Machine Learning

18 de septiembre de 2020

Parte I
Estadística

1. Tratamiento de datos

1.1. Escalamiento

Muchas veces los datos vienen en rangos fuera de $[0, 1]$ por tal motivo es necesario hacer un escalamiento antes de ingresar los datos a la red neuronal. Para realizar esto, se utiliza la siguiente ecuación:

$$\frac{x_{0i} - x_{0min}}{x_{0max} - x_{0min}} = \frac{x_i - x_{min}}{x_{max} - x_{min}}$$

$$\boxed{x_{0i} = \left(\frac{x_{0max} - x_{0min}}{x_{max} - x_{min}}\right) \cdot (x_i - x_{min}) + x_{0min}}$$

- Donde para nuestro caso $[x_{0min}, x_{0max}] = [0, 1]$, por lo tanto la ecuación queda como:

$$x_{0i} = \left(\frac{1}{x_{max} - x_{min}}\right) \cdot (x_i - x_{min}) + 0$$

$$\boxed{x_{0i} = \frac{x_i - x_{min}}{x_{max} - x_{min}}}$$

Una vez que la red neuronal no ha arrojado un resultado, hay que regresar dichos datos al rango original, para ello simplemente hay que despejar x_i en ves de x_{0i}, quedando la ecuación como:

$$\boxed{x_i = \left(\frac{x_{max} - x_{min}}{x_{0max} - x_{0min}}\right) \cdot (x_{0i} - x_{0min}) + x_{min}}$$

- Donde para nuestro caso $[x_{0min}, x_{0max}] = [0, 1]$, tenemos:

$$\boxed{x_i = \frac{x_{0i} - x_{0min}}{x_{0max} - x_{0min}}}$$

1.2. Matriz de datos

Supongamos que sobre los individuos $w_1, ..., w_n$ se han observado las variables $X_1, ..., X_p$. Sea $x_{ij} = X_j(w_i)$la observación de la variable X_j sobre el individuo w_j. La matriz de datos multivariable es:

$$\mathbf{X} = \begin{bmatrix} x_{11} & \cdots & x_{1j} & \cdots & x_{1p} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ x_{i1} & \cdots & x_{ij} & \cdots & x_{ip} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ x_{n1} & \cdots & x_{nj} & \cdots & x_{np} \end{bmatrix}$$

Las filas de $\mathbf{X}$se identifican con los individuos y las columnas de $\mathbf{X}$ con las variables indicaremos:

- x_i la fila i-ésima de $\mathbf{X}$, que operaremos como un vestor columna.
- X_j la columna j-ésima de $\mathbf{X}$.

1.3. Variables compuestas

Algunos métodos de Analisis Multivariable (AM), consisten en obtener e interpretar combinaciones lineales adecuadas de las variables observables. Una variable compuesta Y es una combinación lineal de las variables observables con coeficientes $\mathbf{a} = [a_1, ..., a_p]'$

$$Y = a_1 \cdot X_1 + ... + a_p \cdot X_p$$

Si $\mathbf{X} = [X_1, ..., X_p]$ es la matriz de datos, también podemos escribir

$$Y = \mathbf{X} \cdot \mathbf{a}$$

Si $\mathbf{Z} = b_1 \cdot X_1 + ... + b_p \cdot X_p = \mathbf{X} \cdot b$ es otra variable compuesta, se verifica:

- $\bar{Y} = \bar{\mathbf{x}}' \cdot \mathbf{a}$, $\bar{Z} = \bar{\mathbf{x}}' \cdot \mathbf{b}$
- $var(Y) = \mathbf{a}' \cdot \mathbf{S} \cdot \mathbf{a}$, $var(Z) = \mathbf{b}' \cdot \mathbf{S} \cdot \mathbf{b}$
- $cov(Y, Z) = \mathbf{a}' \cdot \mathbf{S} \cdot \mathbf{b}$

1.4. Matriz de Centrado "H"

Definida por:

$$\boxed{H = I - \frac{1}{n} \cdot \bar{1}_{n \times n}}$$

Donde:

- I : Matriz identidad nxn
- $\bar{1}_{n \times n}$: ;Matriz de unos de dimensión nxn

Se usa para centrar configuración es datos:

Sea «X» una matriz de datos, de dimensión «nxp» entonces "$H \times X$" es una matriz cuyas columnas tienen MEDIA CERO

PROPIEDADES

- H es idempotente, es decir: $H \times H = H$
- $rang(H) = traza(H) = n - 1$

1.5. Media Muestral "$\bar{X}$"

Sea "X" una matriz de «n» datos por «p» variables de interes, entonces la media muestral de cada una de estas variables esta dada por:

$$\boxed{\bar{X} = \frac{1}{n} \cdot X' \cdot \bar{1}_n}$$

Donde $\bar{1}_n$es un vector de unos de dimensión $n \times 1$

1.6. Matriz de Varianza - Covarianza "S"

$$\boxed{S = \frac{1}{n} \cdot X' \cdot H \cdot X}$$

1.6.1. Matriz "D"

Donde "D" es una matriz obtenida a partir de los componentes de la diagonal de la matriz "S" y esta dada por :

$$D = \begin{cases} d_{ij} = s_{ij} & \text{,Para } i = j \\ d_{ij} = 0 & \text{,Para } i \neq j \end{cases}$$

Una matriz muy utilizada es la matriz $D^{-1/2}$, los comanados para implementarla en matlab son los siguientes:

Algoritmo 1 Codigo en MatLab para calcular la matriz $D^{-1/2}$

```
% Sea "X" la matriz de datos de dimensión nxp
S = X'*H*X/n;
D1_2 = inv(diag(sqrt(diag(S))); % Matriz D^(-1/2)
```

1.7. Matriz de Correlación "R"

$$\boxed{R = D^{-1/2} \cdot S \cdot D^{-1/2}}$$

1.8. Matriz de Datos Estandarizados X_0

$$\boxed{X_0 = H \cdot X \cdot D^{-1/2}}$$

Donde:

- X : Es la matriz de datos, de dimensión nxp
- H : Es la matriz de centrado de dimensión nxn
- $D^{-1/2}$: Esta dada por$D^{-1/2} = diag(S)^{-1/2}$
- X_0: Matriz de datos estandarizados, es decir $(\mu = 0, \sigma = 1)$

1.9. Distancia de Mahalanobis

Es una forma de determinar la similitud entre dos variables aleatorias «multidimensionales». Se diferencia de la distancia euclídea en que tiene en cuenta la correlación entre las variables aleatorias.

Formalmente, la distancia de Mahalanobis entre dos variables aleatorias con la **misma distribución de probabilidad** X **y** Y con matriz de covarianza $\mathbf{\Sigma}$ se define como:

$$\boxed{d_m(X,Y) = \sqrt{(X-Y)' \cdot \Sigma^{-1} \cdot (X-Y)}}$$

A continuación se muestra la función implementada en MatLab:

Algoritmo 2 Función de Mahalanobis

```
% La funcion D=maha(X) calcula una matriz de cuadrados de distancias. El elemento (i,j)
% de la matriz D contiene el cuadrado d ela distancia de Mahalanobis entre la fila "i"
% y la fila "j" d ela matriz X.
% Entradas: una matriz X de dimension nxp.
% Salidas: una matriz D de dimension nxn.
%
function D = maha(X)
[n,p] = size(X);
% calculo del vector de medias y de la matriz de covarianzas de X:
S = cov(X,1);
% calculo de las distancias de Mahalanobis (al cuadrado):
D = zeros(n);
invS=inv(S);
for i = 1:n
for j = i+1:n
D(i,j) = (X(i,:)-X(j,:))*invS*(X(i,:)-X(j,:))';
end
end
D = D+D';
end
```

1.10. Analisis de Componentes Principales (ACP)

Muchas veces los datos son demasiado extensos y muchos de los cuales no son significantes y si los incluimos como entradas en la red neuronal, estos incrementaran el calculo computacional y ocacionaran que la red demore en aprender o incluso que jamás logre aprender, para ello se realiza un ACP, que nos permite simplificar los datos con la menor cantidad de perdida de información.

