

U03 PRÁCTICA 2

MÁS ALGORITMOS DE CLASIFICACIÓN

**SISTEMAS
DE APRENDIZAJE
AUTOMÁTICO**

**IES SERRA PERENXISA
TORRENT (VALENCIA)**



CLASIFICACIÓN CON SVC, ÁRBOLES Y ASSEMBLES



ACTIVIDAD 1: CLASIFICAR CON SVM.

Crea el notebook `saa-u03-p02-a1-<tus_iniciales>.ipynb` donde completar la actividad. Comenzamos probando diferentes clasificadores sobre unos datos separables linealmente (con un hiperplano).

CLASIFICADORES SVC CON DATOS SEPARABLES LINEALMENTE

Carga el fichero `Social_Network_Ads.csv` en un `DataFrame` de pandas. Este fichero tiene 3 características que son `edad`, `salario` y `compra`. Contiene información de si los usuarios de una red social compran productos que aparecen en la publicidad de esta red social.

a) Define el target `y` como `compra` y `edad`, `salario` como las predictoras `X`.

Divide en train y test usando semilla aleatoria formada por la cantidad de letras de tu nombre y apellidos (en mi caso sería 449 porque "Jose" tiene 4 letras, "Rosa" tiene otras 4 letras y "Rodríguez" tiene 9 letras).

Luego usa `Counter(colección)` tras importarla `"from collections import Counter"` para calcular las proporciones de valores 0 (no compra) y 1 (si compra) que son las clases de esta tarea de clasificación y asegurarnos de que está balanceado.

```
Columns del DataFrame: Index(['edad', 'salario', 'compra'], dtype='object')
Distribución original de y:  0=64.250% 1=35.750%
Distribución de y_train:    0=62.667% 1=37.333%
Distribución de y_test:     0=69.000% 1=31.000%
```

b) Escala las características para usar SVM. Imprime máximo, mínimo y media de los valores escalados.

```
X_train: min [-1.9399 -1.5843] max: [2.1339 2.389 ] media: [ 1.5173e-17 -5.1070e-17]
X_test:  min [-1.9399 -1.5843] max: [2.1339 2.3596] media: [-0.1339  0.1074]
```

c) Crea un clasificador `SVC` con `kernel="Linear"` y `random_state` al valor de las letras de tus apellidos. El resto de hiperparámetros se ignoran (valores por defecto).

d) Para validarlo muestra matriz de confusión, el informe de clasificación y la métrica usada en el entrenamiento que por defecto es accuracy.

```
**** Matriz de Confusión:
[[64  5]
 [10 21]]
**** Informe de clasificación:
              precision    recall  f1-score   support

      0       0.86      0.93      0.90        69
      1       0.81      0.68      0.74        31

   accuracy              0.85       100
  macro avg              0.84      0.80      0.82       100
weighted avg              0.85      0.85      0.85       100

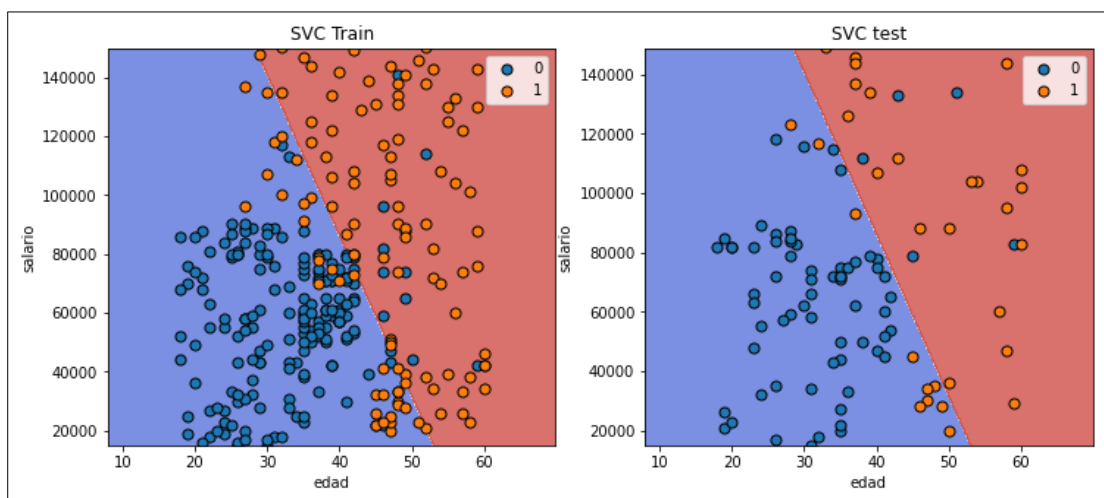
**** Métrica usada para entrenar:
Accuracy: 0.85
```

e) Ahora haz un gráfico que muestre en el train y el test la frontera de decisión. Debes obtener algo similar a esto (en la unidad tienes diferentes listados de como conseguirlos). Define una función `plot_frontera_decision(clasificador, X, y, escalador=None, titulo="")` que:

- Tendrás que importar `matplotlib.colors.ListedColorMap`.
- Define los intervalos donde están los datos: usa la primera (`X[:,0]`) y segunda (`X[:,1]`) columna de `X`.
- Si hay escalador debe recuperar los datos originales aplicando la transformación inversa.
- Luego crea una maya de 500 datos horizontales x 500 verticales que cubran el intervalo de los datos (debes calcular la distancia entre cada punto de la maya en `X1` y `X2` usando el máximo y el mínimo y dividiendo por 500) con `np.meshgrid()`.
- Ahora utilizas `plt.contourf(X1, X2, predicciones.reshape(X1.shape), alpha=0.75, cmap=plt.cm.coolwarm)`. Ten en cuenta que si usas escalador las predicciones se realizan sobre los datos escalados de `X1` y `X2`.
- Ahora añades los puntos de datos como un scatter y puedes ponerle etiquetas edad y salario y titulo (aunque también podría hacerse desde fuera de la función):

