1. Introducción

El concepto de modelo es un elemento fundamental en la representación de problemas y en el aprendizaje automático (*machine learning*).

En la RAE, entre otras definiciones, se encuentren:

♢ Arquetipo o punto de referencia para imitarlo o reproducirlo.

♢ Esquema teórico, generalmente en forma matemática, de un sistema o de una realidad compleja, como la evolución económica de un país, que se elabora para facilitar su comprensión y el estudio de su comportamiento.

1.1. Algoritmo y Modelo

Algoritmo

En *machine learning* es un proceso que utiliza datos para generar un modelo. Es decir, los algoritmos "*aprenden*" de los datos proporcionados.

Modelo

El modelo representa lo aprendido por el algoritmo de *machine learning*.

2. Aprendizaje no supervisado

En el *aprendizaje no supervisado* el proceso se realiza de manera automática, sin la necesidad de ningún supervisor externo (etiquetas de clase). Para ello se emplean técnicas de agrupamiento, reducción de dimensiones y descubrimiento de patrones, gracias a las cuales el sistema selecciona y aprende los objetos que poseen características similares, determinando automáticamente las clases, características y relaciones importantes.

2.1. Análisis de agrupamiento (*clustering*)

En ocasiones es posible dividir una colección de observaciones en distintos subgrupos, basados solamente en los atributos de las observaciones; cuando se puede hacer esta separación, se facilita el entendimiento de la población o el proceso que generó las observaciones. La intención es realizar división de datos en *grupos* (*clusters*) de observaciones que son más similares dentro de un grupo que entre varios grupos. Los grupos son formados ya sea agregando observaciones o dividiendo un gran grupo de observaciones en una colección de grupos más pequeños.

El proceso de generación de grupos incluye dos variedades de algoritmos:

1. Combinar observaciones entre un número fijo de grupos buscando maximizar la similitud dentro de cada grupo

2. Comenzar con grupos de un solo elemento y, recursivamente combinarlos; alternativamente, se puede comenzar con un sólo grupo y recursivamente dividir nuevos grupos

En esta sección se revisarán dos algoritmos populares y representativos de cada una de estas propuestas: agrupación jerárquica y k-medias. Sea $D = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ el conjunto de datos, con $\mathbf{x}_i$ un vector de atributos medidos en una unidad observacional.

2.1.1. Agrupamiento por $k-$medias

A diferencia del agrupamiento jerárquico por aglomeración, en este algoritmo, el analista establece el número de grupos; el algoritmo comienza asignando arbitrariamente los vectores del conjunto de datos $D = \{x_1, \ldots, x_n\}$ a k grupos; estos grupos iniciales se denotan como $\{A_1, \ldots, A_k\}$. Después, se calculan los *centroides* de cada grupo; estos centroides son medias multivariadas de las observaciones de cada grupo y cada centroide representa al grupo para el cálculo de las distancias. También es posible asignar arbitrariamente k medias iniciales y con ellas generar los grupos.

Es muy poco probable que la configuración inicial sea una buena solución, por tanto el algoritmo itera entre dos pasos:

♢ asignar cada observación al grupo más cercano

♢ recalcular los centroides de cada grupo

Si al menos una observación es reasignada en otro grupo, los centroides cambiarán y debe realizarse otra iteración; el algoritmo continuará pasando entre estos dos pasos hasta que ya no hay reasignaciones. En ese momento, cada observación pertenece al grupo que le es más cercano. A partir de la configuración aleatoria inicial, la suma de cuadrados dentro de cada grupo ha sido minimizado; esto se debe a que dicha suma de cuadrados es equivalente a la suma de distancias euclideanas entre las observaciones y los centroides; además, mover cualquiera de las obervaciones incrementará la suma de distancias (y la suma de cuadrados dentro de los grupos). Por tanto, el algoritmo ha alcanzado *una mejor* asignación posible para las observaciones en los grupos y *un mejor* cálculo de los centroides.

Este algoritmo tiene un atractivo considerable porque minimza una función objetivo popular: la suma de cuadrados. Su inconveniente es que debe generar una configuración inicial aleatoria: una configuración diferente por lo general regresa un resultado distinto.

El centroide del grupo A_i es la media multivariada:

$$\overline{\mathbf{x}}_i = n_i^{-1} \sum_{\mathbf{x}_j \in A_i} \mathbf{x}_j,$$

donde, n_i es el número de observaciones que pertenecen a A_i y $\mathbf{x}_j = [x_{j,1}, \ldots, x_{j,h}]^T$; el número de atributos es h y el $l-$ésimo elemento de $\overline{\mathbf{x}}_i$ es:

$$\overline{x}_{i,l} = n_i^{-1} \sum_{\mathbf{x}_j \in A_i} x_{j,l},$$

para $l = 1, \ldots, h$; la distancia entre cada observación $\mathbf{x}_j$ y un grupo A_i, se define como la distancia entre la observación y el centroide del grupo. Como $\overline{\mathbf{x}}_i = [\overline{x}_{i,1}, \ldots, \overline{x}_{i,h}]^T$, la distancia euclideana cuadrada es:

$$d_E^2 (\mathbf{x}_j, \overline{\mathbf{x}}_i) = \sum_{l=1}^{h} (x_{j,l} - \overline{x}_{i,l})^2.$$

Se puede utilizar la distancia cuadrada en lugar de la distancia, dado que el ordenamiento de más cercano a más distante será el mismo.