El ACP parte de una matriz de datos (centrada, es decir media igual a cero) de «n» filas y «p» columnas, que pued considerarse como una muestra de tamaño «n» de un vector aleatorio de dimensión «p»

$$X = [X_1, X_2, ..., X_p]'$$

Se considera la combinación lineal (univariante) de X

$$Y = X't$$

Donde "t" es un vector de pesos de dimensión «p». La primera componente principal aparece como la solución al problema de encontral el vector "t" que maximiza la varainza de "Y" con la condición de normalización $t' \cdot t = 1$. En otras palabras, la expresión $Var(Y)$ en función del vector de pesos "t" da lugar a un problema

variacional que viene por la solución de la primera componente principal. Este problema equivale a encontrar los autovalores y autovectores d ela matriz de covarianzas de "X"

Parte II
Creación de gráficas y visualizaciones

https://nbviewer.jupyter.org/github/GoogleCloudPlatform/training-data-analyst/blob/master/courses/machine_learnir
https://machinelearningmastery.com/test-time-augmentation-with-scikit-learn/
https://colab.research.google.com/drive/1A3TALnuDj0FXqvSBM96Es6BEKaZwbkVz?usp=sharing&fbclid=IwAR2ysx4
https://planetachatbot.com/aprendizaje-no-supervisado-para-multiples-clases-en-python-407cb6be878c

Parte III
Machine Learning

2. Regresión Lineal Simple

En Scikit-Learn cada clase del modelo es representado por una clase Python. Si queremos calcular una regresió lineal simple, nosotros importamos la clase "Regresión Linear"

Algoritmo 3 Generación de datos

```
from sklearn.linear_model import LinearRegression
import matplotlib.pyplot as plt
import numpy as np

#Semilla para generar los mismos numeros aleatorios
rng = np.random.RandomState(42)
x = 10 * rng.rand(50)
y = 2 * x - 1 + rng.randn(50)
plt.scatter
```

La ecuación que se trata de encontrar es:

$$f(x) := y = 2 \cdot x + 1$$

Claramente tiene una intersección en el eje y=1, por tal motivo usamos "fit_intercept=True"

Algoritmo 4 Entrenando el modelo lineal

```
# Instanciando un objeto modelo de la clase "LinealRegression"
model =LinealRegression(fit_intercept=True)

# Convirtiendo los ventores en matrices
X=x[: , np.newaxis]

# Entrenando el modelo con los datos
model.fit(X,y)

# Obteniendo los coeficientes y = A*x + B
A = model.coef_
B = model.intercept_
```

Después de tener el modelo lineal entrenado, se procederá a predecir nuevos valores usando el modelo previamente entrenado, tal como se muestra acontinuación:

Algoritmo 5 Prediciendo nuevos valores con el modelo lineal

```
# Generando nuevos datos
xfit = np.linspace(-1,11)
# Convirtiendo los datos en matriz
Xfit = xfit[:, np.newaxis]
# Calculando los nuevos valores con el modelo entrenado
yfit = model.predict(Xfit)
```

3. Regresión Polinomial

Tomando un modelo multidimencional lineal de la forma:

$$y = a_0 + a_1 \cdot x_1 + a_2 \cdot x_2 + a_3 \cdot x_3 + \ldots$$

Donde los coeficientes x_n están determinados por una función que transforma los datos $f_n(x) = x^n$, nuestro modelo se convierte en una regresión polinomial:

$$y = a_0 + a_1 \cdot x + a_2 \cdot x^2 + a_3 \cdot x^3 + \ldots$$

Denotando que esto es todavía un modelo lineal, haciendo referente a encontrar los coeficientes a_n.

3.1. Funciones Polinomiales Básicas

Esta proyección polinomial es muy util y Scikit-Learn viene integrada con una funcionalidad para cacular los x^n, para ello se usara **PolinomialFeatures** transformer:

Algoritmo 6 Usando "PolinomialFeatures"

```
from sklearn.preprocessing import PolynomialFeatures
x = np.array([2, 3, 4])
poly = PolynomialFeatures(3, include_bias=False)
poly.fit_transform(x[:,None])
```

La salida mostrada en pantalla será

```
array([[ 2.,  4.,  8.],
       [ 3.,  9., 27.],
       [ 4., 16., 64.]])
```

Esta transformación a convertido un arreglo unidimensional en un arreglo tridimensional, con este nuevo arreglo multidimensional, podemos realizar una regresión lineal. La mejor forma de hacer esto es usar un pipeline.

Haciendo que el modelo polinomial tenga 7 grados:

Algoritmo 7 Instanciando modelo polinomial de 7 grados

```
from sklearn.pipeline import make_pipeline
degree = 7
poly_model = make_pipeline( PolynomialFeatures(degree), LinealRegression() )
```

Con esta transformación, podemos construir un modelo mucho mas complejo que relacione "x" e "y". Por ejemplo, modelar una onda seno con ruido ξ :

$$y_{real} = \sin(x_{real}) + \xi$$

Algoritmo 8 Modelado polinomial de una función senoidal

```
rng = np.random.RandomState(1)
# Generando 50 datos aleatorios desde 0 a 10.
x = 10 * rng.rand(50)
y = np.sin(x) + 0.1 * rng.randn(50)
poly_model.fit(x[:, np.newaxis], y)
xfit = np.linspace(0, 10, 1000)
yfit = poly_model.predict( xfit[:, np.newaxis])
plt.scatter(x,y)
plt.plot(xfit, yfit)
```

4. Regresión con Funciones Básicas Gausianas

Otra forma de modelar los datos es usar una suma de funciones Gausianas como base.

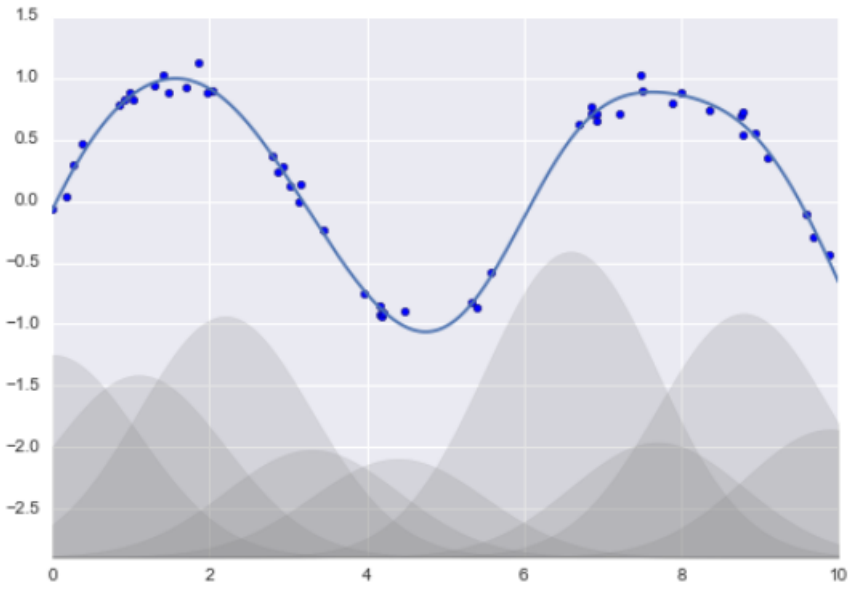


Figura 1: Modelado a partir de funciones Gausianas

Las funciones Gaussianas basicas no estan construidas en Scikit-Learn, poro podemos escribir una transformador que los crea, tal como se muestra a continuación:

Algoritmo 9 Clase Gausiana

```
from sklearn.base import BaseEstimator, TransformerMixin

class GaussianFeatures(BaseEstimator, TransformerMixin):
    """Uniformly spaced Gaussian features for one-dimensional input"""
    def __init__(self, N, width_factor=2.0):
        self.N = N
        self.width_factor = width_factor
         @staticmethod
    def _gauss_basis(x, y, width, axis=None):
        arg = (x - y) / width
        return np.exp(-0.5 * np.sum(arg ** 2, axis))

    def fit(self, X, y=None):
        # create N centers spread along the data range
        self.centers_ = np.linspace(X.min(), X.max(), self.N)
        self.width_ = self.width_factor * (self.centers_[1] - self.centers_[0])
        return self

    def transform(self, X):
        return self._gauss_basis(X[:, :, np.newaxis],
                                 self.centers_,
                                 self.width_, axis=1)

gauss_model = make_pipeline(GaussianFeatures(20),LinearRegression())
gauss_model.fit(x[:, np.newaxis], y)
yfit = gauss_model.predict(xfit[:, np.newaxis])

plt.scatter(x, y)
plt.plot(xfit, yfit)
plt.xlim(0, 10)
```

A continuación se muestra el modelo ajustado con los datos de entrada, como se podra observar, no se ajusta mucho a una onda senoidal.

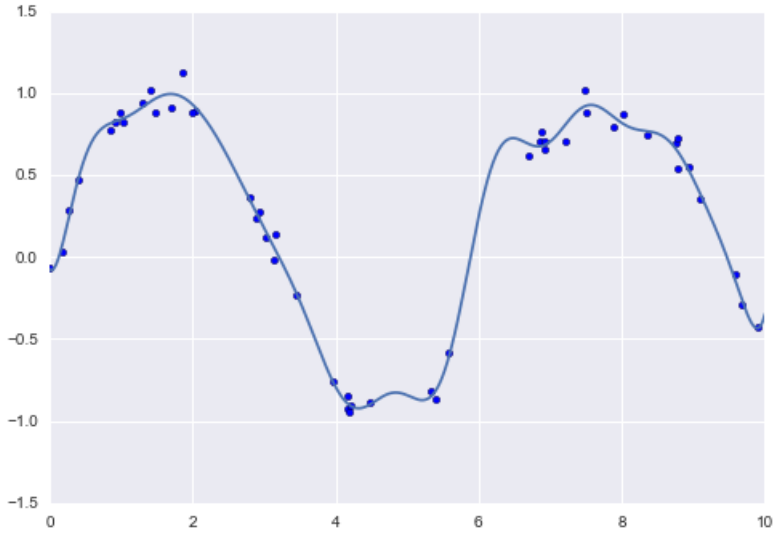


Figura 2: Modelado con funciones Gausianas

5. Regresión Ridge o Regularización L_2

También llamada Tikhonov regularización. Este tipo de regresión usa la penalización de la suma de cuadrados (2-norms) del modelo de coeficientes, en este caso el modelo es:

$$P = \alpha \cdot \sum_{n=1}^{N} \theta_n^2$$

Donde α es un parametro libre que controla la fuerza de la penalidad, este modelo ya viene implementado en Scikit-Learn con «Ridge»:

Algoritmo 10 Uso de «Ridge» en «Scikit-Learn»

```
from sklearn.lineal_model import Ridge
model = make_pipeline(GaussianFeatures(30), Ridge(aplha=0.1))

fig, ax = plt.subplots(2, sharex=True)
model.fit(x[:,np.newaxis],y)
ax[0].scatter(x,y)
ax[0].plot(xfit, model.predict(xfit[:,np.newaxis]))
ax[0].set(xlabel='x', ylabel='y', ylim=(-1.5,1.5))
ax[0].set_title(title)
ax[1].plot(model.steps[0][1].centers_,
           model.steps[1][1].coef_)
ax[1].set(xlabel='basis location',
          ylabel='coefficient',
          xlim=(0,10))
```

El modelo ajusta mucho mejor la onda senoidal, también podemos observar claramente como los coefientes varían para mejorar el ajuste.

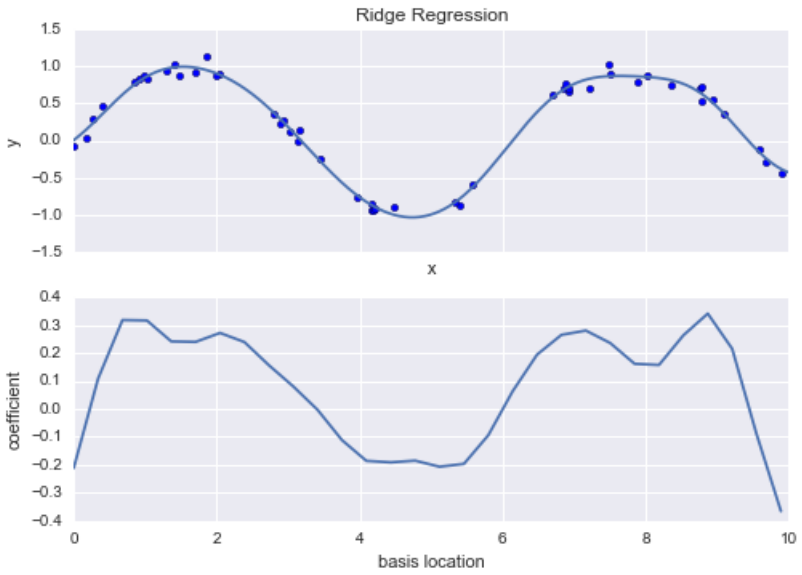


Figura 3: Regresión Ridge

-

6. Regressión Lasso o Regularización L_1

Otra común regularización es conocidad como «Lasso», este se basa en penalizar la suma de los valores absolutos (1–norms) de la regresión de los coeficientes.

$$P = \alpha \cdot \sum_{n=1}^{N} \mid \theta_n \mid$$

Algoritmo 11 Regressión Lasso

```
from sklearn.linear_model import Lasso
model = make_pipeline(GaussianFeatures(30), Lasso(alpha=0.001))
basis_plot(model, title='Lasso Regression')
```

Esta regresión tiende a favorecer modelos dispersos cuando sea posible: es decir, establece preferentemente los coeficientes del modelo exactamente en cero.

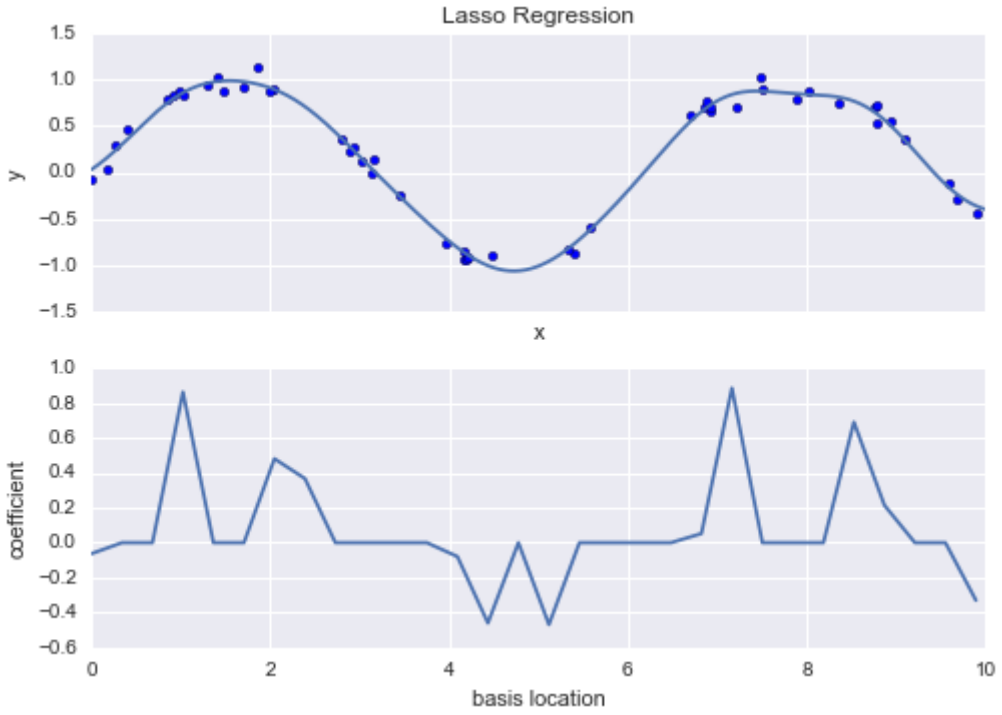


Figura 4: Modelado de onda senoidal usando Lasso

7. Support Vector Machines (SVMs)

Este tipo de modelado es particularmente poderoso y flexible algoritmos de clasificación supervisado.

