Extracción de características

La primera técnica que vamos a ver para la reducción de la información que pasamos a nuestro futuro proceso de aprendizaje es la extracción de características. Se trata de generar un número menor de nuevas características a partir de las que tenemos.

Análisis de componentes principales

Normalmente conocida por sus siglas en inglés, PCA (*Principal Component Analysis*), es probablemente la técnica de extracción de características más conocida.

PCA trabaja sobre datos continuos. Así que debemos haber convertido previamente todas las características a valores continuos.

PCA supone relaciones lineales entre las características por lo que puede no ser adecuada en algunos problemas sin aplicar transformaciones previas. Los algoritmos de aprendizaje basados en máquinas de vectores soporte, que se verán en otras asignaturas, hacen uso de esas transformaciones para funcionar eficazmente en muchos conjuntos de datos.

In [237]:
```
import numpy as np
import pandas as pd
```

Vamos a trabajar con un conjunto de datos pequeño inventado para ver los valores en el proceso. Por conveniencia vamos a separarlo en dos partes: X representa los vectores de características de las variables independientes e Y que representa la variable dependiente.

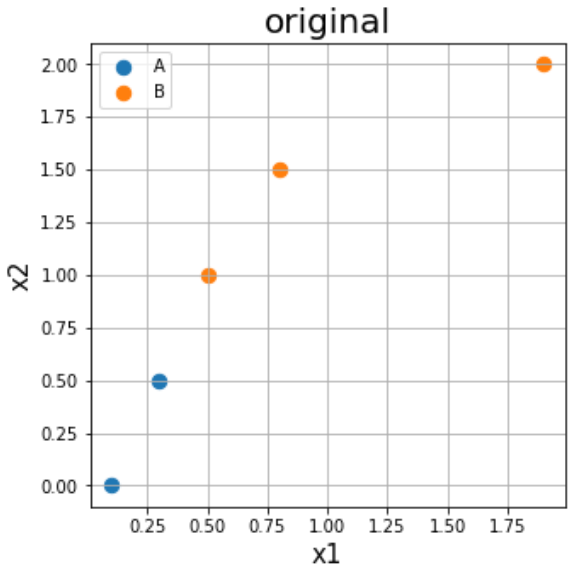
PCA trabaja sólo sobre las características, ignorando la variable dependiente. Por tanto, es lo que llamaremos una técnica no supervisada (se verá en posteriores asignaturas). Lo que estamos aplicando es una especie de técnica de compresión (con perdida) de los datos.

In [238]:
```
tabla = pd.DataFrame(
    {'x1' : [0.1, 0.3, 0.5, 0.8, 1.9],
     'x2' : [0.0, 0.5, 1.0, 1.5, 2.0],
     'y'  : ['A', 'A', 'B', 'B', 'B']
})
x = np.array( tabla.loc[:,['x1', 'x2']] )
y = np.array( tabla.loc[:,['y']] )
```

A continuación lo representamos gráficamente. Observamos que hay bastante relación entre ambas variables (a menor x1, menor x_2 y viceversa). Esta es la relación que queremos capturar para reducir a menos características.

Hemos representado las clases de la variable dependiente (A y B) como referencia de ejemplo de lo que un futuro algoritimo de aprendizaje tendrá que aprender. De esta forma vemos que la clase de objetos con valores pequeños de x_1 y x_2 es A y la de los que tienen valores grandes B. *PCA* ignora esto.

In [239]:
```
from matplotlib import pyplot
fig = pyplot.figure(figsize = (5,5))
ax = fig.add_subplot(1,1,1)
ax.set_xlabel('x1', fontsize = 15)
ax.set_ylabel('x2', fontsize = 15)
ax.set_title('original', fontsize = 20)
objetivos = ['A', 'B']
colores = ['r', 'b']
for obj in objetivos:
    seleccionados = np.array(y == obj)[:,0]
    ax.scatter(x[seleccionados, 0], #x1
               x[seleccionados, 1], #x2
               s=70)
ax.legend(objetivos)
ax.grid()
```