Ejemplo

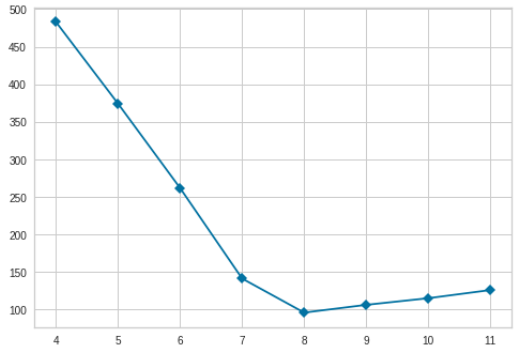
Clasificar las muestras siguientes utilizando $k = 2$:
$[8, 10], [3, 10.5], [7, 13.5], [5, 18], [5, 13], [6, 9], [9, 11], [3, 18], [8.5, 12], [8, 16]$

Ejemplo con Python

El conjunto de datos *xclara* es muy usado para probar este tipo de algoritmos

2.1.1.1. Determinar un buen valor para *k* Al igual que en la mayoría de los modelos, determinar los hiperparámetros requiere iterar el mismo y encontrar un indicador que nos permita elegir su valor.

Para el caso de $k-$medias, la *gráfica de codo* brinda una buena guía para este fin.



Este método recomienda usar como valor para k aquel que corresponde con el codo, es decir, en el que se tiene el mayor cambio en la pendiente.

Tarea: _____

♢ Clasificar las muestras siguientes utilizando k=*2*:
$[1, 12.5], [3, 10.5], [3, 12.5], [3, 14.5], [3, 18], [5, 18], [5, 16], [5, 14.5], [5, 13], [6, 9], [8, 10], [9, 11], [8.5, 12], [7, 13.5], [8, 16], [0.5, 10.5]$

_____*

2.1.2. Modelos de mezclas gausianas (*Gaussian Mixed Models*)

Similar a k-medias, en estos modelos se establece una cantidad k de grupos para iniciar y se inicializan tanto la medias como las varianzas.

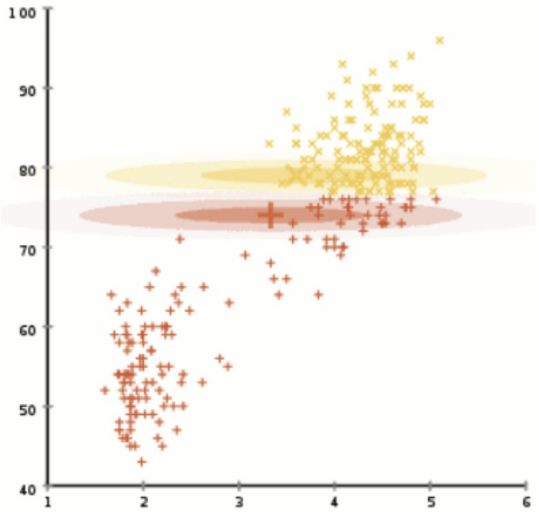


Figura 1: Paso inicial de un modelo de mezclas gaussianas

Iterativamente actualiza la media y la varianza de cada grupo, así como la probabilidad de pertenencia de cada punto a cada grupo y el peso del mismo (cantidad de muestras de cada grupo).

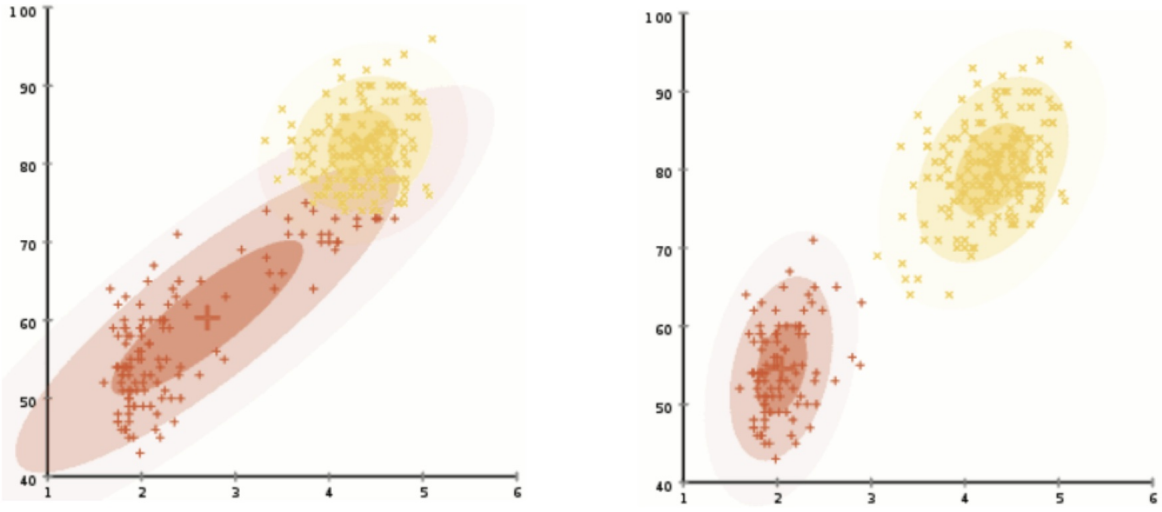


Figura 2: Evolución de un modelo de mezclas gaussianas

Esperanza-Maximización (*Expectation-Maximization*), es la técnica más popular para determinar los parámetros de un modelo de mezclas gaussianas. Se compone de dos pasos:

♢ Paso E: asignar de forma probabilística cada muestra a cada clase, basándose en la hipótesis actual de los parámetros.

