U02 PRÁCTICA 4

ALGORITMOS DE CLASIFICACIÓN

SISTEMAS DE APRENDIZAJE AUTOMÁTICO

IES SERRA PERENXISA
TORRENT (VALENCIA)

ALGORITMOS DE CLASIFICACIÓN

📌ACTIVIDAD 1: MNIST

MNIST es un conjunto de pequeñas imágenes de dígitos escritos por estudiantes de institutos y empleados del censo de USA. Cada imagen está etiquetada con el dígito que representa. Crea el notebook ***saa_02_p04_a1_<tus_iniciales>.ipynb*** donde realizar esta actividad.

> ***Nota:*** recuerda que los datasets cargados por *scikit-Learn* tienen estructura de diccionario incluyendo entre otras las claves:
>
> - ***DESCR***: descripción del dataset.
> - ***data***: ***DataFrame*** de pandas con una fila por instancia o ejemplo de datos y una columna por cada característica.
> - ***target***: un array con los *labels* de cada instancia.

Comprobamos la cantidad de datos y separamos predictoras y target:

```
1  from sklearn.datasets import fetch_openml
2  mnist = fetch_openml('mnist_784', version=1)
3  print("Claves de mnist: ", mnist.keys())
4
5  X, y = mnist["data"], mnist["target"]
6  print("Dimensiones de X:", X.shape)
```

PASO 1: PREPARAR LOS DATOS.

Tiene imágenes cada una con 784 características porque son de tamaño 28×28 pixels y cada pixel representa la intensidad de luz (0 blanco a 255 el negro). Tiene aproximadamente la misma cantidad de ejemplos para cada dígito (0, 1, …, 9). Para dibujar una de ellos lo único que tienes que hacer es extraer una instancia, cambiarle la estructura a 28×28 y usar la función ***imshow()*** de ***matplotlib***:

```
8  import matplotlib as mpl
9  import matplotlib.pyplot as plt
10 un_digito = X.iloc[0].to_numpy()
11 imagen_un_digito = un_digito.reshape(28, 28)
12 plt.imshow(imagen_un_digito, cmap = mpl.cm.binary, interpolation="nearest")
13 plt.axis("off")
14 plt.show()
```

Vamos a entrenar modelos de clasificación binaria que aprendan a identificar los números 2 del resto. Primero vamos a cambiar el tipo de datos del *target* porque nos interesa que sean numéricos 0 (si no es un 2 y 1 cuando sea la imagen de un dígito 2). Además vamos a generar el dataset de *train* y *test*, y luego lo vamos a adaptar para que nos sirva para entrenar el clasificador del 2. El tamaño del test lo dejamos al 1% porque hay bastantes instancias:

```
17 import numpy as np
18 from sklearn.model_selection import train_test_split
19 y = y.iloc[:].to_numpy()
20 y = y.astype(np.uint8)
21 print("Nuevo tipo de y[0]:", type(y[0]))
22
23 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1, stratify=y, random_state=123)
24 y_train_2 = (y_train == 2)
25 y_test_2 = (y_test == 2)    # True para todas las instancias que sean del dígito 2
```

ENTREGAR:

a) Modifica el código para que en la variable *un_digito* quede cargada la primera instancia que sea un 2.

b) Cuando particiones los datos y realices otras operaciones aleatorias utiliza una semilla aleatoria que coincida con la longitud de tu nombre concatenada a la de tu primer apellido y concatenada a la de tu segundo apellido. En mi caso sería el valor 449 porque "Jose"=4, "Rosa"=4 y "=9.

c) Indica cuantas instancias tenemos para entrenar y cuantas para testear.

PASO 2: ENTRENAR UN MODELO QUE IDENTIFIQUE EL DÍGITO 2

Para identificar el número 2 del resto, necesitamos un clasificador binario (2 clases en las que clasificar una instancia). Usamos uno que use Descenso por Gradiente Estocástico (***`SGDClassifier`***) que es escalable (rápido con muchas filas o instancias) y apto para *on-line* al ir aprendiendo instancia a instancia de manera independiente.

```
27  from sklearn.linear_model import SGDClassifier
28  sgd = SGDClassifier(random_state=123)
29  sgd.fit(X_train, y_train_2)
```

Para comprobar si se ha entrenado bien podemos probar la predicción que hace con algún ejemplo que conozcamos como por ejemplo el que tenemos en un_digito:

```
31  print("Predice un 2?:", sgd.predict([un_digito]))
```

Pero lo ideal es medir lo bien que lo hace en muchos datos y además para que el resultado sea fiable esos datos no deben tener sesgos. Así que voy a obtener sus éxitos en varios conjuntos de datos usando validación cruzada de 3-Folds.