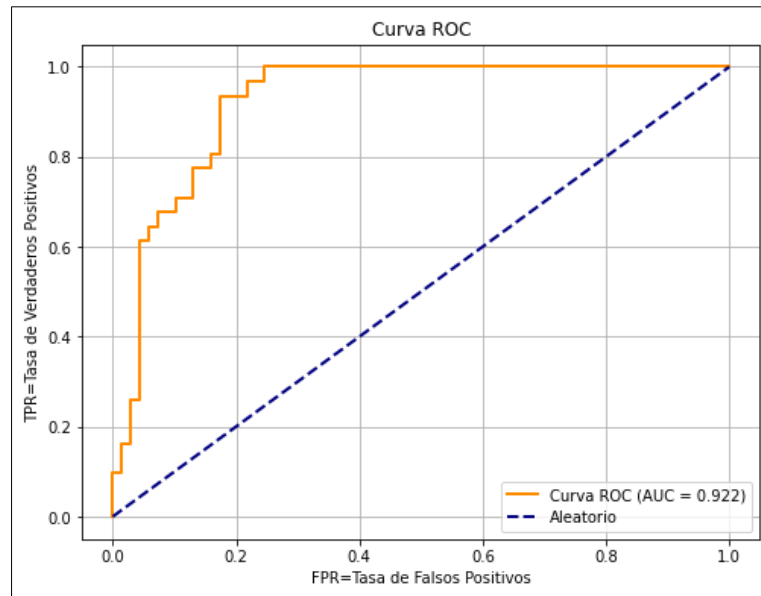
```
24 plt.xlim(X1.min(), X1.max())
25 plt.ylim(X2.min(), X2.max())
26 for i, j in enumerate(np.unique(y_set)):
27     plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1], edgecolors='k', s=50, label = j)
```

- Por último creamos una figura de tamaño 12, 5: `plt.figure(figsize=(12,5))` y la rellenamos haciendo dos llamadas a la función `plot_frontera_decision()`: `plt.subplot(1,2,1) -> dibujar train -> plt.subplot(1,2,2) -> dibujar test`



e) Por último hacemos una función `plot_curva_roc(clf, X_test, y_test)` que defina y dibuje la curva ROC y muestre el AUC. Tienes ejemplos de código en la unidad 2. Ten en cuenta además que hay modelos clasificadores (argumento `clf`) que tienen el método `predict_proba(X_test)` y otros que no lo tienen y en su lugar puedes usar `decision_function(X_test)`. Para saberlo tienes el método `hasattr(clf, "predict_proba")`: que puedes utilizar en un if. Una vez que lo tengas solo tienes que pedir ambas cosas y graficar el resultado:

```
11 fpr, tpr, _ = metrics.roc_curve(y_test, y_scores)
12 roc_auc = metrics.auc(fpr, tpr)
13 # Graficar la curva ROC
14 plt.figure(figsize=(8, 6))
15 plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'Curva ROC (AUC = {roc_auc:.3f})')
16 plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--', label='Aleatorio')
```

**ENTREGA 1:**

a) Completa el código de los diferentes apartados y entrega resultado de su ejecución.

b) No es un mal resultado pero tampoco es muy bueno. Hemos entrenado un modelo asumiendo que los datos son linealmente separables. Si revisas los gráficos que has generado quizás puedas confirmarlo o desmentirlo. Hazlo y dime:

- ¿Qué gráfico te ha convencido? (indica el apartado) _____
- ¿De qué te ha convencido? De que son separables o de que no lo son: _____

CLASIFICADORES SVC CON DATOS NO SEPARABLES LINEALMENTE

Ahora, sobre el mismo conjunto de datos vamos a intentar comparar lo que conseguimos con diferentes tipos de SVC.

a) Vamos a definir en un diccionario diferentes clasificadores SVM que usen diferentes tipos de kernel. Realizamos la importación de elementos y creamos el diccionario.

```
1 from sklearn.model_selection import cross_val_score
2 from sklearn.svm import SVC, LinearSVC, NuSVC
3 from sklearn.svm import SVC, NuSVC
4 import matplotlib.pyplot as plt
5
6 # Definir clasificadores con diferentes kernels
7 clasificadores = {
8     "SVC (RBF)": SVC(kernel="rbf", C=1000, gamma=0.1),
9     "SVC (Polynomial)": SVC(kernel="poly", degree=3, C=1000, gamma='scale'),
10    "SVC (Sigmoid)": SVC(kernel="sigmoid", C=1000, gamma='scale'),
11    "NuSVC": NuSVC()
12 }
```

b) En un bucle vamos a utilizar validación cruzada para buscar el mejor score como la media de scores de 5-folds. modelos que C informe de clasificación y la métrica usada en el entrenamiento que por defecto es accuracy.

```

14 # Evaluar y encontrar el mejor clasificador
15 resultados = {}
16 nombre_mejor_modelo = ""
17 mejor_score = 0
18 loseta_del_grafico = 1
19 plt.figure(figsize=(14,10))
20 for nombre, clf in clasificadores.items():
21     scores = cross_val_score(clf, X_train_escalado, y_train, cv=5, scoring='accuracy')
22     media_de_scores = scores.mean()
23     resultados[nombre] = scores
24     print(f"{nombre}: Precisión media = {media_de_scores:.4f} | Desviación estándar = {scores.std():.4f}")
25     plt.subplot(2, 2, loseta_del_grafico)
26     loseta_del_grafico += 1
27     clf.fit(X_train_escalado, y_train) # Entrenamiento real (validacion cruzada no entrena)
28     plot_frontera(clf=clf, X=X_test_escalado, y=y_test, escalador=s_x, titulo=nombre)
29     if media_de_scores > mejor_score:
30         mejor_score = media_de_scores
31         nombre_mejor_modelo = nombre

```

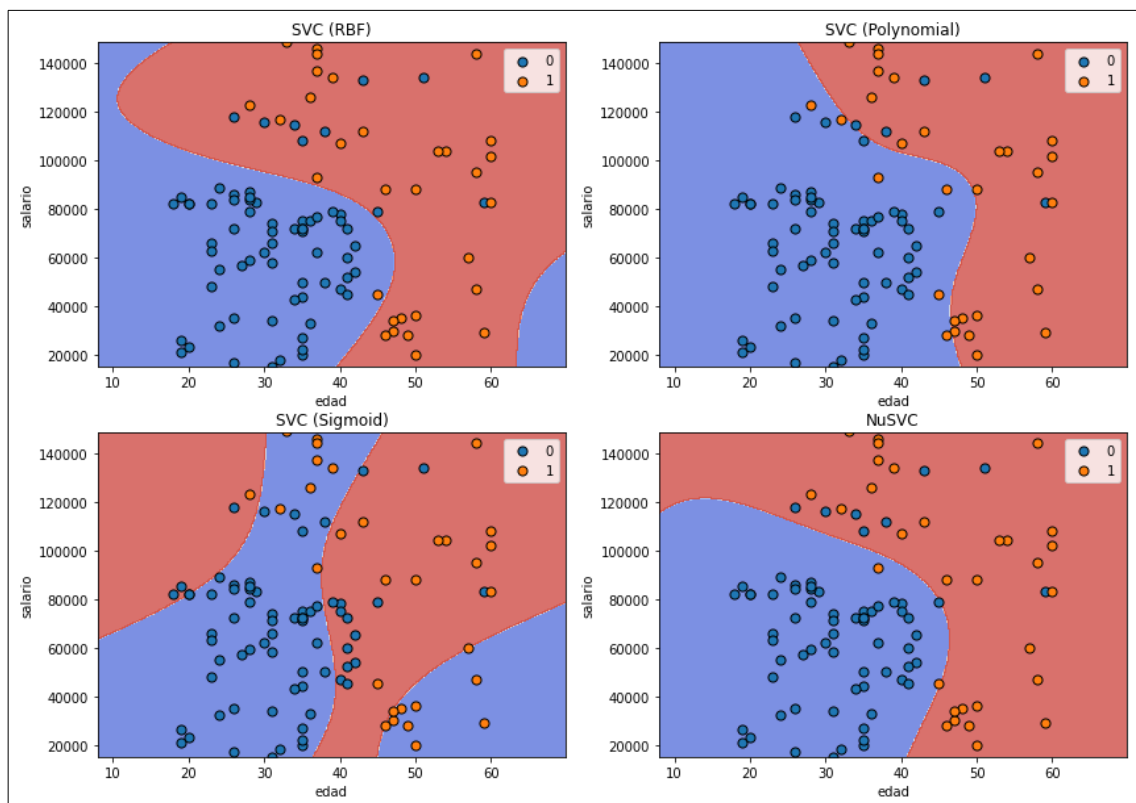
Falta indicar fuera del bucle el mejor clasificador y su score. El resultado de ejecutarlo sería:

```

SVC (RBF): Precisión media = 0.9233 | Desviación estándar = 0.0226
SVC (Polynomial): Precisión media = 0.8467 | Desviación estándar = 0.0414
SVC (Sigmoid): Precisión media = 0.6867 | Desviación estándar = 0.0521
NuSVC: Precisión media = 0.9067 | Desviación estándar = 0.0403

🏆 Mejor clasificador: SVC (RBF) con accuracy media de 0.9233