♢ Paso M: actualizar las hipótesis de los parámetros, en función de las asignaciones actuales.

En el paso E se calcula el valor esperado de las asignaciones de grupos. En el paso M se obtiene la nueva máxima verosimilitud de las hipótesis.

Modelos de mezclas gaussianas vs k-medias

Si bien el algoritmo inicia con un número predefinido de grupos al igual que k-means, existen diferencias fundamentales:

♢ k-medias establece un círculo (hiperesfera) al centro del grupo, con radio definido por el punto más distante

♢ GMM funciona mejor cuando los datos no se ajustan a circunferencias (hiperesferas)

♢ k-medias realiza clasificación dura: devuelve la clase a la que pertenece una muestra, mientras GMM hace clasificación suave: devuelve la probabilidad de pertenencia a cada clase.

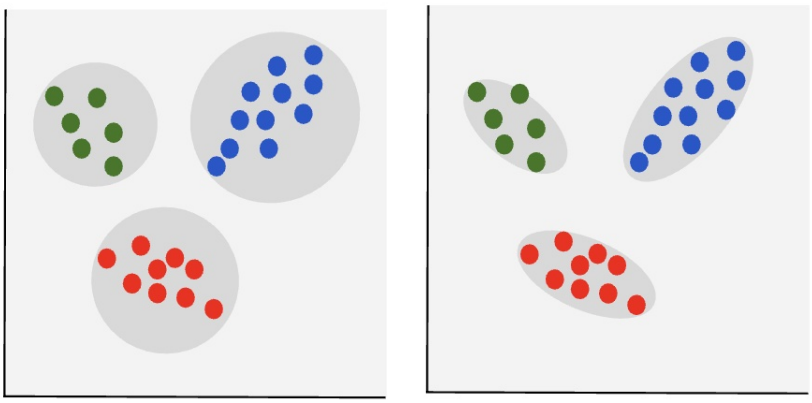


Figura 3: k-medias vs GMM

Ejemplo:

Importando bibliotecas:

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import pyplot as plt
from sklearn.mixture import GaussianMixture
from sklearn.datasets.samples_generator import make_blobs
```

Generamos conjuntos de prueba:

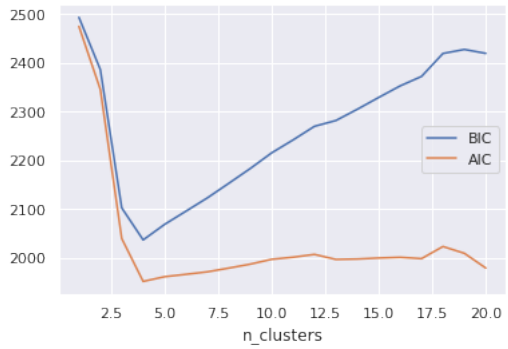
```
X, y = make_blobs(n_samples=300, centers=4, cluster_std=0.6, random_state=0)
plt.scatter(X[:,0],X[:,1])
```

Podemos obtener el número óptimo de grupos con ayuda de los criterios:

♢ *Akaike information criterion*(AIC)

♢ *Bayesian information criterion*(BIC)

```
n_clusters = np.arange(1, 21)
models = [GaussianMixture(n, covariance_type='full',
                          random_state=0).fit(X) for n in n_clusters]
plt.plot(n_clusters, [m.bic(X) for m in models], label='BIC')
plt.plot(n_clusters, [m.aic(X) for m in models], label='AIC')
plt.legend(loc='best')
plt.xlabel('n_clusters')
```



En este caso el criterio indica elegir el valor que minimiza el criterio elegido y para nuestro ejemplo coincide para ambos.

2.2. Detección de anomalías

Detección de anomalías (*outliers*) es la identificación de eventos u observaciones que no son acordes a los patrones esperados en cierto conjunto de datos. Se utiliza comúnmente para eliminar valores anómalos en conjuntos de datos, buscando mejorar el rendimiento de algoritmos y modelos de apredizaje automático.

Figura 4: Anomalía

2.2.1. Desviación estándar

Es una forma simple de detectar valores anómalos en un conjunto de datos.

Si los datos tienen una distribución normal, entonces el 68 % de los mismos se encuentran a una desviación estándar de la media.; el 95 % estarán a dos desviaciones y 99.7 %, a tres desviaciones.

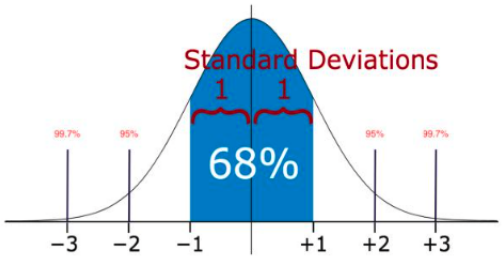


Figura 5: Método de la desviación estándar

Por ejemplo, un código sencillo para detectar anomalías se presenta a continuación.

Biblioteca y conjunto de datos ficticio con distribución normal:

```
import numpy as np
data = np.random.randn(50000) * 20 + 20
```

Función para encontrar anomalías:

```
def find_anomalies(data):
  anomalies = []
  # límites superior e inferior
  data_std = np.std(data)
  data_mean = np.mean(data)
  anomaly_cut = data_std * 3
  low_lim = data_mean - anomaly_cut
  upp_lim = data_mean + anomaly_cut
 # Obtenemos las anomalías
  for d in data:
    if d > upp_lim or d < low_lim:
      anomalies.append(d)
  return anomalies
```

Y probamos con el conjunto ficticio:

```
anom = find_anomalies(data)
len(anom)
```

Nota: Es muy importante realizar alguna prueba de normalidad al conjunto de datos para verificar que cumple con el requisito, entre otras opciones, existen: histograma, gráfica de cuartiles, prueba de Shapiro.

2.2.2. Gráfica de *bigotes* (*boxplot*)

Una *boxplot* es una representación gráfica de datos números con sus cuartiles. Es una forma simple y muy efectiva de encontrar anomalías.

Los bigotes (whiskers) marcan los límites inferior y superior de la distribución de datos: cualquier dato fuera de esos límites se puede considerar anómalo.

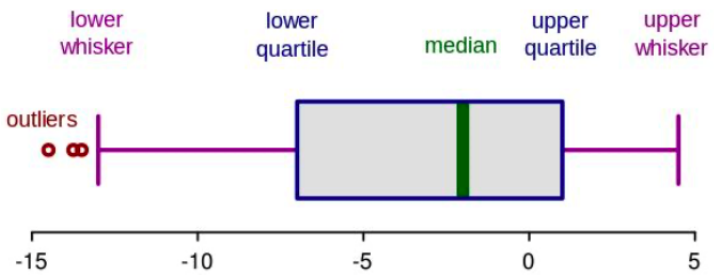


Figura 6: *Boxplot*

Para construirlo se utiliza el concepto de *rango intercuartil* (IQR), una medida de dispersión que divide un conjunto de datos en cuatro subgrupos.

Dados n datos:

♢ $IQR = Q3 - Q1$

♢ $Q1 =$ mediana de los $\frac{n}{2}$ datos menores

♢ $Q3 =$ mediana de los $\frac{n}{2}$ datos mayores

♢ $Q2 =$ mediana de todos los datos

♢ $lw = Q1 - 1.5 \times IQR$

♢ $uw = Q3 + 1.5 \times IQR$

Para un conjunto con distribución normal, se obtiene el siguiente diagrama:

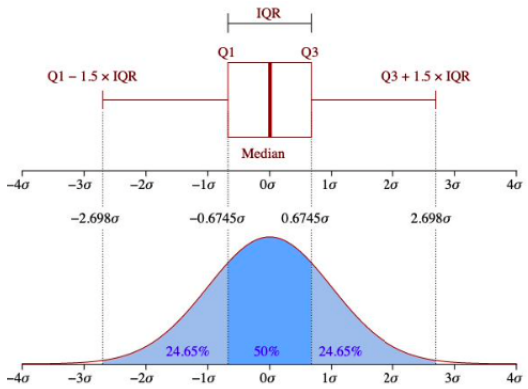


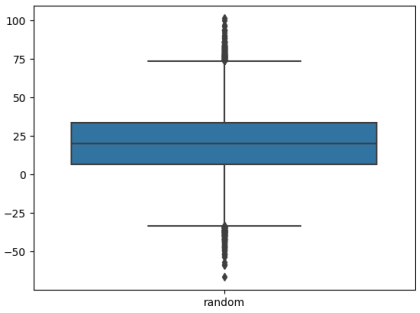
Figura 7: *Boxplot* de un conjunto con distribución normal

Por ejemplo, para un conjunto creado con distribución normal:

```
data = np.random.randn(50000) * 20 + 20
data = pd.DataFrame(data, columns=['random'])
data.head()
```

Podemos generar el *boxplot* con *seaborn*:

```
import seaborn as sns
sns.boxplot(data)
```

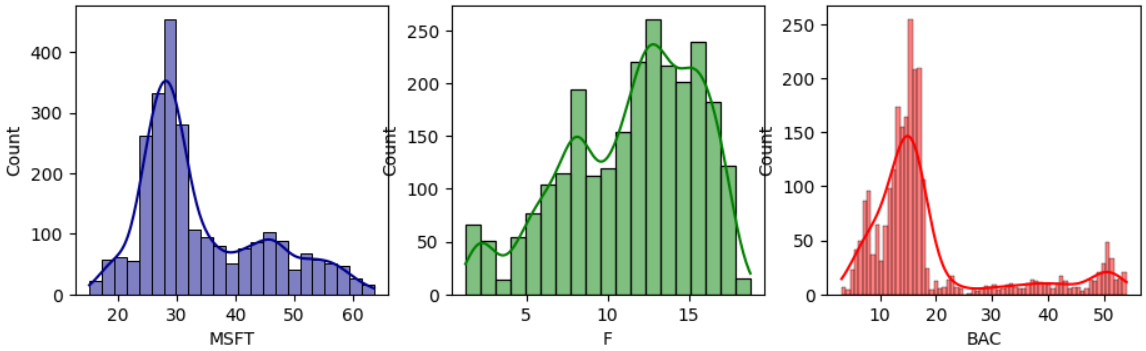


Un ejemplo más interesante con precios de acciones de compañías:

```
url = 'https://bit.ly/47bBIcL'
stocks = pd.read_csv(url, header='infer')
stocks.index = stocks.Date
stocks = stocks.drop(['Date'],axis=1)
stocks.head()
```

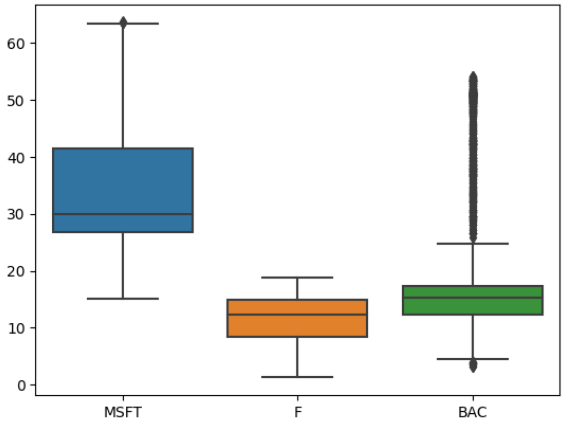
Exploramos los datos con un histograma para verificar si tienen distribución normal:

```
import matplotlib.pyplot as plt
import seaborn as sns
fig, ax = plt.subplots(nrows=1, ncols=3, figsize=(11,3))
sns.histplot(stocks.MSFT, ax=ax[0], color="darkblue", kde=True)
sns.histplot(stocks.F, ax=ax[1], color='green', kde=True)
sns.histplot(stocks.BAC, ax=ax[2], color='red', kde=True)
```



Notamos que no tienen distribución normal, entonces utilizamos un boxplot para ver las anomalías:

```
import seaborn as sns
sns.boxplot(data)
```


60
50
40
30
20
10
0
MSFT
F
BAC

2.3. Reducción de dimensiones

2.3.1. Seleccionando características significativas

Si se observa que un modelo tiene mejor rendimiento con el conjunto de entrenamiento que con el de pruebas, es un indicador de *sobreajuste*; esto significa que el modelo adecúa sus parámetros a las observaciones de los datos de entrenamiento, pero no generaliza bien a nuevos datos.

Entre las posibles soluciones al sobreajuste, se tiene la reducción de la dimensión de los datos.

2.3.1.1. Selección secuencial de características Los algoritmos de selección secuencial son una familia de algoritmos voraces (*greedy*) usados para reducir la dimensión d de características a un subespacio de dimensión k que cumple $k < d$. El objetivo es seleccionar automáticamente un subconjunto de características que sean las más relevantes para el problema; con esto se puede mejorar la eficiencia o reducir errores de generalización del modelo al eliminar características irrevantes o ruido.

Un algoritmo de selección común es el llamado ***Sequential Backward Selection*** (***SBS***) cuyo objetivo es reducir la dimensión del espacio de características con un mínimo de pérdidas en el rendimiento del modelo mejorando la eficiencia computacional. Además, SBS puede mejorar también el poder predictivo de un modelo si está sobreajustado.

La idea detrás de SBS es simple: eliminar secuencialmente características de un conjunto hasta que se tenga el número deseado de ellas. Para determinar qué columna eliminar se debe establecer una función criterio $\mathcal{J}$; uno muy usado es simplemente comparar el rendimiento del modelo antes y después de eliminar una columna; la característica a eliminar será aquella que cause el menor descenso en el rendimiento. El algoritmo puede describirse en cuatro pasos:

Algoritmo 1 *Sequential Backward Selection*

1. Inicializar el algoritmo con $k = d$, d es la dimensión del espacio original de características $\mathbf{X}_d$
2. Determinar la característica x^- que maximiza el criterio: $x^- = \arg\max\left(\mathcal{J}\left(X_k - x\right)\right)$, donde $x \in X_k$
3. Eliminar la característica x^- del conjunto: $X_{k-1} = X_k - x^-; k = k - 1$
4. Si k es mayor que el número deseado de características, repetir desde el paso 2; de otro modo, terminar la ejecución

2.3.1.2. Reducción no supervisada: Análisis de Componentes Principales (PCA) Otra forma de reducir la cantidad de datos a procesar, es la compresión de datos, dado que nos permite almacenar y analizar gran catidad de datos producidos y recolectados por medio tecnológicos. Mientras que SBS es un algoritmo de *selección*, PCA (***Principal Component Analysis***) es un algoritmo de *extracción* de características. La diferencia principal es que en un algoritmo de selección, se conservan las características originales y en uno de extracción, los datos se transforman o proyectan en un nuevo espacio de características.

PCA ayuda a identificar patrones en los datos basado en la correlación entre las características; es decir, intenta encontrar las direcciones de máxima varianza en datos con muchas dimensiones y

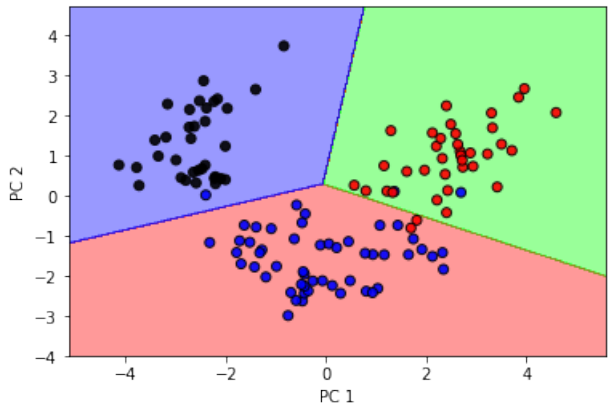
las proyecta en un nuevo subespacio con igual o menos dimensiones que el original. El algoritmo se compone de los siguientes pasos:

Algoritmo 2 *Principal Component Analysis*

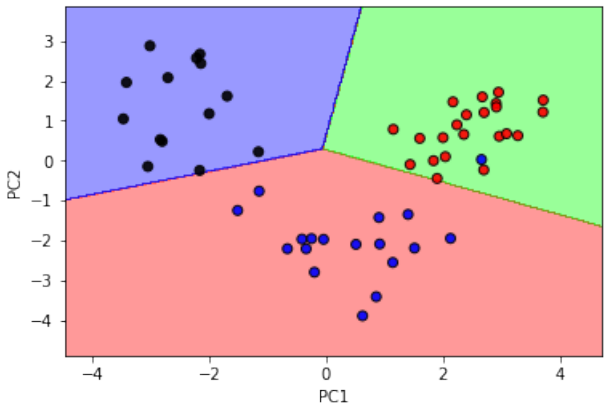
1. Estandarizar el conjunto de datos de dimensión d
2. Obtener la matriz de covarianza
3. Descomponer la matriz de covarianza en sus eigenvalores y eigenvectores
4. Ordenar los eigenvalores de manera decreciente de acuerdo a sus correspondientes eigenvectores
5. Seleccionar los k eigenvectores que corresponden con los k mayores eigenvalores; k es la dimensión de nuevo subespacio de características ($k < d$)
6. Construir una matriz de proyección $\mathbf{W}$ con los primeros k eigenvectores
7. Transformar el conjunto de datos de entrada $\mathbf{X}$ de dimensión d utilizando la matriz de proyección $\mathbf{W}$ para obtener el nuevo subespacio de características de dimensión k

Implementación de PCA desde cero:

Usando PCA de *scikit-learn*:




```
# Graficar los resultados para el conjunto de prueba
```



2.3.1.3. Reducción de dimensión supervisada: Análisis Lineal Discriminante (LDA) ***Linear Discriminant Analysis*** (***LDA***) es una técnica de extracción de características que puede usarse para incrementar la eficiencia computacional y reducir los sobreajustes. Formulado inicialmente por Ronald A. Fischer (https://bit.ly/3f9e6KF) en 1936 con el conjunto de datos de flores iris para problemas de clasificación de dos clases. En 1948 C. Radhakrishna Rao (https://bit.ly/2VX3Hub) lo generalizó para problemas multiclase bajo el supuesto de covarianzas de clase iguales y clases con distribuciones normales.

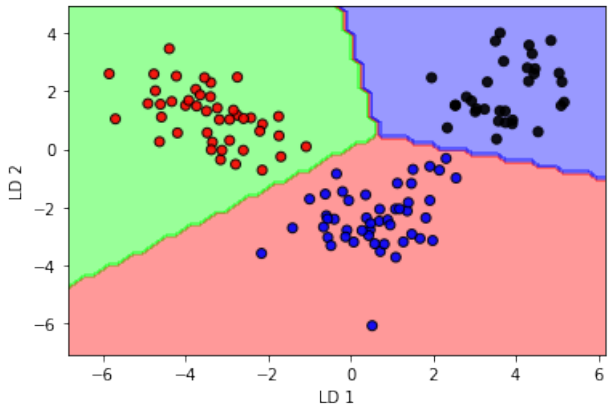
En general, los conceptos detrás de LDA son muy similares a PCA: mientras PCA busca las componentes ortogonales de varianza mínima, el objetivo de LDA es encontrar un subespacio de características que optimice la separabilidad de clases.

El algoritmo se compone de los siguientes pasos:

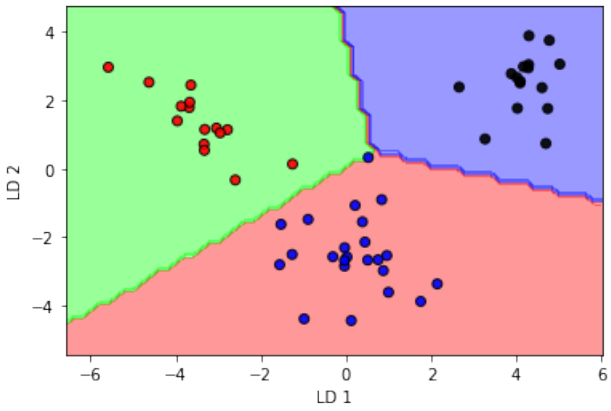
Algoritmo 3 *Linear Discriminant Analysis*

1. Estandarizar el conjunto de datos de dimensión d
2. Para cada clase, calcular su vector de medias (de dimensión d)
3. Obtener la matriz de dispersión (*scatter matrix*) entre clases S_B y la matriz de dispersión de la propia clase S_W
4. Determinar los eigenvalores y eigenvectores correspondientes a la matriz $S_W^{-1}S_B$
5. Ordenar los eigenvalores de manera decreciente de acuerdo a sus correspondientes eigenvectores
6. Seleccionar los k eigenvectores que corresponden con los k mayores eigenvalores para construir una $d \times k-$dimensional matriz de proyección $\mathbf{W}$; los eigenvectores son las columnas de dicha matriz
7. Proyectar las muestras sobre el nuevosubespacio de características utilizando la matriz de proyección $\mathbf{W}$

Usando LDA de *scikit-learn*:




```
# Graficar los resultados para el conjunto de prueba
```



2.3.1.4. Reducción no lineal de dimensión: Análisis de Componentes Principales por Núcleos (KPCA) Algunos algoritmos de aprendizaje automático suponen separabilidad lineal de los datos de entrada; por ejemplo, el perceptrón requiere separabilidad perfecta para garantizar convergencia. Por otro lado, existen algoritmos que suponen que la falta de separabilidad lineal se debe a ruido; por ejemplo, la regresión logística.

Sin embargo, si tenemos un problema no lineal, muy comunes en aplicaciones reales, las técnicas de transformación lineal para reducir la dimensión (PCA, LDA) pueden no ser una buena opción. En esta sección se presenta una versión *kernelizada* (por núcleos) de PCA: ***Kernel Principal Component Analysis*** (***KPCA***) para transformar datos que no son linealmente separables en un subespacio nuevo de menor dimensión que sea adecuado para clasificadores lineales.

Funciones de núcleos

Cuando tenemos problemas no lineales, se pueden abordar proyectándolos en un espacio de características de dimensión superior en el que las clases son linealmente separables. Para transformar las muestras $\mathbf{x} \in \mathbb{R}^d$ sobre un espacio superior $k-$dimensional, debe definirse una función de mapeo ϕ:

$$\phi : \mathbb{R}^d \rightarrow \mathbb{R}^k;\ (k \gg d)$$

ϕ puede verse como una función que genera combinaciones no lineales de las características originales del conjunto de datos $d-$dimensional original sobre un espacio de características $k-$dimensional. Por ejemplo, para un vector de dos dimensiones $\mathbf{x} \in \mathbb{R}^2$ un mapeo potencial al espacio de 3 dimensiones $\mathbb{R}^3$ puede ser:

$$\mathbf{x} = [x_1, x_2]^T$$

$$\downarrow$$
$$\phi$$
$$\downarrow$$

$$\mathbf{z} = \left[x_1^2, \sqrt{2x_1x_2}, x_2^2\right]^T$$

De esta forma, se puede utilizar PCA en un espacio de orden superior para después proyectarlo sobre un espacio de dimensión menor en el que las muestras puedan separarse con un clasificador lineal.

Para implementar un KPCA con *función de base radial* (***Radial Basis Function*** **RBF** o Gaussiano) como kernel, se realizan los siguientes pasos:

Algoritmo 4 *Kernel PCA*

1. Obtener la matriz de núcleos $\mathbf{K}$, donde debe calcularse: $\mathtt{k}\left(x^{(i)}, x^{(j)}\right) = \exp\left(-\gamma \left\|x^{(i)} - x^{(j)}\right\|^2\right)$; $\gamma = \dfrac{1}{2\sigma}$ para cada par de muestras:
$$\mathbf{K} = \begin{bmatrix} \mathtt{k}\left(x^{(1)}, x^{(1)}\right) & \mathtt{k}\left(x^{(1)}, x^{(2)}\right) & \cdots & \mathtt{k}\left(x^{(1)}, x^{(n)}\right) \\ \mathtt{k}\left(x^{(2)}, x^{(1)}\right) & \mathtt{k}\left(x^{(2)}, x^{(2)}\right) & \cdots & \mathtt{k}\left(x^{(2)}, x^{(n)}\right) \\ \vdots & \vdots & \ddots & \vdots \\ \mathtt{k}\left(x^{(n)}, x^{(1)}\right) & \mathtt{k}\left(x^{(n)}, x^{(2)}\right) & \cdots & \mathtt{k}\left(x^{(n)}, x^{(n)}\right) \end{bmatrix}$$
Es decir, si tenemos 100 muestras de entrenamiento, su matriz de núcleos sería de dimensión 100×100
2. Centrar la matriz $\mathbf{K}$ utilizando: $\mathbf{K}' = \mathbf{K} - 1_n\mathbf{K} - \mathbf{K}1_n + 1_n\mathbf{K}1_n$; donde 1_n es una matriz $n \times n-$dimensional en la que todos los valores son $\dfrac{1}{n}$
3. Seleccionar los k eigenvectores de la matriz centrada que corresponden con los k mayores eigenvalores

El centrado de la matriz (paso 2) se requiere porque no es posible garantizar que el nuevo espacio esté centrado en cero. Otras funciones de núcleo comúnmente usadas son el *kernel polinomial* y el *kernel tangente hiperbólico* (sigmoide).

Ejemplo: Separando medias lunas

Probando el código con conjuntos de datos no lineales; primero creando un *dataset* bidimensional de 100 muestras con forma de media luna:

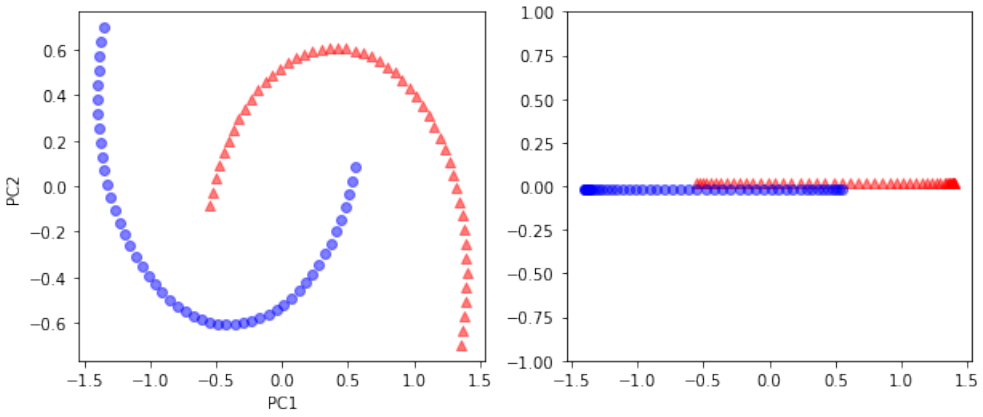
```
# Bibliotecas
import numpy as np
import matplotlib.pyplot as plt
```