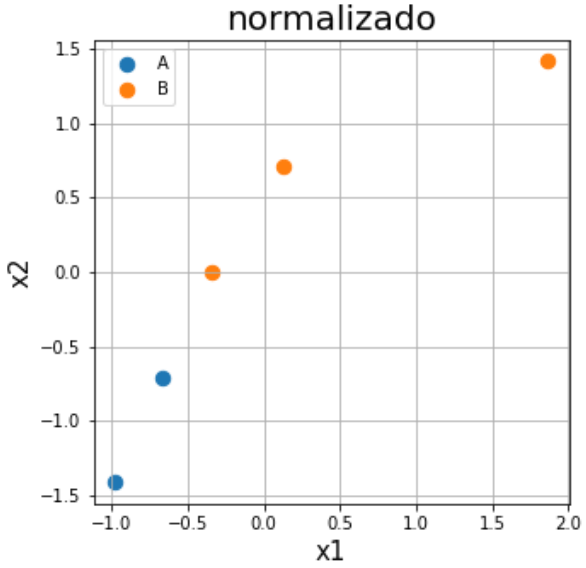
El análisis de componentes principales se hace sobre las matrices de covarianza. Por esta razón, si una característica tuviese valores en el rango de los miles se consideraría mucho mas importante que una variable cuyos valores están en el rango de las decenas. Normalmente no deseamos esto. Para hacer que *PCA* sea independiente de las unidades en que se midan las propiedades (por ejemplo, metros o kilometros) y, en general, independiente del rango en el que las variables tomen valores, lo habitual es normalizar los valores (vimos como hacerlo en el tema 2.1):

In [240]:
```
from sklearn import preprocessing
x = preprocessing.StandardScaler().fit_transform(x)
x
```

Out[240]:
```
array([[-0.97835132, -1.41421356],
       [-0.66275412, -0.70710678],
       [-0.34715692,  0.        ],
       [ 0.12623888,  0.70710678],
       [ 1.86202349,  1.41421356]])
```

La forma es la misma pero cambiando los valores de las variables. Veamos la gráfica.

In [241]:
```
fig = pyplot.figure(figsize = (5,5))
ax = fig.add_subplot(1,1,1)
ax.set_xlabel('x1', fontsize = 15)
ax.set_ylabel('x2', fontsize = 15)
ax.set_title('normalizado', fontsize = 20)
objetivos = ['A', 'B']
colores = ['r', 'b']
for obj in objetivos:
    seleccionados = np.array(y == obj)[:,0]
    ax.scatter(x[seleccionados, 0], #x1
               x[seleccionados, 1], #x2
               s=70)
ax.legend(objetivos)
ax.grid()
```



Podemos aplicar el análisis de componentes principales (*PCA*) usando la clase `PCA` del módulo `decomposition` de sklearn.

In [242]:
```
from sklearn import decomposition
pca = decomposition.PCA(n_components=1)
pca.fit(x)
```

Out[242]:
```
PCA(n_components=1)
```

Podemos ver los componentes calculados en el siguiente atributo del objeto `pca` creado. En este caso sólo hay un componente (porque pusimos `n_components=1`), que tiene los valores de los pesos asignados a las dos variables originales.

In [243]:
```
pca.components_
```

Out[243]:
```
array([[0.70710678, 0.70710678]])
```

De esta forma, podemos transformar los datos originales, para que tengan una única variable que retenga gran parte de la información de las dos variables, multiplicando la matriz de datos original por los componentes:

In [244]:
```
x @ pca.components_.T
```

Out[244]:
```
array([[-1.69179886],
       [-0.96863793],
       [-0.24547701],
       [ 0.58926437],
       [ 2.31664944]])
```

Podemos ver el ratio de la varianza que se conserva de los datos originales en el siguiente atributo:

```
In [245]: pca.explained_variance_ratio_
```

```
Out[245]: array([0.95747989])
```