Algoritmo 12 Generando datos

```
from sklearn.datasets.samples_generator import make_blobs
X,y = make_blobs(n_samples=50, centers=2,
                 random_state=0, cluster_std=0.60)
plt.scatter( X[:,0], X[:,1], c=y, s=50, cmap='autumn')
```

Un clasificador linear debería intentar dibujar una linea que separe los dos tipos de datos. Support vector machines ofrece una forma dibujar lineas que separen los datos, permitiendo clasificarlos por clases.

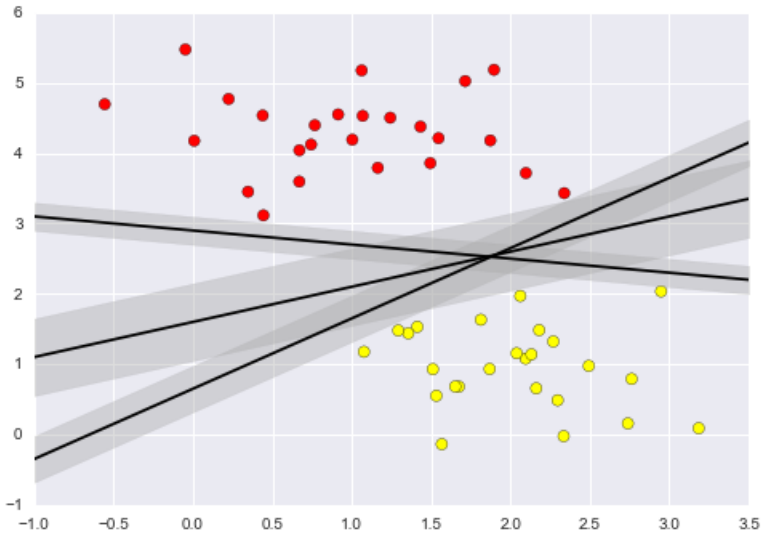


Figura 5: SVM para clasificación

A continuación entrenaremos el SVM de Scikit-Learn con esos datos, notando que el paraemtro C=1E10 será explicado más adelante.

Algoritmo 13 Entrenando el modelo SVM

```
from sklearn.svm import SVC
model = SVC( kernel='linear', C=1E10 )
model.fit(X,y)
```

Para una mejor visualización, se agregara los limites de decisión del SVM, se implementa una función «plot_svc_decision_function»:

Algoritmo 14 Implementación «plot_svc_decision_function»

```
def plot_svc_decision_function(model, ax=None, plot_support=True):
    """Plot the decision function for a 2D SVC"""
    if ax is None:
        ax = plt.gca()
    xlim = ax.get_xlim()
    ylim = ax.get_ylim()
    # create grid to evaluate model
    x = np.linspace(xlim[0], xlim[1], 30)
    y = np.linspace(ylim[0], ylim[1], 30)
    Y, X = np.meshgrid(y, x)
    xy = np.vstack([X.ravel(), Y.ravel()]).T
    P = model.decision_function(xy).reshape(X.shape)
    # plot decision boundary and margins
    ax.contour(X, Y, P, colors='k',
               levels=[-1, 0, 1], alpha=0.5,
               linestyles=['--', '-', '--'])
    # plot support vectors
    if plot_support:
        ax.scatter(model.support_vectors_[:, 0],
                   model.support_vectors_[:, 1],
                   s=300, linewidth=1, facecolors='none');
    ax.set_xlim(xlim)
    ax.set_ylim(ylim)
```

Con tan solo dos líneas de código ahora podremos visualizar nuestros resultados:

Algoritmo 15 Visualización del modelo SVM

```
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
plot_svc_decision_function(model);
```

Los gráficos son los siguientes:

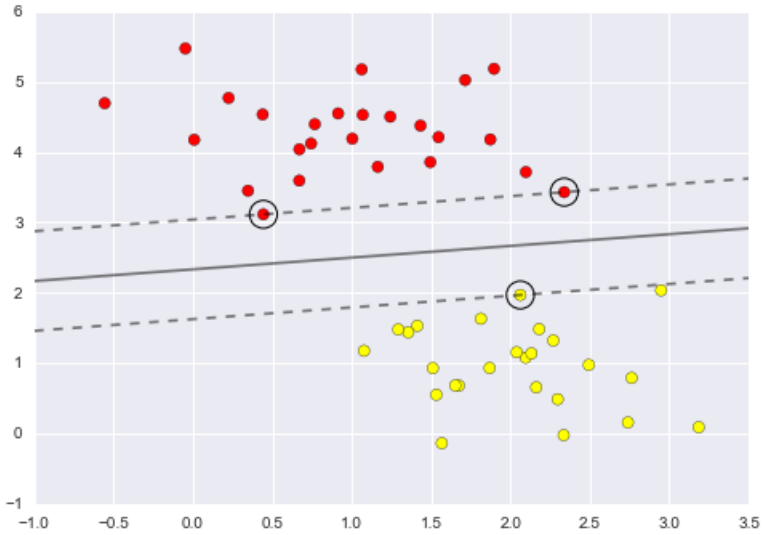


Figura 6: Limites del modelo SVM entrenado.

La validación del modelo se realizar con 60 y 120 puntos nuevos generados de forma aleatoria, todo esto definido en una función «plot_svm»:

Algoritmo 16 Validación del model SVM

```
def plot_svm(N=10, ax=None):
    X, y = make_blobs(n_samples=200, centers=2,
                      random_state=0, cluster_std=0.60)
    X = X[:N]
    y = y[:N]
    model = SVC(kernel='linear', C=1E10)
    model.fit(X, y)
    ax = ax or plt.gca()
    ax.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
    ax.set_xlim(-1, 4)
    ax.set_ylim(-1, 6)
    plot_svc_decision_function(model, ax)
fig, ax = plt.subplots(1, 2, figsize=(16, 6))
fig.subplots_adjust(left=0.0625, right=0.95, wspace=0.1)
for axi, N in zip(ax, [60, 120]):
    plot_svm(N, axi)
    axi.set_title('N = {0}'.format(N))
```

En los gráficos motrados se observa que el modelo SVM clasifica de forma muy aceptable los datos de validación generados.