```
# Datos: medias lunas
from sklearn.datasets import make_moons
X, y = make_moons(n_samples=100, random_state=123)
plt.scatter(X[y==0, 0], X[y==0, 1], color='red', marker='^', alpha=0.5)
plt.scatter(X[y==1, 0], X[y==1, 1], color='blue', marker='o', alpha=0.5)
plt.show()
```

Primero tratemos de separar los conjuntos usando PCA:

```
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)
```

```
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(10,4))
ax[0].scatter(X_pca[y==0, 0], X_pca[y==0, 1], color='red',
              marker='^', alpha=0.5)
ax[0].scatter(X_pca[y==1, 0], X_pca[y==1, 1], color='blue',
              marker='o', alpha=0.5)
ax[1].scatter(X_pca[y==0, 0], np.zeros((50,1))+0.02, color='red',
              marker='^', alpha=0.5)
ax[1].scatter(X_pca[y==1, 0], np.zeros((50,1))-0.02, color='blue',
              marker='o', alpha=0.5)
ax[0].set_xlabel('PC1')
ax[0].set_ylabel('PC2')
ax[1].set_ylim([-1,1])
ax[0].set_xlabel('PC1')
plt.show()
```



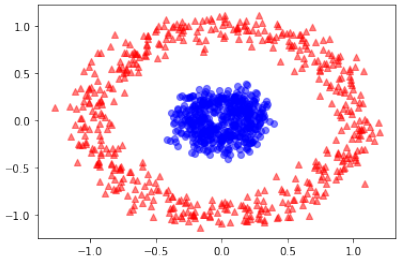
Transformó los datos a un conjunto linealmente separable. Por supuesto que existe *KernelPCA* implementado en *sklearn*:

```
from sklearn.decomposition import KernelPCA
kpca = KernelPCA(n_components=2, kernel='rbf', gamma=15)
X_kpca = kpca.fit_transform(X)
# se puede graficar con el mismo código del anterior
```

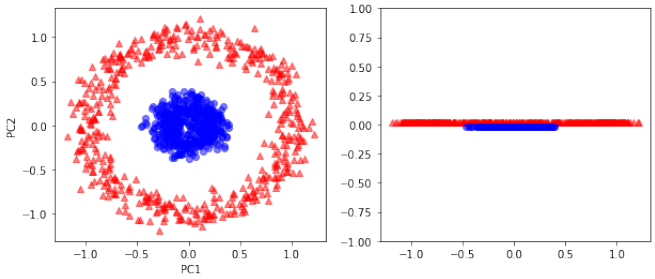
Tarea

Otro conjunto interesante para probar *rbf_kpca* son círculos concéntricos; *sklearn* incluye un generador de dicho conjunto:

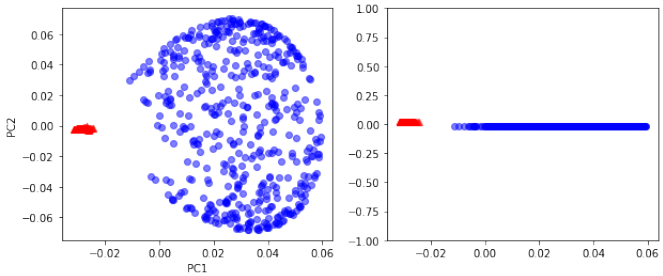
```
# Datos: círculos concéntricos
from sklearn.datasets import make_circles
X, y = make_circles(n_samples=1000, random_state=123, noise=0.1, factor=0.2)
```



1. Probar que PCA no puede separar este conjunto:



2. Mostrar que tanto *rbf_kpca* (propio) como KernelPCA de *sklearn* separan el conjunto de forma correcta:



3. Probar con una función de *kernel* diferente sobre el conjunto de medias lunas y el conjunto de círculos concéntricos.

*