Usamos la función ***`cross_val_score()`*** para validar el clasificador usando *k-folds validación cruzada* con un *k* de 3 (recuerda que la *3-folds validación cruzada* divide el dataset *train* en 3 trozos y recorre todos los trozos validando en cada uno el modelo que entrena con el resto de trozos). Copia este código a continuación y completa los apartados que hay que entregar:

```
33  from sklearn.model_selection import cross_val_score
34  cvs = cross_val_score(sgd, X_train, y_train_2, cv=3, scoring="accuracy")
35  print(f"SGD: Accuracy en 3-folds: {cvs} Media: {cvs.mean()*100:.4f}%" )
```

ENTREGAR:

a) ¿Qué porcentaje de aciertos tiene el modelo cuando realiza predicciones?

b) Siendo un clasificador, ¿Si se acerca al 100% es un buen indicador de que ha aprendido bien?

c) Escribe el código del clasificador *"Siempre Negativo"* de abajo y comprueba de la misma manera que hemos realizado con ***`SGDClassifier`*** si hace su trabajo con éxito (ten en cuenta que no necesita entrenamiento, cuando le pregunten si es un 2 siempre va a responder que no) por tanto no lo entrenes, solo lo validas y compruebas sus porcentajes de éxitos con este comportamiento que es independiente de los datos con los que trabaje.

```
37  from sklearn.base import BaseEstimator
38  class DonVangall(BaseEstimator):
39      def fit(self, X, y=None):
40          pass
41      def predict(self, X):
42          return np.zeros( (len(X), 1), dtype=bool)
```

```
Siempre Negativo: Accuracy en 3-folds: [ [redacted] 28571] Media: [redacted]%
```

d) A la vista de los resultados del apartado **c)** ¿La *accuracy* es fiable para saber si un clasificador es bueno?

PASO 3: VALIDAR EL MODELO CON LA MATRIZ DE CONFUSIÓN.

Para calcular la matriz de confusión necesitamos predicciones que se puedan comparar con los valores reales de la columna target. Por ahora los datos de test los tenemos escondidos y reservados para las fases finales del proyecto, justo antes de decidir si ponemos en producción el sistema. Así que utilizamos la función ***cross_val_predict()***. A continuación obtenemos la matriz de confusión con ***confusion_matrix()***.

```
48  from sklearn.model_selection import cross_val_predict
49  y_train_pred = cross_val_predict(sgd, X_train, y_train_2, cv=3)
50
51  from sklearn.metrics import confusion_matrix
52  print("Matriz de confusión:\n", confusion_matrix(y_train_2, y_train_pred))
```

Además *scikit-learn* tiene funciones para cada métrica:

```
54  from sklearn.metrics import precision_score, recall_score, f1_score
55  print("Precisión:", precision_score(y_train_2, y_train_pred))
56  print("recall:", recall_score(y_train_2, y_train_pred) )
57  print("F1-score:", f1_score(y_train_2, y_train_pred))
```

Si queremos las métricas desglosadas por clases podemos pedir el informe de clasificación con la función ***clasification_report()***.

ENTREGAR:

a) Cada fila de la matriz de confusión representa una clase ()Real ()Predicha.
Cada columna de la matriz de confusión representa una clase ()Real ()Predicha.

b) Para que el clasificador sea bueno ¿Dónde deben estar los valores más altos de la matriz de confusión? ( )En la diagonal. ( )Fuera de la diagonal.

c) Observa la salida del informe de clasificación e indica la clase con peor accuracy pero no de todos los intentos sino de los intentos que sean positivos (es decir, de todos los positivos cuantos ha clasificado correctamente).

PASO 4: EQUILIBRO PRECISIÓN / RECALL.