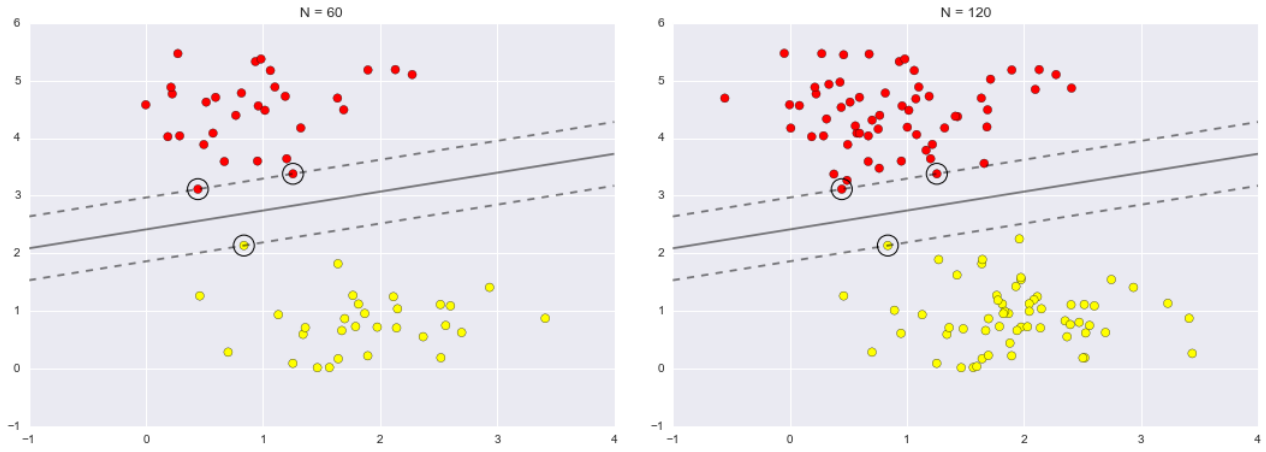


Figura 7: Validación del modelo SVM

8. Clasificador Kernel-SVM

Al mezclar SVM con Kernel, pudiendo ser «regresiones lineales», obtenemos un modelado muy poderoso, sobretodo para datos con elevada dimensionalidad.

Algoritmo 17 Entrenando un modelo Kernel-SVM

```
from sklearn.datasets.samples_generator import make_circles
from sklearn.svm import SVC
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

X, y = make_circles(100, factor=.1, noise=.1)

clf = SVC(kernel='linear').fit(X,y)

plt.scatter(X[:,0], X[:,1], c=y, s=50, cmap='autumn')
plot_svc_decision_function(clf, plot_support=False)
```

Tal como se observa en la visualización los datos no son linealmente separables.

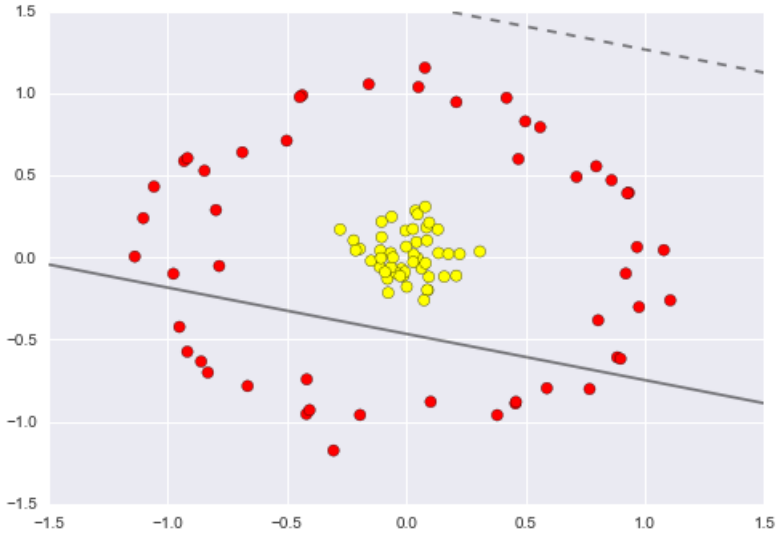


Figura 8: Clasificación de datos usando un «Kernel Lineal»

Agregando una dimensión adicional a los datos, clasificarlos se combierte en un problema lineal, para ello aplicaremos la transformación $r = f(x)$ definida como:

$$r = e^{-x^2+1}$$

Después de aplicar la transformación y agregar la nueva coordenada, vamos dibujar usando 3 dimensiones

Algoritmo 18 Conversion de datos 2D a 3D.

```
from ipywidgets import interact, fixed
from mpl_toolkits import mplot3d

def plot_3D(X, y, z, elev=None, azim=None):
    ax = plt.subplot(projection='3d')
    ax.scatter3D( X[:,0], X[:,1], z, c=y, s=50, cmap='autumn')
    ax.view_init(elev=elev, azim=azim)
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_zlabel('z')
    plt.show()

r = np.exp(-(X ** 2).sum(1))
plot_3D( X, y, r)
```

La gráfica generada se muestra a continuación:

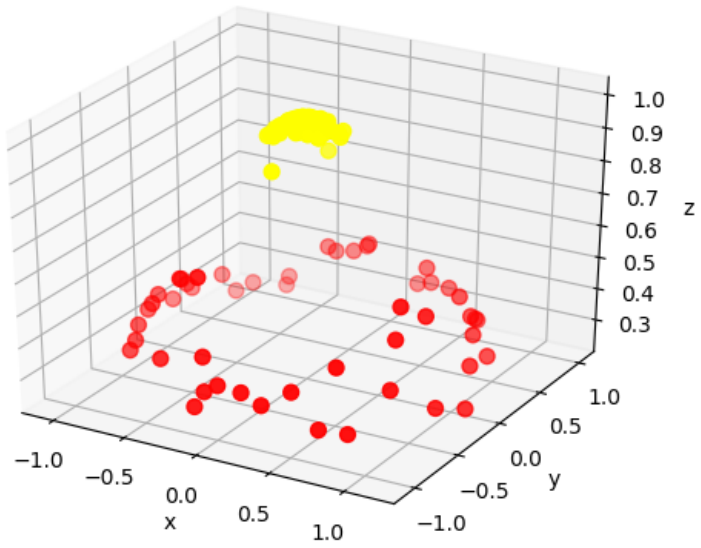


Figura 9: Visualización de los datos usando Kernel-SVM

En esta estrategia, es computacionalmente costoso, ya que X tiene la forma $n \times n$. Sin embargo, con un truco en el núcleo, es posible hacer un ajuste de los datos transformados por el núcleo de forma implícita, es decir, ¡sin countruir nunca l arepresentación N-dimensional completa de la proyección del núcleo!. Este truco esta integrado en el núcleo de SVM y es una de las razones por las que es poderoso.

En Scikit-Learn, podemos usar el «kernel-SVM» cambiando de «linear» a «rbf» (radial basis function):

Algoritmo 19 Usando «Radial Basis Fuction» Kernel-SVM

```
# Using a kernel Radial Basis Function
clf = SVC(kernel='rbf', C=1E6)
clf.fit(X,y)

plt.scatter(X[:,0], X[:,1], c=y, s=50, cmap='autumn')
plot_svc_decision_function(clf)
plt.scatter(clf.support_vectors_[:,0], clf.support_vectors_[:,1],
            s=300, lw=1, facecolors='none')
```

Usando el «Kernel-SVM», podemos transformar datos en el mismo interior de los algoritmos para ejecutar metodos no lineales de forma más rápido, especialmente para modelos en los que se puede aplicar este «truco».