```



ENTREGA 2:

- Completa el código de los diferentes apartados y entrega resultado de su ejecución (recuerda modificar **random_state** a tu semilla personal).
- En la línea 27 del código que aparece en el apartado b), se vuelve a entrenar el clasificador de la iteración ¿Es necesario? ¿No queda entrenado al llamar antes a **cross_val_score()**?

c) Añade otro gráfico donde aparezcan las curvas **ROC** con el valor **AUC** de los diferentes clasificadores.

d) Intenta mejorar el desempeño de los clasificadores afinando los hiperparámetros de cada modelo que hay en el diccionario de clasificadores copiando el código en una nueva celda a continuación y mostrando su ejecución incluidos los gráficos.

CLASIFICADORES SVC CON DATOS NO BALANCEADOS

Cuando tenemos **datos desbalanceados**, un clasificador tiende a favorecer a la clase más abundante donde se obtienen mejores métricas de desempeño que en el resto de clases. Para solucionar esto, podemos usar varias técnicas, por ejemplo:

- **SMOTE** (*Synthetic Minority Over-sampling Technique*): aumenta la clase minoritaria generando datos sintéticos.
- **Ponderar** (`class_weight='balanced'`): ajusta el coste de los errores para dar más peso a la clase minoritaria.
- **Resamplio Manual**: Duplica ejemplos de la clase minoritaria o elimina de la clase mayoritaria.
- **Cambiar umbral de decisión**: ajusta la frontera de decisión de la clasificación.

a) Crea de manera sintética unos 1000 datos desbalanceados de 20 características, con 2 clases (el 90% de ejemplos es de una) y el otro 10% es de la otra clase usando tu propia semilla aleatoria personal.

```
1 # Crear un conjunto de datos desbalanceado
2 from sklearn.svm import SVC
3 from sklearn.datasets import make_classification
4 from sklearn.model_selection import train_test_split
5 from sklearn.metrics import classification_report
6
7 # Crear dataset desbalanceado
8 X, y = make_classification(n_classes=2, weights=[0.9, 0.1], n_samples=1000, random_state=449)
```

b) Dividir los datos en 2 particiones train y test con el 30% de datos para test y usando tu semilla aleatoria. Clasifica de manera normal estos datos con un clasificador **SVC(kernel='rbf', random_state=449)** y luego imprime un informe de los resultados que obtienes con los datos de validación `classification_report(y_test, y_pred)`.

	precision	recall	f1-score	support
0	0.92	1.00	0.96	269
1	0.90	0.29	0.44	31
accuracy			0.92	300
macro avg	0.91	0.64	0.70	300
weighted avg	0.92	0.92	0.91	300

c) **Ponderar las clases con pesos**: ahora repetimos el mismo código pero añadimos el parámetro `class_weight="balanced"` cuando creamos el clasificador **SVC**. Vuelve a entrenar, genera predicciones e imprime el informe de clasificación. Indica donde hay mejora respecto del caso b).

d) **Balanceo con SMOTE**: aplica **SMOTE** solo en el conjunto de entrenamiento para evitar fuga de datos. Copia otra vez el código del apartado b) y al principio importa el objeto SMOTE: `from imblearn.over_sampling import SMOTE` (quizás tengas que instalar).

```
1 # Balanceo con SMOTE
2 from imblearn.over_sampling import SMOTE # pip install imbalanced-learn
```

Después de particionar configura el objeto **SMOTE** y le pasas los datos de entrenamiento como se ve en la figura cambiando la semilla aleatoria por la tuya:

⊘ Nunca lo apliques antes de dividir los datos.

```
7 # Aplicar SMOTE para sobre-sampear la clase minoritaria
8 smote = SMOTE(sampling_strategy='auto', random_state=449)
9 X_resampled, y_resampled = smote.fit_resample(X_train, y_train)
```

Y ahora entrena el mismo clasificador pero con los datos modificados por **SMOTE**. Genera predicciones e imprime el informe de clasificación. Indica donde hay mejora respecto del caso b).

e) **Ejemplo de Balanceo manual con resampleo:** copia de nuevo el código del apartado b) y al principio importa: `from sklearn.utils import resample`. Tras hacer la partición en *train* y *test*, haz que la cantidad de filas de la clase minoritaria coincida con la cantidad de filas de la mayoritaria con el siguiente código:

```
7 # Dividir en clases
8 X_mayoritaria = X_train[y_train == 0]
9 X_minoritaria = X_train[y_train == 1]
10
11 # Duplicar la clase minoritaria
12 X_minoritaria_aumentada = resample(X_minoritaria, replace=True, n_samples=len(X_mayoritaria), random_state=449)
13 y_mayoritaria = [0] * len(X_mayoritaria)
14 y_minoritaria = [1] * len(X_minoritaria_aumentada)
15
16 # Crear nuevo dataset balanceado
17 X_balanceado = np.vstack((X_mayoritaria, X_minoritaria_aumentada))
18 y_balanceado = np.array(y_mayoritaria + y_minoritaria)
```

Y ahora entrena el mismo clasificador pero con los datos balanceados. Genera predicciones e imprime el informe de clasificación. Indica donde hay mejora respecto del caso b).

