona el artículo:

In [174]:
```
inconsistent_examples(['x1', 'x2', 'r1', 'r2'], 'clase', tabla_xor)
```

Out[174]: 0.0

Ejercicio: implementar la medida RSC. Está definida formalmente en el artículo anterior ("Consistency measures for feature selection"). Pista, por si esa definición resulta dificil de entender: cuando en un grupo hay algún ejemplo conflicitivo se cuenta como que todos son inconsistentes.

Ejercicio (avanzado, opcional): implementar la medida IEP. Se puede hacer con el mismo esquema de las otras pero hay que contar el número de pares de ejemplos inconsistentes que se generan en cada grupo.

La implementación de arriba usa tablas hash y funciona muy bien con menos de 100 características seleccionadas. En el siguiente artículo teneis más información sobre la implementación eficiente de estas medidas, con optimizaciones para cuando se van a usar repetidamente sobre los mismos datos. Os puede interesar por lo menos echar un vistazo a su Fig. 19.

Arauzo-Azofra, A., Jiménez-Vílchez, A., Molina-Baena, J., & Luque-Rodriguez, M. (2019). Algorithmic cache of sorted tables for feature selection. Data Mining and Knowledge Discovery, 33(4), 964-994. (https://www.researchgate.net/profile/Antonio-Arauzo-Azofra/publication/331990030_Algorithmic_cache_of_sorted_tables_for_feature_selection/links/5ca4ce58a6fdcc12ee911191/Algorithmic-cache-of-sorted-tables-for-feature-selection.pdf)

Medidas de información

Basadas en la teoría de la información de Shanon (https://es.wikipedia.org/wiki/Teor%C3%ADa_de_la_informaci%C3%B3n) se pueden generar varias medidas para conjuntos de caracteristicas. Vamos a ver como ejemplo la principal (de la que derivan las demás), la medida de la información mútua (https://es.wikipedia.org/wiki/Informaci%C3%B3n_mutua) que aportan las características seleccionadas sobre la clase.

In [175]:
```
import math

def informacion_mutua(seleccionadas, objetivo, tabla):
    """
    Información mútua de Shannon
    """
    x, y = tabla[seleccionadas], tabla[[objetivo]]

    # Contar ejemplos de cada patron
    tabla_hash = collections.defaultdict(collections.Counter)
    for ejemplo, obj in zip(x.itertuples(), y.itertuples()):
        tabla_hash[tuple(ejemplo[1:])][obj[1]] += 1

    # Suma las cuentas de las clases para obtener P(C)
    class_counter = collections.Counter()
    for ej in tabla_hash:
        class_counter += tabla_hash[ej]

    n = len(y)

    # H(class)
    hc = 0
    for _, c in class_counter.items():
        p = float(c) / n
        if p > 0:
            hc += - p * math.log(p, 2)

    # H(class|selection)
    hc_s = 0
    for ex, ex_class_counter in tabla_hash.items():
        s_count = sum(ex_class_counter.values())
        h = 0
        for _, c in ex_class_counter.items():
            p = float(c) / s_count
            if p > 0:
                h += p * math.log(p, 2)

        hc_s += - ( float(s_count) / n ) * h

    # I(S,C) = H(C) - H(C|S)
    return hc - hc_s

informacion_mutua(['x1', 'r2'], 'clase', tabla_xor)
```

Out[175]: 0.0

In [176]:
```
informacion_mutua(['x1', 'x2'], 'clase', tabla_xor)
```

Out[176]: 1.0

Estrategia envolvente

Otra forma de evaluar conjuntos de caracteristicas que es muy empleada es la llamada estrategia envolvente. Utilizar el mismo algoritmo de aprendizaje que se usará luego, para usar su rendimiento sobre el conjunto de características como la valoración de ese conjunto. Para evitar un sobre-ajuste se realiza una particion de los datos usando una parte para entrenar (*train*) y otra para valorar el rendimiento (*test*).

In [177]:
```
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

def envolvente(seleccionadas, objetivo, tabla):
    X_train, X_test, y_train, y_test = train_test_split(tabla[seleccionadas],
                                                        tabla[objetivo], test_size=0.33)
    clf = SVC()
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    return accuracy_score(y_test, y_pred, normalize=True)

envolvente(['x1', 'r2'], 'clase', tabla_xor)
```

Out[177]: 0.0

In [178]:
```
envolvente(['x1', 'x2'], 'clase', tabla_xor)
```

Out[178]: 1.0