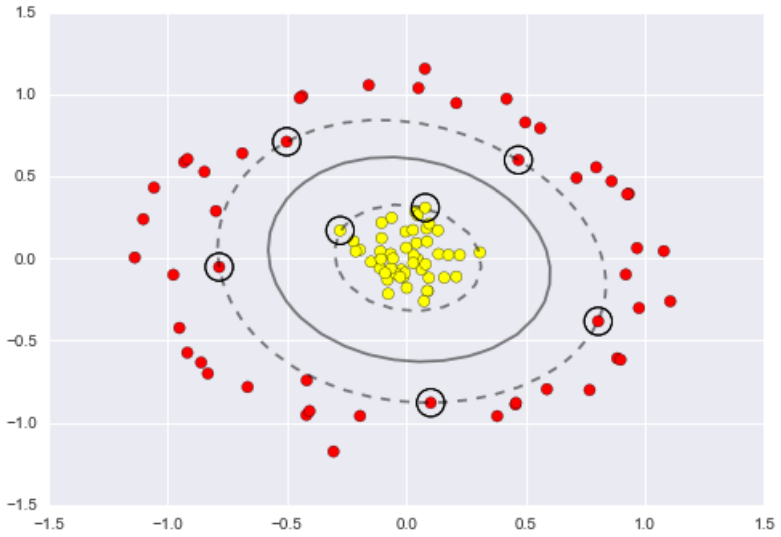


Figura 10: Clasificación de datos usando «Kernel RDF-SVM»

9. Clasificador SVM con margenes suavizados

Algoritmo 20 Datos solapados

```
X, y = make_blobs(n_samples=100, centers=2,
                  random_state=0, cluster_std=1.2)
plt.scatter(X[:,0], X[:,1], c=y, s=50, cmap='autumn')
plt.show()
```

Tal como se observa a continuación los datos se solapan, esto podría hacer la clasificación más dificil.

Figura 11: Visualización de datos solapados

Para manejar este tipo de casos, la implementación de SVM tiene un «fudge-factor» que suavisa los margenes, la dureza de los margenes es ajustado por un parámetro C. Para un «C» muy grande, el margen es duro, y los puntos no pueden estar en él. Para un «C» más pequeño el margen es más suave y puede crecer hasta abarcar algunos puntos.

Algoritmo 21 Clasificación de datos con diferentes valores de suavizado.

```
# Creating data: 100 samples with 2 centers
X, y = make_blobs(n_samples=100, centers=2,
                  random_state=0, cluster_std=0.8)

fig, ax = plt.subplots(1, 2, figsize=(16,6))
fig.subplots_adjust(left=0.0625, right=0.95, wspace=0.1)

for axi, C in zip(ax, [10.0, 0.1]):
    model = SVC(kernel='linear', C=C).fit(X,y)
    axi.scatter(X[:,0], X[:,1], c=y, s=50, cmap='autumn')
    plot_svc_decision_function(model,axi)
    axi.scatter(model.support_vectors_[:,0],
                model.support_vectors_[:,1],
                s=300, lw=1, facecolors='none')
    axi.set_title('c = {0:.1f}'.format(C), size=14)

plt.show()
```

El valor óptimo para el parametro «C» depende del dataset, por tanto debería ser ajustado usando «cross-validation» o un procedimiento similar.

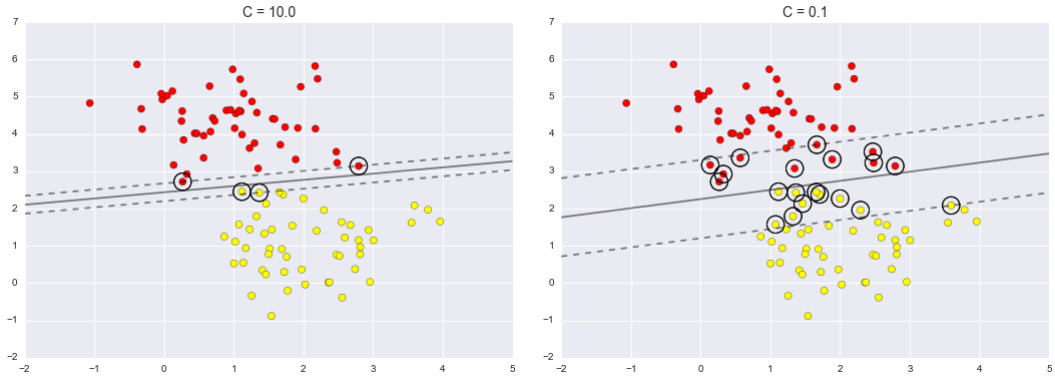


Figura 12: SVC con margenes suavizados (C=10.0 & C=0.1).

10. Face Recognition usando SVM

Usaremos rostros previamente clasificados, el cual consiste en miles de fotos de varias figuras públicas. Un buscador para estos datos esta integrada en Scikit-Learn:

Algoritmo 22 Fotos de rostros clasificados.

```
import matplotlib.pyplot as plt
from sklearn.datasets import fetchs_lfw_people

faces = fetch_lfw_people(min_faces_per_person=60)
print(faces.target_names)
print(faces.images.shape)
```

Esta base de datos cargada consta de rostros de personajes como: «Ariel Sharon», «Colin Powell», «Donald Rumsfeld», «George W Bush», «Gerald Schoroeder», «Hugo Chavez», «Junichiro Koizumi» y «Tony Blair».

El tamaño de este dataset consta de 1348 muestras con imagenes de «62x47»

A continuación se mostrará algunas imagenes de los rostros del dataset cargado.

Algoritmo 23 Mostrando algunos rostros

```
fig, ax = plt.subplots(3,5)
for i, axi in enumerate(ax.flat):
    axi.imshow(faces.images[i], cmap='bone')
    axi.set(xticks=[], yticks=[],
            xlabel=faces.target_names[faces.target[i]])
```

Figura 13: Algunos rostros del dataset.

Cada imagen contiene [62x47] cerca de 3000 Pixeles. Podríamos considerar cada valor del pixel como una característica pero a menudo es más efectivo usar algun preprocesador para extraer las características más significativas, aquí usaremos el «Análisis de Componentes Principales» para extraer 150 componentes principales e introducirlas al clasificador SVM. Podemos hacer esto de manera directa empaquetando el procesador y clasificador en un único «pipeline»:

Algoritmo 24 Empaquetando PCA y SVM en un pipeline

```
from sklearn.svm import SVC
from sklearn.decomposition import PCA as RandomizedPCA
from sklearn.pipeline import make_pipeline

pca = RandomizedPCA(n_components=150, whiten=True, random_state=42)
svc = SVC(kernel='rbf', class_weight='balanced')

model = make_pipeline(pca,svc)
```

Para validar nuestro modelo, dividiremos los datos en 2 conjuntos: «entrenamiento» y «prueba».

Algoritmo 25 Creando los conjuntos de datos «entrenamiento» y «prueba».

```
#Deprecated: cross_validation by model_selection
from sklearn.cross_validation import train_test_split

Xtrain, Xtest, ytrain, ytest = train_test_split(faces.data,
                                                faces.target,
                                                random_state=42)
```

Finalmente usaremos un «grid search cross-validation» para explorar las combinaciones de parametros. Aqui ajustaremos «C» (controlar la dureza de los margenes) y «gamma» (controlar el tamaño del núcleo de la funcion de base radial)

Algoritmo 26 Training model using grid search cross-validation

```
from sklearn.model_selection import GridSearchCV

param_grid = { 'svc__C'     : [    1,     5,    10,    50],
               'svc__gamma' : [.0001, .0005, .001, .005]}
grid = GridSearchCV(model, param_grid)
grid.fit(Xtrain, ytrain)

print(grid.best_params_)
```

Los mejores valores encontrados para los parametros son : $'svc\_\_c' : 10, 'svc\_\_gamma' : 0.001$.
Ahora usaremos este model para predecir valores con los datos de prueba.