f) **Cambiar el umbral de decisión:** copia una vez más el código del apartado b). Tras particionar, cuando se va a crear el modelo **SVC** debes añadir el parámetro **probability=True** porque necesitamos las probabilidades de pertenencia a cada clase. Entrena al modelo normalmente. Luego haz una predicción de las probabilidades de la clase minoritaria con la sentencia: `y_prob = svc.predict_proba(X_test)[: , 1]`. Ahora cambia estas predicciones bajando el umbral para favorecer a la clase minoritaria (en este caso la 1):

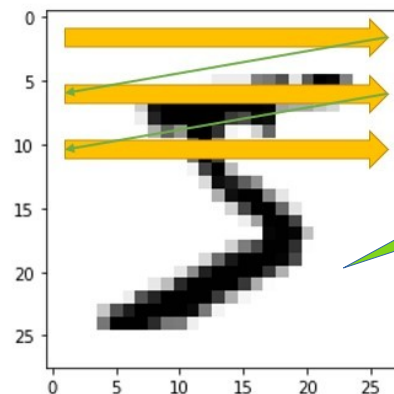
```
13 # Ajustar umbral (ejemplo: en lugar de 0.5, usar 0.3 para favorecer la minoría)
14 y_limite_pred = (y_prob > 0.3).astype(int)
```

Genera predicciones e imprime el informe de clasificación. Indica donde hay mejora respecto del caso b).

CLASIFICAR MNIST

El [dataset MNIST](#) es considerado el "Hola Mundo" de la visión artificial. Contiene un conjunto *train* de 60.000 imágenes de dígitos manuscritos (de 0 a 9) y otro conjunto de *test* con 10.000 muestras adicionales. Las muestras incluidas en el conjunto de entrenamiento fueron el resultado de escanear dígitos manuscritos de 250 personas (estudiantes de institutos y empleados de la oficina del Censo de los Estados Unidos). El dataset de pruebas contiene dígitos escaneados de otras 250 personas diferentes, lo que permite asegurar que los modelos obtenidos son capaces de interpretar dígitos incluso de personas no involucradas en la generación de los datos de entrenamiento. Los datos están balanceados, particionados y desordenados.

Cada una de las 60.000 muestras tiene 785 columnas: la primera columna (la columna de índice 0) contiene la etiqueta de cada muestra (el número representado en la imagen) y las 784 restantes contienen los píxel de la imagen, habiéndose registrado éstos de izquierda a derecha y de arriba abajo. Por tanto cada imagen tiene $(28 \times 28) = 784$ píxels.



⚠ En los datos, el fondo de la imagen es negro y el trazo blanco!!
 Img: fondo=255 trazo=0
 Datos: fondo=0 trazo:255

Como los clasificadores *SVM* son clasificadores binarios, será necesario utilizar **one-versus-rest** para clasificar los 10 dígitos (cada dígito es una clase). El uso de **One-versus-Rest** es automático así que no tenemos que hacer nada al respecto, *scikit* detectará que hay varias clases y lo aplicará generando 10 modelos diferentes, uno para cada clase.

a) **Primero cargamos el dataset** y luego lo particionamos dando el 20% para test y usando tu semilla aleatoria. Te paso la parte de la carga de los datos:

```
1 from sklearn.datasets import fetch_openml
2 from sklearn.model_selection import train_test_split
3
4 # Cargar datos
5 mnist = fetch_openml('mnist_784', version=1, cache=True, as_frame=False)
6 X = mnist["data"]
7 y = mnist["target"].astype(np.uint8)
```

💡 **Cuidadooor!!** El entrenamiento en local con todos los datos podría superar los 20 minutos de cálculos según el hardware que tenga tu ordenador.

Para evitar tirarnos unas horas haciendo pruebas para nada, en vez de entrenar con los 60 mil datos y testear con los 10 mil de test vamos a coger un 1% de los datos: las primeras 600 muestras por ejemplo (filas de la 0 a la 599 para train) y de 600-799 para test. Así trabajaremos sobre estimaciones y cuando tengamos las cosas más claras: quemamos la CPU!!

b) **Comenzamos con un clasificador SVM lineal sencillo.** Crea un objeto *LinearSVC* al que llamas *clf_1* indicando tu semilla aleatoria y un máximo de 200 iteraciones y lo entrenas con *X_train_pruebas = X[0:600]* y con *y_train_pruebas = y[0:600]*.

c) **Medir su desempeño.** Primero creamos *X_test_pruebas = X[600:800]* e *y_test_pruebas = y[600:800]*. Hacemos predicciones y medimos el *accuracy* (que será una estimación del real). Por ahora vemos si tenemos *underfitting* (un puntaje malo de la métrica). Yo obtengo **0.76**, no está mal para tan pocos datos. Pero posiblemente sea mejorable.

```
✓ 0.0s
Accuracy de LinearSVC en y_test_pruebas: 0.7600
```

d) **Intentamos mejorar la métrica normalizando los datos** con estas sentencias antes de dárselos a un nuevo *LinearSVC* llamado *clf_2*:


```

1 from sklearn.preprocessing import StandardScaler
2
3 s_train = StandardScaler()
4 X_train_escalado = s_train.fit_transform( X_train_pruebas.astype(np.float32) )
5 X_test_escalado = s_train.transform( X_test_pruebas.astype(np.float32) )

```

Nota: en este caso en que las características son valores de píxel, siempre van a ir de 0 a 255. Por tanto conocemos el mínimo y el máximo posible. Si no quieres usar un pipeline y prefieres escalar de forma manual, no tienes más que dividir por 255 cada valor de píxel. Yo en los ejemplos uso el escalador en un pipeline: **datos** → (**escala** → **clasifica**) pero en el último apartado he usado solo el clasificador: por tanto el modelo necesita recibir los datos ya escalados.

Me baja el puntaje y además obtengo un mensaje diciendo que el modelo no converge invitándome a subir las iteraciones. Pero quizás es que el modelo lineal sea demasiado simple. Así que cambiamos a un modelo **SVC** con **kernel** de tipo **"rbf"** que es el que se utiliza cuando no sabes la distribución de los datos (es un todo terreno).

```

✓ 0.6s Python
Accuracy de LinearSVC alimentado con datos escalados en y_test_pruebas: 0.7400
d:\python\lib\site-packages\sklearn\svm\_base.py:1235: ConvergenceWarning: Liblinear failed to converge, increase the number of iterations.