Scikit-Learn no nos permite acceder directamente a los umbrales, pero como nos deja consultar los *scores* que usa para predecir llamando al método ***decision_function()*** del clasificador en vez de usar ***predict()***, lo que nos devuelve un score para cada instancia y entonces hacemos predicciones usando el umbral que queramos basándonos en estos *scores*:

```
64  y_scores = sgd.decision_function([un_digito])
65  print("Scores del primer 2:", y_scores)
66  umbral = 0
67  prediccion_de_un_digito = (y_scores > umbral)
68  print(f"Con un umbral de {umbral} Un 2 es un 2= {prediccion_de_un_digito}")
```

El clasificador ***SGDClassifier()*** usa un umbral igual a 0, así que el código anterior devuelve el mismo resultado que ***predict()***. Si cambiamos el umbral:

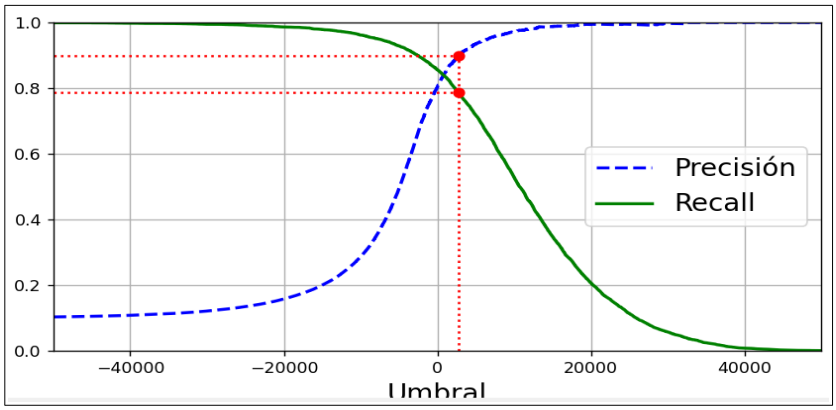
```
70  umbral = 8000
71  prediccion_de_un_digito = (y_scores > umbral)
72  print(f"Con un umbral de {umbral} Un 2 es un 2= {prediccion_de_un_digito}")
```

ENTREGAR:

a) A la vista del resultado, si subes el umbral ¿Subes la *precisión* o el *recall*?

b) Si queremos saber cuanto *recall* vamos a tener si queremos alcanzar una precisión del 90%, lee el texto y completa el código siguiente para conseguir el gráfico de la figura.

Como el umbral controla ambas métricas (*precisión* y *recall*) ¿Cómo elegir uno? Utilizamos todas las instancias del entrenamiento usando ***cross_val_predict()*** pero indicando que quieres *scores de decisión* en vez de predicciones. Con estos *scores* calculas las métricas de *precisión* y *recall* para todos los posibles umbrales usando la función ***precision_recall_curve()*** y puedes visualizarlo en un gráfico para ver como quedarán ambas métricas al escoger un umbral. Normalmente si aumentas el umbral sube la *precisión* (bajan los *FP*) y baja el *recall* (suben los *FN*), aunque puede haber zonas (según se distribuyen los *scores* de decisión) donde se produzcan bajadas o subidas.




```python
def plot_precision_recall_vs_umbrales(precisiones, recalls, umbrales):
    plt.plot(umbrales, precisiones[:-1], "b--", label="Precisión")
    plt.plot(umbrales, recalls[:-1], "g-", label="Recall")
    # Destacar los umbrales con linewidth=2 en plot()
    # Añadir leyendas con plt.legend()
    # Etiquetas al eje x con plt.xlabel()
    # Visualizar una rejilla con plt.grid()
    # Definir rango de valores de ejes: [x_desde, x_hasta, y_desde, y_hasta] con plt.axis()
```

```python
y_scores = cross_val_predict(sgd, X_train, y_train_2, cv=3, method="decision_function")
precisiones, recalls, umbrales = precision_recall_curve(y_train_2, y_scores)
recall_precision_90 = recalls[np.argmax(precisiones >= 0.90)]

umbral_para_precision_90 = umbrales[np.argmax(precisiones >= 0.90)]

plt.figure(figsize=(8, 4))
plot_precision_recall_vs_umbrales(precisiones, recalls, umbrales)
plt.plot([umbral_para_precision_90, umbral_para_precision_90], [0., 0.9], "r:")
plt.plot([-50000, umbral_para_precision_90], [0.9, 0.9], "r:")
plt.plot([umbral_para_precision_90], [0.9], "ro")
plt.show()
```