Algoritmo 27 Validando el modelo

```
model = grid.best_estimator_
yfit = model.predict(Xtest)
```

Podemos obtener una mejor visión de nuestro clasificador generando un reporte, tal como se muestra a continuación:

Algoritmo 28 Reporte de clasificación

```
from sklearn.metrics import classification_report
print(classification_report(ytest, yfit,
                            target_names=faces.target_names))
```

Según se observa en el reporte, podemos

	precision	recall	f1-score	support
Ariel Sharon	0.65	0.73	0.69	15
Colin Powell	0.80	0.87	0.83	68
Donald Rumsfeld	0.74	0.84	0.79	31
George W Bush	0.92	0.83	0.88	126
Gerhard Schroeder	0.86	0.83	0.84	23
Hugo Chavez	0.93	0.70	0.80	20
Junichiro Koizumi	0.92	1.00	0.96	12
Tony Blair	0.85	0.95	0.90	42
accuracy			0.85	337
macro avg	0.83	0.84	0.84	337
weighted avg	0.86	0.85	0.85	337

Figura 14: Reporte de clasificación del modelo «PCA-SVM»

También podemos mostrar la matriz de confusión, para tener una mejor visualización de nuestro modelo.

Algoritmo 29 Matriz de Confusión

```
from sklearn.metrics import confusion_matrix
import seaborn as sns

mat = confusion_matrix(ytest, yfit)
sns.heatmap(mat.T, square=True, annot=True, fmt='d', cbar=False,
            xticklabels=faces.target_names,
            yticklabels=faces.target_names)

plt.xlabel('true label')
plt.ylabel('predicted label')
```

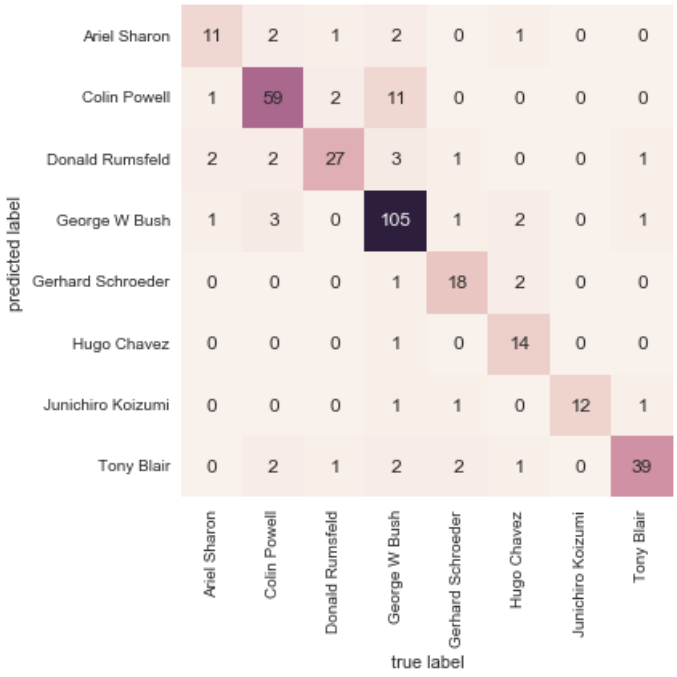


Figura 15: Matriz de confusión del clasificador de rostros.

Para este tipo de aplicaciones, es mejor usar OpenCV, que tiene implementado modelos para extraer caracteristicas de las imagenes independiente de la pixelación.

11. Clasificador Naive Bayes

Cargando los datos del "iris" (n_sampes x n_features)

Algoritmo 30 Cargando datos "iris"

```
import seaborn as sns
iris = sns.load_dataset('iris')
```

Una visualización parcial del dataframe se puede realizar ejecutando el comando "iris.head()"

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0,2	setosa

Cuadro 1: iris.head()

La matriz de características será "X" y el objetivo o target será la columna "species", el cual puede tomar 3 valores: "setosa", "versicolor" y "virginica".

Algoritmo 31 "features" & "target"

```
# Eliminamos la columna 'species' del dataframe
X_iris = iris.drop('species', axis=1)
# Almacenamos el target "species" en la variable "y_iris"
y_iris = iris['species']
```

Se procederá a dividir los datos en 2 tipos:

- Training set: datos para el entrenamiento del modelo
- Testing set: datos para probar el modelo entrenado

Este tipo de partición de los datos puede ser manual, pero es mas conveniente usar "train_test_split"

Algoritmo 32 Dividiendo los datos en entrenamiento y prueba

```
#Deprecado: from sklearn.cross_validation import train_test_split
from sklearn.model_selection import train_test_split
Xtrain, Xtest, ytrain, ytest = train_test_split(X_iris, y_iris, random_state=1, test_size
    =0.33)
```

Para hacer la clasificación se va a realizar un modelo conocido como Naive Bayes Gaussiano, asumiendo que cada clase parte de una distribución Gaussiana.

Algoritmo 33 Entrenando el modelo Gaussiano

```
# 1.Choose model class
from sklearn.naive_bayes import GaussianNB

# 2.Instantiate model
model = GaussianNB()

# 3.Fit model to data
model.fit(Xtrain, ytrain)

# 4.Predict on new data
y_model = model.predict(Xtest)
```

Finalmente, podemos usar "accuracy_score" para ver la proporción predicha con respecto al valor verdadero.

Algoritmo 34 accuracy_score

```
from sklearn.metrics import accuracy_score
accuracy_score(ytest, ymodel)
```

Nuestro modelo tiene un 97.36 % de precisión, como se puede observar el algoritmo de clasificación naive es efectivo para este caso particular de datos.

12. Reducción de componentes usando PCA

Un ejemplo del problema del aprendizaje no supervisado es la reducción de la dimensionalidad, para hacer más facil su visualización. Usando los datos "iris", vamos extraer las características escenciales de la data. A menudo, la reducción dimensional es usada para visualizar los datos de forma más fácil en 2 dimensiones.

Algoritmo 35 Análisis de Componentes Principales con los datos de "iris"

```
# 1.Choose the model class
from sklearn.decomposition import PCA

# 2.Instantiate the model with hyperparameters
model = PCA(n_components=2)

# 3.Fit to data. Notice 'y' is not specified!
model.fit(X_iris)

# 4.Transform the data to two dimensions
X_2D = model.transform(X_iris)
```

Los datos de "X_iris" tiene 4 caracteristicas, las cuales fueron reducidas a 2 componentes en "X_2D".

Algoritmo 36 Agregando PCA1 y PCA2 al dataframe "iris.

```
iris['PCA1'] = X_2D[:,0]
iris['PCA2'] = X_2D[:,1]
sns.lmplot('PCA1','PCA2', hue='species', data=iris, fit_reg=False)
```

A continuación se muestran los nuevos datos en una representación bidimensional.

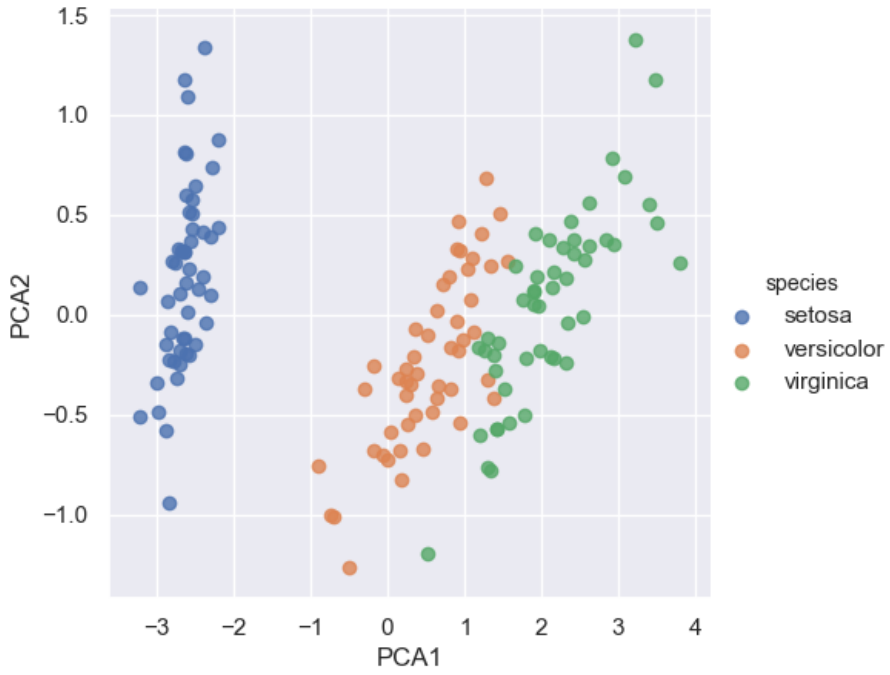


Figura 16: PCA1 y PCA2 de los datos “iris”.