```

e) Cambio de clasificador: copio el código del apartado anterior a una nueva celda del *notebook* previamente etiquetada con el texto de este apartado. En el código lo único que cambia es el modelo al que llamamos **clf_3** y su importación. Le indicamos de hiperparámetros además de nuestro **random_state** y un número de iteraciones de 200 el **gamma="scale"**. Lo entrenamos y lo validamos. Ahora vuelvo a obtener el mismo puntaje que al inicio: **0.76**

f) Mejorar la métrica cambiando hiperparámetros: ¿Qué hiperparámetros? ¿Cuales son sus mejores valores? Los más influyentes son **C** y **gamma**. Si no tenemos una intuición a priori de cuales serían los posibles valores adecuados, podemos utilizar *una búsqueda aleatoria* que nos oriente. En una nueva celda del *notebook* haz estas importaciones:

```

1 from sklearn.model_selection import RandomizedSearchCV
2 from scipy.stats import reciprocal, uniform

```

Configuramos un objeto que use validación cruzada y que configure con diferentes valores aleatorios un modelo. No nos va a dar la configuración óptima salvo que le indiquemos un montón de pruebas, pero nos va a permitir orientarnos sobre alrededor de qué valores podemos tener un buen resultado.

```

4 # La dist. reciprocal escoge valores con la misma prob. en escala logaritmica
5 # - En rangos de diferentes escalas como [0.001 a 20] P(coger 0.015) == P(0.15) == P(coger 1.5) == P(coger 15)
6 # - Se usa cuando el parámetro afecta exponencialmente al modelo.
7 # La dist.uniforme todos los valores tienen la misma prob.
8 # - Se usa cuando el valor del parámetro afecta linealmente al modelo.
9 distribuciones_de_parametros = {"gamma": reciprocal(0.001, 10), "C": uniform(1, 20)}
10 cv_aleatoria = RandomizedSearchCV(clf_3, distribuciones_de_parametros, n_iter=30, verbose=2,
11 | | | | | | | | | | cv=3, random_state=449)
12 cv_aleatoria.fit(X_train_escalado, y_train_pruebas)

```

Cambia el número de iteraciones a 30 o 40 y pon tu semilla aleatoria. Por último añade una sentencia que imprima qué configuración ha sido la mejor: **cv_aleatoria.best_estimator_** y el **score** que consigue con esta configuración: **cv_aleatoria.best_score_**. En mi caso:

```

Mejor estimador: SVC(C=np.float64(1.0), gamma=np.float64(0.001),
max_iter=200, random_state=449) con score 0.84

```

g) **Mejorar la métrica buscando configuraciones indicadas:** con la información obtenida en la búsqueda aleatoria creamos un *grid search* en otra celda dando 5 o 7 posibles valores a cada hiperparámetro. El valor indicado en la búsqueda aleatoria puede utilizarse como valor máximo o como mínimo o como valor central. Yo voy a crearme un diccionario con los posibles valores de cada hiperparámetro `{"hp1": [v1, v2...], "hp2": [v1, v2...]}` y construyo un objeto `GridSearchCV()` con el que probar la mejor combinación. Luego imprimo la mejor configuración y el *score* que se obtiene con ella en el conjunto de prueba ya escalado (recuerda que usamos un 1% de los datos *train*, así que en realidad es una estimación de si mejoro o no):

```
Mejor estimador: SVC(C=1000000, gamma=0.0001, max_iter=200, random_state=449) con score 0.84
```

h) **Con esta configuración entreno:** cojo de `X_train= X[0:60000,:]` todas las columnas de las primeras 60 mil muestras, en `X_test=X[60000:, :]` las 10 mil para test, `y_train=y[0:60000]` las 60000 muestras de test y en `y_test=y[60000:]` las 10000 para test.

Puedes convertir los datos de entrenamiento a `np.float32` y dividirlos entre 255 o puedes usar un escalador: creo un *pipeline* llamado `detector_digitos` que escale las características `X_train` y `X_test`, datos que luego recoge el `SVC` que hay en la siguiente etapa y que configuras con `kernel="rbf" C=<mejor_valor_encontrado>, gamma=<mejor_valor_encontrado>, random_state=tu_semilla`.

Entrena el modelo o el *pipeline* con todos los datos, predice con él usando los datos del test y muestra el *score* y el informe de clasificación. En mi caso la peor precisión la tengo en 7.

```
***** Accuracy de detector_digitos: 0.9853
***** Informe de clasificación:
```

	precision	recall	f1-score	support
0	0.98	0.99	0.99	980
1	0.99	0.99	0.99	1135
2	0.98	0.98	0.98	1032
3	0.98	0.99	0.98	1010
4	0.99	0.98	0.99	982
5	0.99	0.98	0.99	892
6	0.99	0.99	0.99	958
7	0.98	0.98	0.98	1028
8	0.98	0.98	0.98	974
9	0.97	0.97	0.97	1009
accuracy			0.99	10000
macro avg	0.99	0.99	0.99	10000
weighted avg	0.99	0.99	0.99	10000

i) **Guarda el modelo en un fichero:** Si tu modelo obtiene un *accuracy* superior al 95% y no tiene peores el resto de métricas de clasificación, guarda en un fichero de disco el objeto de una de estas 2 maneras y con el nombre `"detecta_digit_<tus_iniciales>"` y diferente extensión según la forma de guardarlo que elijas:

- **Usando joblib (recomendado):** es más eficiente para modelos grandes que tienen grandes arrays de *NumPy* (como *SVM*, *RandomForest*, etc.)
- **Usando pickle:** es más lento con arrays grandes.

```
1 import joblib
2 joblib.dump(detector_digitos, 'detecta_digit_josrosrod.joblib')
3
4 import pickle
5 with open('detecta_digit_josrosrod.pkl', 'wb') as f:
6     pickle.dump(detector_digitos, f)
```

Para cargar de nuevo el modelo cuando queramos volver a utilizarlo desde un programa:

- Usando *joblib*:

```
modelo = joblib.Load('modelo_guardado.joblib')
```

- Usando *pickle*:

```
with open('modelo_guardado.pkl', 'rb') as f:
    modelo = pickle.Load(f)
```