Mirando el gráfico puedes saber que necesitas un umbral de unos ocho mil para tener una precisión del 90% pero podemos ser más precisos usando ***np.argmax()*** para encontrar el primer índice del máximo valor (en este caso el primer valor True). Para hacer predicciones (en el dataset train por ahora) en vez de llamar a ***clasificador.predict()*** ejecutas las sentencias:

```python
umbral_para_precision_90 = umbrales[np.argmax(precisiones >= 0.90)]
print("Umbral para alcanzar precisión del 90%:", umbral_para_precision_90)
y_train_pred_90 = (y_scores >= umbral_para_precision_90)
print("Precision con umbral",umbral_para_precision_90, "= ", precision_score(y_train_2, y_train_pred_90))
print("Recall con umbral",umbral_para_precision_90, "= ", recall_score(y_train_2, y_train_pred_90))
```

c) ¿Qué valores de *recall* vamos a tener si queremos alcanzar una precisión del 90%?

PASO 5: LA CURVA ROC.

La curva *receiver operating characteristic* (*ROC*) y la métrica *AUC* son otras herramientas usadas en los clasificadores. También se utilizan para comparar varios clasificadores. Para dibujarla hay que calcular el *TPR* y el FPR para diferentes valores de umbral usando la función ***roc_curve()***:

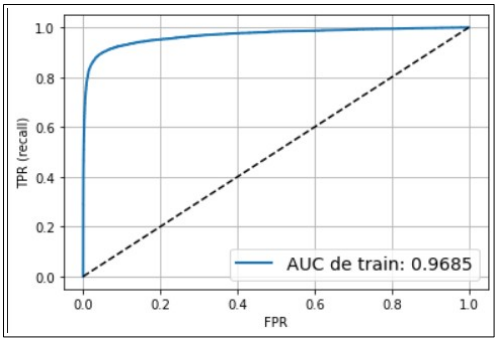
```
110  from sklearn.metrics import roc_curve
111  from sklearn.metrics import roc_auc_score
112  fpr, tpr, umbrales = roc_curve(y_train_2, y_scores)
113
114  def plot_curva_roc(fpr, tpr, label=None):
115      # Dibujar los fpr y sus tpr asociados con plot() ancho de línea 2 y etiqueta label
116      # Dibujar una línea punteada en negro "k--" con plot() desde (0,0) a (1,1)
117      # plt.legend(loc="lower right", fontsize=14)
118      # Poner etiqueta del eje Y a "FPR (recall)" con ylabel()
119      # Poner etiqueta del eje X a "TPR" con xlabel()
120      # Activar la rejilla con grid()
```

ENTREGAR:

a) Completa el código de la función ***plot_curva_roc()*** sustituyendo cada línea comentada por el código que la implementa.

b) ¿Qué *AUC* tendrá un clasificador binario aleatorio? ¿Y un clasificador binario perfecto?

c) Sabiendo que puedes calcular el área bajo la curva con la función ***roc_auc_score(y_train_2, y_scores)*** añade el código en una llamada a ***plot_curva_roc()*** que genere esta gráfica:



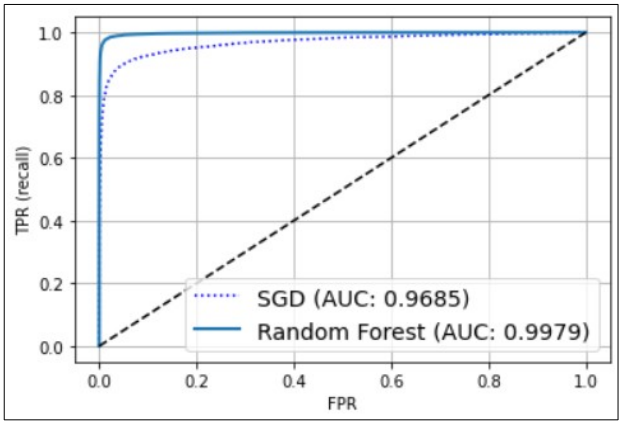
Vamos a entrenar un clasificador ***RandomForestClassifier*** y comparamos su curva *ROC* y su métrica *AUC* con el ***SGDClassifier***. Necesitamos los *scores* de cada instancia en el dataset *train*. Pero debido a la manera de funcionar de *Ramdom Forest* no tiene ***decision_function()***. En su lugar tiene un método ***predict_proba()*** que devuelve un array con una fila para cada instancia y una columna para cada clase con la que trabaja el clasificador. Cada valor representa la probabilidad de que cada instancia pertenezca a cada clase. Por ejemplo en nuestro caso una fila de una instancia podría tener:

clase No 2	clase 2
:	:
0.7	0.3
:	:

```
# En Python:
[
   :
 [0.7, 0.3]
  :
]
```

```
125  from sklearn.ensemble import RandomForestClassifier
126
127  rfc = RandomForestClassifier(random_state=123)
128  y_probs_forest = cross_val_predict(rfc, X_train, y_train_2, cv=3, method="predict_proba")
129
130  y_scores_forest = y_probs_forest[:, 1]    # score = probabilidad de la clase positiva
131  fpr_forest, tpr_forest, umbrales_forest = roc_curve(y_train_2, y_scores_forest)
132
133  plt.plot(fpr, tpr, "b:", label= f"SGD (AUC: {roc_auc_score(y_train_2, y_scores):.4f})")
```

d) Tras dibujar el gráfico (en la línea 133) haz una llamada a ***plot_curva_roc()*** para dibujar la curva del clasificador *Random Forest* y muestra el gráfico para que quede como el siguiente.



e) Basándonos en la métrica ¿Cuál de los dos clasificadores reconoce mejor el dígito 2?

PASO 6: CLASIFICACIÓN MULTICLASE

Un clasificador debe clasificar una instancia en alguna de las clases. Si hay 2 clases hablamos de un clasificador binario. Si hay más de 2, hablamos de clasificador multiclase o multinomial. Algunos algoritmos como *Random Forest* o *Bayes* la soportan directamente. Otros como *SVM* o los clasificadores lineales son estrictamente binarios pero se pueden entrenar varios de ellos (uno para diferenciar entre cada clase y el resto) y predecir con todos para saber cuál es la clase más probable: si tengo 3 clases se fabrica uno para (clase1, resto), otro para (clase2,resto) y otro para (clase3, resto). Esta técnica se llama ***one-versus-rest OvR***.

Otra estrategia es entrenar por parejas de clases de forma que si tenemos 3 clases uno aprenda a diferenciar (entre 1 y 2), otro (entre 1 y 3), otro (entre 2 y 3). Si tienes N clases, necesitas entrenar N x (N -1) / 2 clasificadores. Esto se llama ***one-versus-one: OvO***.

Scikit-Learn detecta automáticamente cuando estás usando un algoritmo clasificador binario para realizar una clasificación multiclase y automáticamente ejecuta ***ovr*** (excepto para *SVM* que usa ovo). Vamos a probar y para ver los detalles de la decisión puedes usar el método ***decision_function()***:

```
137  sgd.fit(X_train, y_train)     # y_train, no es y_train_2
138  print("Predicción de un 2", sgd.predict([un_digito]))
139  scores_de_un_digito = sgd.decision_function([un_digito])
140  print("scores de un_digito",scores_de_un_digito)
```

ENTREGAR:

a) En ejecutar las sentencias de la 137 a la 138 habrá tardado algo más de tiempo (si estás realizando la práctica en un *jupyter notebook* y has dejado este código en su propia celda) ¿Cuánto? ¿Cuántos modelos ha entrenado Scikit-learn? ¿Qué método ha usado?

b) ¿Cuántos scores devuelve la sentencia 139? ¿Cuál es la posición del más alto?

Vamos a obtener la clase que ha obtenido un mayor *score*: sabiendo que cuando se entrena un clasificador, este almacena en su atributo ***classes_*** ordenadas por valor en algún lugar del array.

```
142   mayor = np.argmax(scores_de_un_digito)
143   print("Clases del clasificador:", sgd.classes_)
144   print("La clase con mayor score:", sgd.classes_[mayor])
```

Si quieres forzar a *Scikit-Learn* a usar los métodos *ovo* o bien *ovr* puedes usar las clases ***OneVsOneClassifier*** o ***OneVsRestClassifier*** respectivamente. Debes crear una instancia del objeto pasando en la llamada un clasificador como en este código:

```
146   from sklearn.multiclass import OneVsOneClassifier
147   ovo = OneVsOneClassifier(SGDClassifier(random_state=123))
148   ovo.fit(X_train.values, y_train) # df.values quita cabeceras del df y no da warning
149   print("SGD: Prediccion de un dígito 2:", ovo.predict([un_digito]))
150   print("SGD: Longitud de los estimadores", len(ovo.estimators_))
151   # Entrenar un bosque aleatorio es más sencillo porque soporta multiclase
152   rfc.fit(X_train.values, y_train)                  # Uso values para que no use cabeceras y no de warning
153   print("Random Forest: Predicción de un dígito 2:",rfc.predict([un_digito]))
154   print("Random Forest: Probabilidades predichas del dígito:", rfc.predict_proba([un_digito]))
```

c) ¿Cuántos modelos se han entrenado en ovo? ¿Cuántos en *rfc*? ¿Cuál es el resultado de la línea 150? ¿Si lo hiciésemos con *rfc* ¿Cuál sería el resultado?