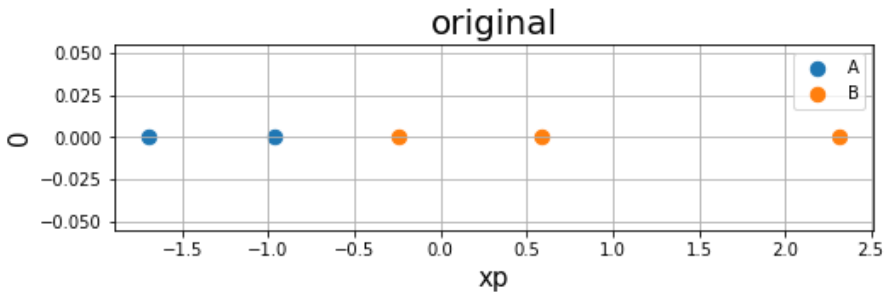
En vez de hacer el `.fit(x)` y luego multiplicar nosotros por los pesos, se puede hacer directamente llamando a `fit_transform`. (Antes lo hemos hecho por pasos para ilustrar el proceso y comprobar ahora que el resultado es el mismo.)

```
In [246]: xp = pca.fit_transform(x)
          xp
```

```
Out[246]: array([[-1.69179886],
                 [-0.96863793],
                 [-0.24547701],
                 [ 0.58926437],
                 [ 2.31664944]])
```

De esta forma, hemos reducido la dimensión de los datos de dos a una sola variable. Podemos ver en la representación de los datos siguiente, que seguimos pudiendo diferenciar fácilmente los objetos de clase A frente a los de clase B usando la nueva variable xp.

```
In [247]: fig = pyplot.figure(figsize = (8,2))
          ax = fig.add_subplot(1,1,1)
          ax.set_xlabel('xp', fontsize = 15)
          ax.set_ylabel('0', fontsize = 15)
          ax.set_title('original', fontsize = 20)
          objetivos = ['A', 'B']
          for obj in objetivos:
              seleccionados = np.array(y == obj)[:,0]
              ax.scatter(xp[seleccionados, 0],
                         np.zeros([sum(seleccionados)]),
                         s=70)
          ax.legend(objetivos)
          ax.grid()
```



Podemos aplicar esto sobre datos con más dimensiones para ayudarnos a visualizarlo [1]. Por ejemplo, es dificil representar un problema con cuatro dimensiones pero podemos ver como queda si lo reducimos a dos.

In [248]:
```
iris = pd.read_csv("https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data",
                   names=['sepal length','sepal width','petal length','petal width','target'])
iris
```

Out[248]:

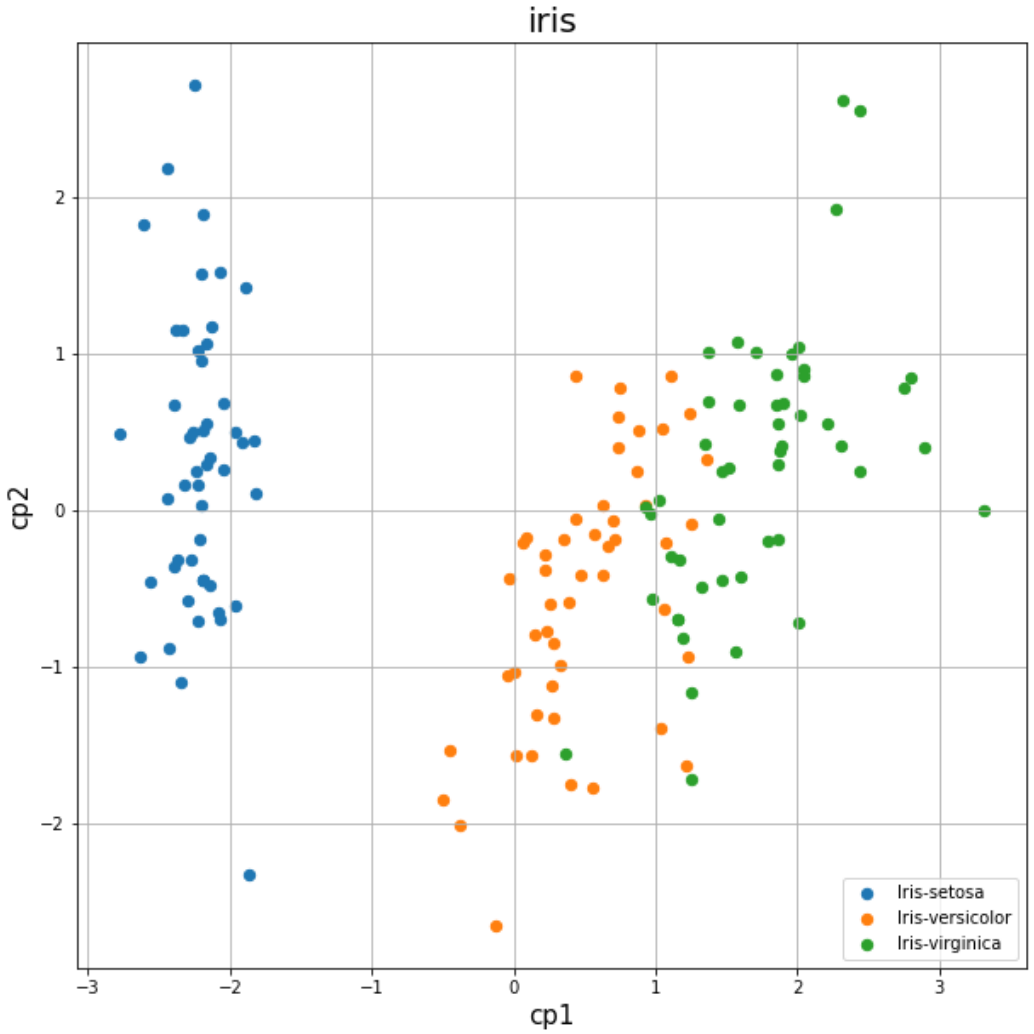
	sepal length	sepal width	petal length	petal width	target
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa
...	...	...	...	...	...
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