j) **Aplicación de escritorio:** hacemos una pequeña aplicación de escritorio que nos permita escribir en pantalla con el ratón y nos diga el número que hemos escrito.

```
1  import tkinter as tk
2  from tkinter import messagebox
3  import joblib
4  import numpy as np
5  from PIL import Image, ImageDraw, ImageOps
6
7  # Cargar el modelo SVC que espera un array de 784 características ya escaladas
8  modelo = joblib.load('detecta_digit_josrosrod.joblib')
9  print(print("Soporte de clases:", modelo.classes_)) # comprobar clases 0-9

11 # Crear la interfaz gráfica con Tkinter
12 class App(tk.Tk):
13     def __init__(self):
14         super().__init__()
15         self.title("Reconocer Números Dibujados")
16         self.geometry("400x500")
17         # Lienzo para dibujar
18         self.canvas = tk.Canvas(self, width=280, height=280, bg='white', cursor="cross")
19         self.canvas.pack(pady=10)
20         # Imagen para almacenar el dibujo
21         self.image = Image.new("L", (64, 64), color=255) # Fondo blanco, 64x64 píxel
22         self.draw = ImageDraw.Draw(self.image)
23         # Botones
24         tk.Button(self, text="Predecir", command=self.predecir).pack(pady=5)
25         tk.Button(self, text="Limpiar", command=self.limpiar).pack(pady=5)
26         # Eventos de dibujo
27         self.canvas.bind("<B1-Motion>", self.dibujar)

29     def dibujar(self, event):
30         x, y = event.x, event.y
31         radio = 8 # Grosor del trazo
32         self.canvas.create_oval(x - radio, y - radio, x + radio, y + radio, fill='black', outline='black')
33         # Escalar al tamaño de la imagen de 64x64 y dibujar sobre ella
34         escala_x = x * (64 / 280)
35         escala_y = y * (64 / 280)
36         self.draw.ellipse((escala_x - 2, escala_y - 2, escala_x + 2, escala_y + 2), fill=0)

38     def predecir(self):
39         imagen_convertida = self.centrar_y_convertir_a_28x28(self.image) # Procesar imagen dibujada
40         if imagen_convertida is None:
41             messagebox.showinfo("Predicción", "No se ha escrito nada")
42         else:
43             imagen_array = imagen_convertida.astype(np.float32) / 255 # No dividir si usas StandarScaler
44             print(imagen_array.shape, " media: ", imagen_array.mean() )
45             prediccion = modelo.predict(imagen_array)
46             prediccion1 = modelo.decision_function(imagen_array)
47             messagebox.showinfo("Predicción", f"Número detectado: {prediccion}\nPuntajes:\n{prediccion1}")

49     def limpiar(self):
50         self.canvas.delete("all")
51         self.draw.rectangle([0, 0, 64, 64], fill=255)
```

```
53     def centrar_y_convertir_a_28x28(self, imagen_original):
54         imagen_invertida = ImageOps.invert(imagen_original) # invertir imagen
55         imagen_28x28 = imagen_invertida.resize((28, 28), Image.LANCZOS) # A 28x28
56         return np.array(imagen_28x28, dtype=np.uint8).flatten().reshape(1,-1)

58     if __name__ == "__main__":
59         app = App()
60         app.mainloop()
```

ENTREGA 3:

- a) Completa los apartados del a) al j) entregando código y resultados de ejecución.
- b) ¿Qué ocurre si aumentas el trazo del programa de prueba del reconocimiento de números?
¿Y si lo disminuyes? Aumenta a 10 y prueba a ver si te reconoce los números del 0 al 9.
¿Mejora, empeora o es indiferente? Disminuye a 4 y vuelve a repetir la prueba. ¿Afecta o no al desempeño? ¿A qué puede deberse?



ACTIVIDAD 2: ÁRBOLES DE DECISIÓN.

Crea el notebook `saa-u03-p02-a2-<tus_iniciales>.ipynb` donde completar la actividad. Comenzamos probando diferentes clasificadores sobre unos datos separables linealmente (con un hiperplano).

CLASIFICADORES BASADOS EN ÁRBOLES DE DECISIÓN

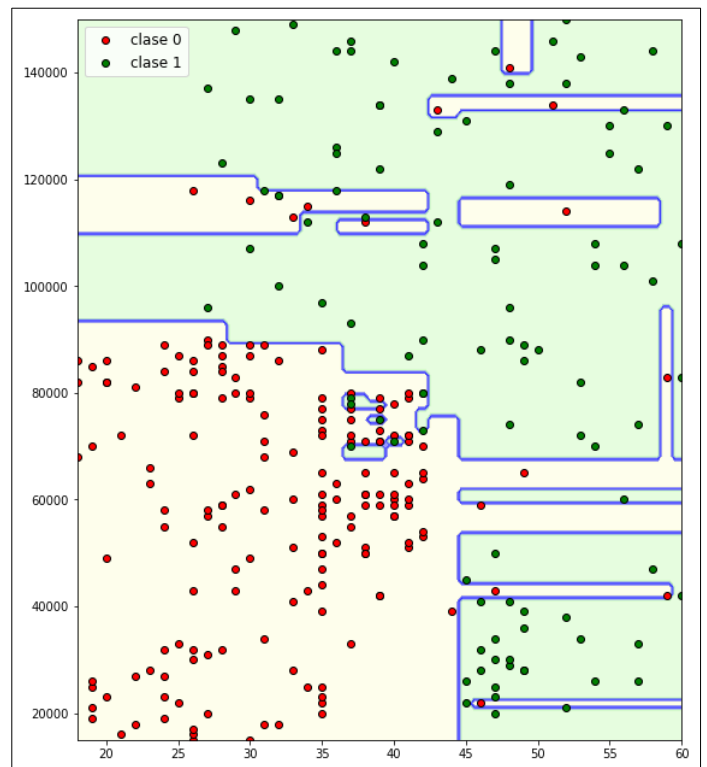
Carga el mismo fichero `Social_Network_Ads.csv` de la actividad 1 en un `DataFrame` de pandas.

a) Define el target `y` como `compra` y `edad`, `salario` como las predictoras `X`. Divide en train y test dando un 30% de los datos para test, usando semilla aleatoria formada por la cantidad de letras de tu nombre y apellidos (en mi caso sería 449 porque "Jose" tiene 4 letras, "Rosa" tiene otras 4 letras y "Rodríguez" tiene 9 letras). Intenta que la partición sea homogénea.

b) Crea un clasificador `DecisionTreeClassifier` llamado `clf_1` con `random_state` a tu semilla. Debes validarlo imprimiendo la `accuracy` tanto en el conjunto de entrenamiento como en el de test. Imprime además el informe de clasificación del test. Después crea una figura de 25 de ancho y 20 de alto y dibuja el árbol con `plot_tree()` indicando `class_names` y `filled=True`. Responde a estas preguntas:

- ¿Cuál es el algoritmo o criterio de particionamiento que usa por defecto el árbol?
- ¿Tiene overfitting el modelo que has entrenado?

c) Mira en la unidad y adapta algún código de ejemplo para crear la función `frontera_decision_tree(arbol,X,y)` que crea una malla de 100x100 puntos (desde el mínimo al máximo de los datos de las dos primeras dimensiones de `X`) y dibuja la superficie de decisión creada por un árbol previamente entrenado que viene como argumento de la función. Luego sitúa los datos de cada muestra con un color asociado a su clase (el árbol guarda la cantidad de características en `arbol.n_classes_` y sus valores en `arbol.classes_`) en la superficie (cada clase con su propio color. Puedes usar un mapa de colores cargado como `cmap=plt.get_cmap("nombre")` donde puedes indicar `"tab10"`, `"tab20"`, `"hsv"`, `"Set1"`, `"nipy_spectral"` que se pueden interpolar: `color=cmap(i/arbol.n_classes_)`.