d) En el resultado de ejecutar la línea 154 ¿Qué probabilidad da a que el ejemplo sea un 2? ¿Cuál es la siguiente probabilidad más alta? ¿De qué clase se trata?

Si queremos evaluar varios modelos lo normal será usar validación cruzada usando ***cross_val_score()***. Y si aplicas preproceso de datos cabe la posibilidad de que mejores la métrica. Escalaremos los datos de train a ver si mejoramos la métrica.

```
156   cv1 = cross_val_score(sgd, X_train.values, y_train, cv=3, scoring="accuracy")
157   print("SGD: Accuracy de 3-fold CV:", cv1, f"media: {cv1.mean():.4f}")
158
159   from sklearn.preprocessing import StandardScaler
160   escaleitor = StandardScaler()
161   X_train_escalado = escaleitor.fit_transform(X_train.astype(np.float64))
162   cv2= cross_val_score(sgd, X_train_escalado, y_train, cv=3, scoring="accuracy")
163   print("SGD: Accuracy de 3-folds CV tras escalar:", cv2, f"media: {cv2.mean():.4f}")
```

e) ¿Cuáles son los valores de cv1 y cv2 y la media? ¿Qué mejora, si la hay, del % medio hay al escalar?

PASO 7: ANÁLISIS DE ERROR

En un proyecto real además de habríamos realizado análisis exploratorio de datos, ingeniería de características y localizado el mejor modelo con la mejor configuración de manera automatizada con ***GridSearchCV()***. Si lo hemos realizado y ya tenemos un modelo prometedor, nos falta saber si generaliza bien. Vamos a comprobar que cantidad y tipos de error comete al realizar su trabajo a través de la matriz de confusión con la función ***cross_val_predict()*** y luego ***confusion_matrix()***. Para ver una representación visual de la matriz de confusión vamos a utilizar ***matplotlib.matshow()*** o hacer un mapa de calor.

```
165   X_test_escalado = escaleitor.transform(X_test.values)
166   y_test_pred = cross_val_predict(sgd, X_test_escalado, y_test, cv=3)
167   mc = confusion_matrix(y_test, y_test_pred)
168   print("Matriz de confusión de SGD:\n", mc)
```

La diagonal parece que tiene los valores más altos, lo que es buena señal. Podemos dividir la cantidad de errores entre el número de ocurrencias de la cada característica por si el conjunto no estuviese balanceado y así visualmente detectar clases concretas donde hay problemas. La diagonal de la matriz normalizada la ponemos a 0.

```
170  suma_filas = mc.sum(axis=1, keepdims=True)
171  mc_normalizada = mc / suma_filas
172  np.fill_diagonal(mc_normalizada, 0)
173  plt.matshow(mc_normalizada, cmap=plt.cm.gray)
174  plt.show()
```

📌ACTIVIDAD 2: MODELO DE REGRESIÓN LOGÍSTICA.

PASO 1: CARGA DE DATOS Y CREACIÓN DEL MODELO.

Crea el notebook ***saa_02_p04_a2_<tus_iniciales>.ipynb*** donde realizar esta actividad. Queremos usar el fichero **wine.csv** para implementar un clasificador que nos indique el productor de la Toscana (primera columna, pueden ser 3 y ya están codificados como 1, 2 y 3) a partir de parámetros obtenidos de análisis químicos de sus vinos.