150 rows × 5 columns

In [249]:
```
# Convertir a ndarray y normalizar
caracteristicas = ['sepal length', 'sepal width', 'petal length', 'petal width']
x = df.loc[:, caracteristicas].values
y = df.loc[:,['target']].values
x = preprocessing.StandardScaler().fit_transform(x)

# Componentes principales
pca = decomposition.PCA(n_components=2)
xpca = pca.fit_transform(x)

# Gráfica
fig = pyplot.figure(figsize = (10,10))
ax = fig.add_subplot(1,1,1)
ax.set_xlabel('cp1', fontsize = 15)
ax.set_ylabel('cp2', fontsize = 15)
ax.set_title('iris', fontsize = 20)
objetivos = ['Iris-setosa', 'Iris-versicolor', 'Iris-virginica']
colores = ['r', 'g', 'b']
for obj, color in zip(objetivos, colores):
    seleccionados = np.array(y == obj)[:,0]
    ax.scatter(xpca[seleccionados, 0], xpca[seleccionados, 1], s=40)
ax.legend(objetivos, loc="lower right")
ax.grid()
```



Podemos ver como con dos componentes una de las clases se puede diferenciar muy bien, mientras que las otras dos no tanto. Podemos ver que la varianza explicada entre los dos componentes es muy alta (casi el 96%):

In [250]:
```
pca.explained_variance_ratio_
```

Out[250]:
```
array([0.72770452, 0.23030523])
```

Podría ser que esa poca información perdida haga funcionar algo peor a un algoritmo de aprendizaje y, con problemas tan pequeños, quizá no merezca la pena esta reducción. Sin embargo, es muy útil en problemas más grandes. En la referencia [1], hay también un ejemplo de como se puede aplicar en el reconocimiento de digitos (un problema con 784 características) y en la referencia [2] de como puede resultar útil aplicado antes del aprendizaje con redes neuronales profundas.

Ejercicio: Aplica *PCA* a los datos sobre vino que vimos en el tema 1.3.

[1] PCA using Python (scikit-learn) https://towardsdatascience.com/pca-using-python-scikit-learn-e653f8989e60 (https://towardsdatascience.com/pca-using-python-scikit-learn-e653f8989e60)

[2] Principal Component Analysis (PCA) in Python https://www.datacamp.com/community/tutorials/principal-component-analysis-in-python (https://www.datacamp.com/community/tutorials/principal-component-analysis-in-python)