Luego creas una figura de tamaño (9,11) y haces una llamada pasando el `X_test`, `y_test` y aparece una figura similar a la derecha que es el resultado de usar `X_train`, `y_train`.

d) Crea un nuevo árbol `DecisionTreeClassifier` llamado `clf_2` con `random_state` a tu semilla y criterio de división `'entropy'` y repite las mismas operaciones del apartado b) incluido el dibujo del árbol y el dibujo de la llamada a la frontera de decisión. Responde:

- ¿Cambia la estructura del árbol?

- ¿Cambian las fronteras?
- Los valores del accuracy sugiere cambios en overfitting/underfitting

e) Crea un nuevo árbol **DecisionTreeClassifier** llamado **clf_3** con **random_state** a tu semilla y criterio de división '**log_loss**' y repite las mismas operaciones del apartado b) incluido el dibujo del árbol y el dibujo de la llamada a la frontera de decisión. Responde:

- ¿Cambia la estructura del árbol?
- ¿Cambian las fronteras?
- Los valores del accuracy sugiere cambios en overfitting/underfitting

ENTREGA 4:

a) Completa los apartados entregando código, resultados de ejecución y respuestas a las preguntas.

MÉTODOS PARA REDUCIR EL OVERFITTING

Los árboles de decisión tienden a **sobreajustar** si se dejan crecer sin restricciones. Vamos a probar algunos métodos para prevenir o reducir este resultado:

f) **Limitar la profundidad del árbol (max_depth) o la cantidad de hojas finales del árbol (max_leaf):** reduce la complejidad evitando que el árbol se expanda demasiado. Copia el código del apartado b) y cambia el nombre del clasificador por **clf_4** y le añades el hiperparámetro **max_depth=<profundidad_anterior - 4>**. Muestra **accuracy** en train y test, el informe de clasificación, dibuja el árbol y las fronteras de decisión en el test. Responde:

- ¿Baja el score? ¿Baja el overfitting? ¿El nuevo árbol es más sencillo o más complejo? ¿Las fronteras de decisión se fijan más en los detalles y el ruido? (tienen zonas más irregulares y pequeñas) o ¿tienen menos zonas y con líneas más suaves?

g) **Limitar número mínimo de muestras por nodo (min_samples_split y min_samples_leaf):** reduce la complejidad evitando que el árbol tenga ramas con poca cantidad de muestras. Copia el código del apartado f) y cambia el nombre del clasificador por **clf_5** y quitas el hiperparámetro **max_depth** y añades los hiperparámetros **min_samples_split=10** y **min_samples_leaf=5**. Muestra **accuracy** en train y test, el informe de clasificación, dibuja el árbol y las fronteras de decisión en el test. Responde:

- ¿Baja el score? ¿Baja el overfitting? ¿El nuevo árbol es más sencillo o más complejo? ¿fronteras de decisión irregulares más suaves?

h) **Limitar número de divisiones (min_impurity_decrease):** reduce la complejidad evitando hacer divisiones que no supongan una mejora importante en la capacidad de clasificar. El significado de "mejora importante" se define con el hiperparámetro. Copia el código del apartado f) y cambia el nombre del clasificador por **clf_6** y dejas solamente el hiperparámetro **min_impurity_decrease** a un valor que reduzca el overfitting pero no baje de 95 el puntaje. Muestra **accuracy** en train y test, el informe de clasificación, dibuja el árbol y las fronteras de decisión en el test. Responde:

- ¿Baja el score? ¿Baja el overfitting? ¿El nuevo árbol es más sencillo o más complejo? ¿fronteras de decisión irregulares más suaves?

i) **Usar validación cruzada para combinar restricciones:** utiliza un objeto **GridSearchCV()** con **cv=5** y tu semilla aleatoria que utilice un diccionario donde has definido posibles valores de **max_depth**, **min_samples_split** y **min_impurity_decrease** y encuentra una buena combinación de estos 3 hiperparámetros. A continuación copia el código del apartado f) y cambia el nombre del clasificador por **clf_7** pero ahora le asignas el mejor modelo de la grid search (**best_estimator_**) y lo entrenas con todos los datos y vlidas. Muestra **accuracy** en train

y test, el informe de clasificación, dibuja el árbol y las fronteras de decisión en el test. Responde:

- ¿Baja el score? ¿Baja el overfitting? ¿El nuevo árbol es más sencillo o más complejo? ¿fronteras de decisión irregulares más suaves?

```
Accuracy en Train: 0.9071
Accuracy en Test: 0.9250
Informe de clasificación:
```

	precision	recall	f1-score	support
0	1.00	0.88	0.94	77
1	0.83	1.00	0.91	43
accuracy			0.93	120
macro avg	0.91	0.94	0.92	120
weighted avg	0.94	0.93	0.93	120

```
Mejor configuración: {'max_depth': 4, 'min_impurity_decrease': 0.0001, 'min_samples_split': 10}
```

j) **Combinar varios modelos:** utiliza bagging o boosting para reducir el bias y la varianza de un solo modelo. Adapta la siguiente estructura y la evalúas en un bucle creando figura con 6 gráficos: la primera fila 3 árboles, la segunda fila 3 fronteras de decisión. Escribe además el score ent rain y test de cada uno para saber si bajan el overfitting.

```
8 modelos = {
9     "Random Forest": RandomForestClassifier(n_estimators=100, max_depth=5, random_state=449), # bagging
10    "Gradient Boosting": GradientBoostingClassifier(n_estimators=100, max_depth=5, random_state=449), # boosting
11    "XGBoost": xgb.XGBClassifier(n_estimators=100, max_depth=5, random_state=449)
12 }
```

ENTREGA 5:

a) Completa los apartados entregando código, resultados de ejecución y respuestas a las preguntas.



ACTIVIDAD 3: RECONOCIMIENTO DE CARAS.