En la práctica de la unidad 1 vimos que una manera de mejorar los datos era usar una técnica de ***selección de características*** que consiste en realizar un estudio de sus propiedades estadísticas individuales y por parejas, y así descubrir si algunas podían descartarse porque en vez de aportar información lo que hacían era en el mejor de los casos estorbar y en otros confundir a los algoritmos de aprendizaje al meter ruidos y hacerlos aprender mal. Ahora vamos a realizar algo parecido, pero con otro enfoque. Primero vamos a construir el modelo y luego vamos a descubrir qué características son las que más información aportan y cuales son prescindibles.

a) Usa este código en el fichero ***saa_u02_p04_a2_<tus_iniciales>.py*** (o notebook) para cargar los datos y completa el resto de apartados. Pero cuando particiones los datos y realices otras operaciones donde intervenga el azar añade una semilla aleatoria para que el proceso sea repetible (que coja los mismos datos, que evolucione igual, etc.):

```
# -*- coding: utf-8 -*-
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split

df_vino = pd.read_csv('wine.csv', header=None)
df_vino.columns = ['Class', 'Alcohol', 'Ácido Málico', 'Posos',
                   'Alcalinidad de posos', 'Magnesio', 'Total Fenoles',
                   'Flavanoides', 'No flavanoides fenoles', 'Proanticianinas',
                   'Intensidad Color', 'Saturación',
                   'OD280/OD315 de vinos diluidos', 'Prolina']
print('Clases', np.unique(df_vino['Class']))
print(df_vino.head())
# Dividir en train + test
X, y = df_vino.iloc[:, 1:].values, df_vino.iloc[:, 0].values
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=0)
```

b) Crea un objeto ***multiclass.OneVsRestClassifier(LogisticRegression(..))*** para que utilice el ***método uno contra el resto*** y el solver **'liblinear'** (tienes ejemplos en la unidad, aunque en las nuevas versiones de scikit-learn el parámetro ***multi_class='ovr'*** del método ***fit()*** de ***LogisticRegression*** está marcado como deprecated), por este motivo te pido que uses esta nueva clase. Debes pasarle un objeto estimador que en este caso es uno de regresión logística. Puedes acceder a ellos a través de ***objeto_ovr.estimators_***.

c) Imprime los atributos ***intercept_*** y **coef_** del modelo Lineal ¿Qué estás imprimiendo?

```
Intercepción: [-0.24594673764818956, 0.2792676834394041, -0.08219085411102253]
Coeficientes: [array([[-0.82741013,  0.98514359,  0.77193287, -0.54595872,  0.00648908,
         0.13691811,  1.30022534,  0.09528332, -0.33495873, -0.11386646,
        -0.13028017,  0.63452038,  0.0157244 ]]), array([[ 0.89877154, -1.29900046, -0.5105268 ,  0.2888685 , -0.01453144,
         0.31738496,  0.13892529,  0.26561357,  0.72130305, -1.50841937,
         0.71408719,  0.33816043, -0.01312294]]), array([[-3.89427350e-01,  6.27034347e-01,  1.33492326e-02,
         1.00444526e-01,  3.44156542e-02, -7.87534038e-01,
        -1.45192565e+00, -1.41907725e-01, -7.08051880e-01,
         9.24164343e-01, -5.05708733e-01, -1.15695064e+00,
        -1.49464247e-04]])]
```

***Nota**: como hay 3 posibles clases (1, 2 y 3) hay 3 estimadores diferentes, uno para cada clase.*

e) Entrena el modelo y muestra los valores de *la matriz de confusión* y las métricas de eficiencia (*accuracy, recall positivo y negativo, sensitivity* y *F1-score)* o alternativamente un informe de clasificación donde aparezcan.

```
Matriz de confusión:
 [[13  1  0]
 [ 1 17  0]
 [ 0  1 21]]
```

```
Informe de la clasificación:
              precision    recall  f1-score   support

           1
           2
           3

    accuracy
   macro avg
weighted avg
```