13. Clustering con Gaussian Mixture Model (GMM)

A continuación se va a clasificar los datos usando un poderoso método llamado Gaussian Mixture Model (GMM).

Algoritmo 37 Clustering con GMM

```
# 1.Choose the model class
# Deprecated : from sklearn.mixture import GMM
from sklearn.mixture import GaussianMixture

# 2.Instantiate the model with hyperparameters
model = GMM(n_components=3, covariance_type='full')

# 3.Fit to data. Notice 'y' is not specified!
model.fit(X_iris)

# 4.Determine cluster labels
y_gmm = model.predict(X_iris)
```

Cargando la columna de datos predichos al dataframe 'iris'

Algoritmo 38 Graficando los datos clasificados con GMM

```
iris['cluster'] = y_gmm
sns.lmplot('PCA1', 'PCA2', data=iris, hue='species',
col='cluster', fit_reg=False)
```

A continuación veremos los datos clasificados pero para tener una mejor visualización usaremos PCA1 y PCA2, calculados anteriormente.

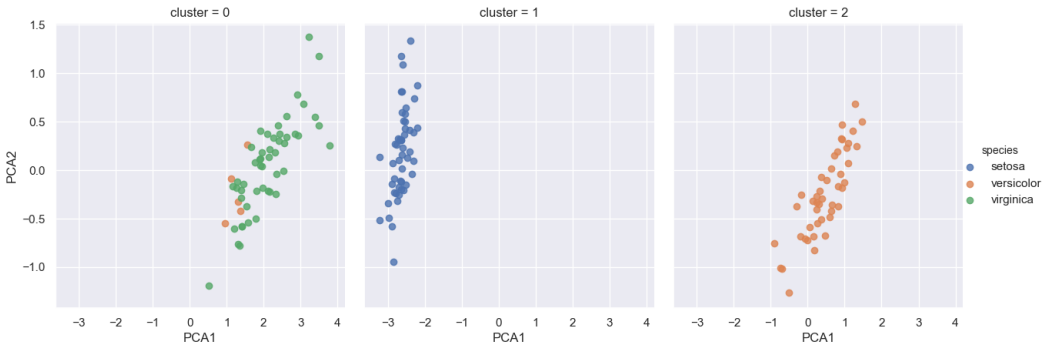


Figura 17: PCA1 y PCA2 usando GMM

Como se puede observar en el "cluster 0", vemos 5 puntos de color naranja que en comparación con lo demás en general es un pequeño error aceptable del modelo entrenado.

14. Árboles de Decisión y Bosques Aleatorios

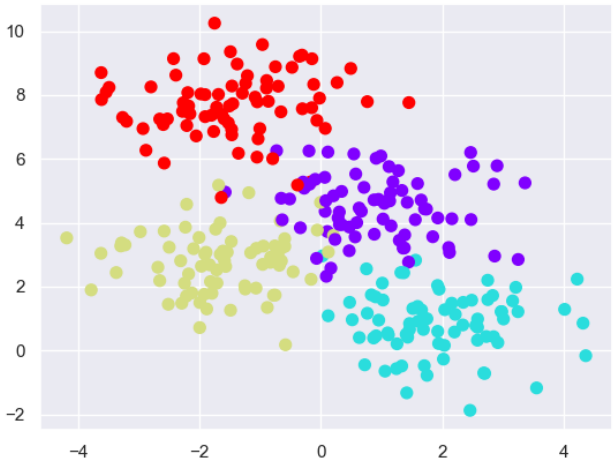
Algoritmo 39 Datasets con 4 tipos.

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import make_blobs

sns.set()
X, y = make_blobs(n_samples=300,  centers=4,
                  random_state=0, cluster_std=1.0)

plt.scatter(X[:,0], X[:,1], c=y, s=50, cmap='rainbow')
plt.show()
```

Algoritmo 40 Dataset para clasificar con «Random Forest»



Un árbol simple de decisión interactúa con los datos dividiendolos usando ejes, acorde a algún criterio cuantitativo «depth». La figura acontinuación representa para los primeros cuatro niveles de un árbol de clasificación:

Algoritmo 41 Entrenando y dibujando el árbol de decisiones

```
# Creating dataset
X, y = make_blobs(n_samples=300, centers=4,
                  random_state=0, cluster_std=1.0)

# Creating subplots 1x4
fig, ax = plt.subplots( 1, 4, figsize=(16,3) )
fig.subplots_adjust(left=0.02, right=0.98, wspace=0.1)

# Fiting and plotting
for axi, depth in zip(ax, range(1,5)):
    model = DecisionTreeClassifier(max_depth=depth)
    visualize_tree(model, X, y, axi=axi)
    axi.set_title('depth = {0}'.format(depth))

# Show figure: Decision-tree-levels
plt.show()
```

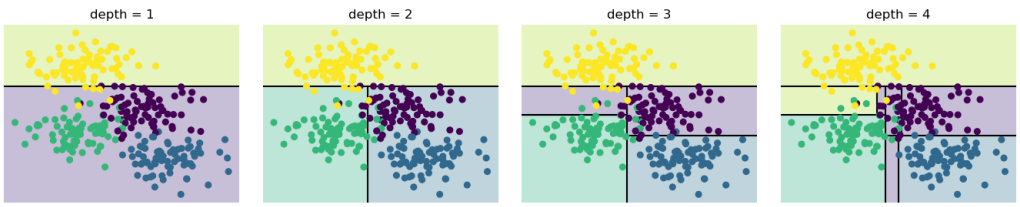


Figura 18: Visualización para los diferentes niveles del árbol de decisión.

Parte IV
Hiperparametros y Validación de Modelos

15. Validar un modelo usando validación cruzada

Para realizar la validación de forma rápida se utilizare "train_test_split", con el parámetro train_size=0.5 los datos seran divididos en 50 % para entrenamiento y 50 % para la validación.

Algoritmo 42 Validación cruzada usando "train_test_split"

```
#Deprecated: cross_validation by model_selection
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# split the data with 50% in each set
X1, X2, y1, y2 = train_test_split(X, y, random_state=0, train_size=0.5)

# fit and evalue the model
y2_model = model.fit(X1, y1).predict(X2)
y1_model = model.fit(X2, y2).predict(X1)

# compute the accuracy by every group
accuracy_score(y1,y1_model), accuracy_score(y2, y2_model)
```

Podemos expandir para una validación cruzada y usarla para más pruebas, por ejemplo separar en 5 grupos (cv=5) y validar cada grupom para ello se usara "cross_val_score", se entrenara el model con 4/5 y se evaluara el modelo con el 1/5 restante.

Algoritmo 43 Validación cruzada de 5 grupos usando "cross_val_score"

```
from sklearn.cross_validation import cross_val_score
cross_vol_score(model, X, y, cv=5)
```

La respuesta es una <numpy.array> por ejemplo: array([0.96666667, 0.96666667, 0.93333333, 1.])

También podemos hacer una validación "uno contra todos", dejamos unicamente un dato para realizar la validación, para ello se usa "LeaveOneOut", tal como se muestra a continuación:

Algoritmo 44 Validación cruzada uno contra todos usando "LeaveOneOut"

```
from sklearn.cross_validation import LeaveOneOut
scores = cross_val_score(model, X, y, cv=LeaveOneOut(len(X)))
scores_mean = scores.mean()
```

16. Validación de curvas