Crea el notebook *saa-u03-p02-a3-<tus_iniciales>.ipynb* donde completar la actividad. Vamos a usar el dataset *faces.images* que tiene 400 imágenes de caras, que pertenecen a 40 personas diferentes que aparece cada uno en 10 imágenes. Cada imagen está compuesta por una matriz de 64x64 píxeles. La clave *faces.data* tiene la misma cantidad de datos pero en filas de 4096 características en lugar de usar una estructura de tipo matriz ($4096 = 64 \times 64$).

```
1 from sklearn.datasets import fetch_olivetti_faces
2 faces = fetch_olivetti_faces() # Importamos el dataset de caras
3 print("Claves del dataset: ", faces.keys())
4 print("Dimensiones de faces.images:", faces.images.shape)
5 print("Dimensiones de faces.data: ", faces.data.shape)
6 print("Dimensiones de faces.target:", faces.target.shape)
```

a) Examinamos los datos para comprobar si tenemos que escalar los datos o no, por ejemplo. Completa las sentencias que calculan los siguientes datos:

```
Máximo en faces.data: 1.0
Mínimo en faces.data: 0.0
Media en faces.data: 0.5470426
```

b) Creamos un método para ver cierta cantidad de fotos usando *matplotlib*. Pasamos el array de imágenes, el array que contiene los valores del target y los índices desde y hasta que seleccionan el rango de imágenes a mostrar. En cada imagen aparece el target en rojo y el índice de la foto en verde.

```
1 import matplotlib.pyplot as plt
2
3 def print_caras(imagenes, target, desde=0, hasta=0):
4     # configuramos el tamaño de las imágenes por pulgadas
5     fig = plt.figure(figsize=(12, 12))
6     fig.subplots_adjust(left=0, right=1, bottom=0, top=1, hspace=0.05, wspace=0.05)
7     for i in range(desde, hasta):
8         p = fig.add_subplot(20, 20, i + 1, xticks=[], yticks=[]) # graficar foto en matriz 20x20
9         p.imshow(imagenes[i], cmap=plt.cm.bone)
10        # etiquetamos las imágenes con el valor objetivo (target value)
11        p.text(5, 19, str(target[i]), bbox=dict(facecolor='red', alpha=0.5))
12        p.text(5, 55, str(i), bbox=dict(facecolor='green', alpha=0.5))
13
14 print_caras(faces.images, faces.target, 0, 20)
```

c) Vas a utilizar un clasificador *SVC* de nombre *svc_1*. Las nuevas instancias serán clasificadas como pertenecientes a una determinada categoría en función de que lado del hiperplano entran. Vamos a importar la clase *SVC* desde el módulo *sklearn.svm* y vas a indicar que va a usar un kernel de tipo "*linear*".

d) Divide en *train* y *test* los datos siento *X=faces.data* e *y=faces.target* indicando un 25% de los datos para test y una semilla aleatoria obtenida a partir de tu nombre y apellidos (en mi caso sería *random_state=449* porque "Jose" tiene 4 letras, "Rosa" tiene otras 4 letras y "Rodríguez" tiene 9).

e) Utilizaremos validación cruzada usando *k-fold* de 5. Nos vamos a hacer un método que lo aplique y calcule los puntajes (*accuracy* por defecto). Copia y adapta el código en una celda cambiando la semilla aleatoria por la tuya.

```

1 from sklearn.model_selection import cross_val_score, KFold
2 from scipy.stats import sem
3
4 def evaluar_validacion_cruzada(clf, X, y, K):
5     cv = KFold(n_splits=K, shuffle=True, random_state=449) # creamos un k-fold cross validation iterator
6     scores = cross_val_score(clf, X, y, cv=cv) # por defecto el puntaje es exactitud (accuracy)
7     print(scores)
8     print(f"Accuracy media: {np.mean(scores):.4f} (+/-{sem(scores):.4f})")
9
10 evaluar_validacion_cruzada(svc_1, X_train, y_train, 5)

```

f) Muestra informe de clasificación y matriz de confusión.

```

**** Exactitud training set: 1.0
**** Exactitud testing set: 0.97
**** Informe del Clasificador SVC(kernel='linear'):
      precision    recall  f1-score   support

      0         0.67      1.00      0.80         2
      1         1.00      1.00      1.00         2
      2         1.00      1.00      1.00         1
      3         1.00      1.00      1.00         3
      39         1.00      1.00      1.00         3

   accuracy          0.97      100
  macro avg          0.98      100
 weighted avg          0.98      100

**** Matriz de Confusión:
[[2 0 0 ... 0 0 0]
 [0 2 0 ... 0 0 0]
 [0 0 1 ... 0 0 0]
 ...
 [0 0 0 ... 1 0 0]
 [0 0 0 ... 0 3 0]
 [0 0 0 ... 0 0 3]]

```


ENTREGA 6:

a) Código y capturas de todos los pasos.

ACTIVIDAD 4: CLASIFICAR OBRAS DE ARTE.

Crea el notebook `saa-u03-p02-a4-<tus_iniciales>.ipynb` donde completar la actividad. La actividad va a consistir en implementar un modelo clasificador al que podamos pasar un cuadro y este intente indicarnos el autor de la obra de arte y con qué probabilidad es de este artista.

DEFINIR DATASET

Debes acceder a la carpeta  carpeta y buscar un cuadro de cada uno de estos pintores: *Cezánne, Dalí, Gauguin, Goya, Kandinsky, Miró, Monet, Picasso, Renoir, Rubens, Sorolla, Tiziano, Vangogh y Velázquez*. Ya hay 14 ficheros que puedes usar además de los que tu mismo busques. Debes dejar en la carpeta tus ficheros con el formato `pintor-<tus_iniciales>-fichero.png`.

Una vez tengamos suficientes datos, puedes utilizar la misma técnica que usamos en la práctica anterior para generar un .csv con los datos. Elige tu mismo un tamaño de la imagen (recuerda que nuestros recursos son locales y a mayor tamaño más características y mayor procesamiento necesitaremos, intenta que el tamaño tenga la relación de aspecto 4:3).

Nuestro target será el nombre del artista. Será por tanto una clasificación multiclase.

IMPLEMENTAR CLASIFICADORES

Prueba al menos a entrenar 2 clasificadores:

- El primero: SVC ó Árbol ó RandomForest.
- El segundo: método ensemble: Boost/Bagging.

Puedes utilizar lo que hicimos en la práctica anterior: defines un diccionario que contenga pipelines que preprocesen y entrenen a cada clasificador y lo haces en un bucle. O puedes hacerlo uno a uno de forma manual, como prefieras.

MEDIR DESEMPEÑO

Mide el desempeño de todos, muestra informes de entrenamiento y la matriz de confusión. Haz un gráfico de barras donde aparezcan los puntajes obtenidos por cada uno en train y test.

MEJORAR ALGUNO AJUSTANDO HIPERPARÁMETROS

Escoge el que quieras y define el objetivo que quieres conseguir: o mejorar su puntaje o bajar el overfitting, mediante el ajuste de uno o varios de sus hiperparámetros. Mide y muestra de nuevo el desempeño en train y en test.

ENTREGA 7:

- Fotografías subidas a la carpeta compartida: _____
- Fichero csv con los datos de las fotografías transformado y código que las transforma.
- División de los datos en train y test dejando el 80% de datos para train e intentando no desbalancear los datos y con tu propia semilla aleatoria.
- Realizar entrenamiento de los 4 clasificadores.
- Mostrar informe de clasificación.
- Mostrar matriz de confusión.
- Mostrar gráfico comparativo.
- Mejora del desempeño de alguno cambiando sus hiperparámetros.