f) Muestra la ***curva ROC*** y ***el valor AUC*** de cada clase y del modelo en global. Para hacerlo, hay que binarizar las predicciones (generando una columna con valores 0/1 para las predicciones de cada clase) y obtener los ***scores*** (distancia de cada predicción a cada clase) con la función ***decision_function()***. Como posiblemente sea complicado, te paso el código para que lo añadas a tu programa y respondas ¿En qué clase obtiene mejor y peor resultado?

```
from sklearn.metrics import roc_curve, auc
import matplotlib.pyplot as plt
from sklearn.preprocessing import label_binarize
fpr = dict()
tpr = dict()
roc_auc = dict()
clases = df_vino['Class'].unique()                    # Diferentes clases: diferentes valores de Class
n_clases = clases.shape[0]                            # Cantidad de clases diferentes
y_test_bin = label_binarize(y_test, classes=[1, 2, 3])  # Binarizar el test
y_score = ovs.decision_function(X_test)               # Distancia a cada clase
for i in range(n_clases):
    fpr[i], tpr[i], _ = roc_curve(y_test_bin[:,i], y_score[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])
fpr["micro"], tpr["micro"], _ = roc_curve(y_test_bin.ravel(), y_score.ravel())    # micro-medias ROC y AUC
roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])
# Dibujar la curva ROC
plt.figure()
plt.plot(fpr["micro"], tpr["micro"], label='Curva ROC micro-average (área = {0:0.2f})'.format(roc_auc["micro"]))
for i in range(n_clases):
    plt.plot(fpr[i], tpr[i], label='Curva ROC de clase {0} (área = {1:0.2f})'.format(clases[i], roc_auc[i]))
plt.plot([0, 1], [0, 1], 'k--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('Ratio de Falsos Positivos')
plt.ylabel('Ratios de True Positivos')
plt.title('ROC para multi-class')
plt.legend(loc="lower right")
plt.show()
```

PASO 2: MEJORAR MODELO

Aunque el modelo funciona bastante bien usando regresión logística, vamos a intentar mejorarlo en este paso usando técnicas de selección de características. El objetivo es detectar que columnas podemos eliminar. En esta ocasión vamos a usar la regularización para detectar que características son prescindibles.

MEJORAR DATOS QUE ALIMENTAN AL ALGORITMO/MODELO

Y ahora vamos a entrenar un nuevo modelo usando la misma semilla aleatoria que antes, pero:

g) ¿Qué significa *estratificar los datos* al hacer la división en *train* + *test*? Añade en la llamada al método de particionar en *train* y test el parámetro `stratify=y`

h) Utiliza un objeto ***StandardScaler*** para estandarizar los datos de *train* y *test* antes de entrenar. ¿Porqué es interesante estandarizar los datos?

USAR REGULARIZACIÓN L1

Estamos usando un modelo con 13 predictoras. La regresión logística es propensa a sufrir de *overfitting* si hay muchas predictoras. Vamos a usar regularización ℓ1 para que deje a 0 las características menos importantes y al detectarlas poder eliminarlas del entrenamiento.

i) Indica que quieres usar regularización de tipo ℓ1 y deja el parámetro C= 1.0 ¿Qué ocurre si lo bajas, aumenta o disminuye la fuerza de la regularización?

j) Vuelve a imprimir los coeficientes y los puntos de intercepción.

k) Como hemos usado una regularización de tipo ℓ1, nos habrá anulado las características menos importantes. Vamos a aprovechar esto para mostrarlas ordenadas de mayor a menor importancia. Haz un listado por consola donde aparezca el nombre de la característica, su importancia con 4 decimales y la importancia acumulada (la suma de importancias desde la más a la menos importante, una especie de ratio de importancia si usamos desde la primera hasta la actual).

```
 0) Prolina                        importancia: 0.2067 Acumulada: 0.2067
 1) Flavanoides                    importancia: 0.1862 Acumulada: 0.3928
 2) Intensidad Color               importancia: 0.1524 Acumulada: 0.5452
 3) Alcohol                        importancia: 0.1272 Acumulada: 0.6724
 4) Posos                          importancia: 0.0914 Acumulada: 0.7638
 5) Saturación                     importancia: 0.0893 Acumulada: 0.8531
 6) Alcalinidad de posos           importancia: 0.0665 Acumulada: 0.9196
 7) OD280/OD315 de vinos diluidos  importancia: 0.0456 Acumulada: 0.9652
 8) Ácido Málico                   importancia: 0.0322 Acumulada: 0.9974
 9) Magnesio                       importancia: 0.0026 Acumulada: 1.0000
10) Proanticianinas                importancia: 0.0000 Acumulada: 1.0000
11) No flavanoides fenoles         importancia: 0.0000 Acumulada: 1.0000
12) Total Fenoles                  importancia: 0.0000 Acumulada: 1.0000
```

Para calcular la importancia de cada variable, miramos su coeficiente (la pendiente que tiene el hiperplano en esa variable, de manera que hacemos la suma del valor absoluto de las pendientes). Ten en cuenta que como hay 3 clases (3 productores posibles), los coeficientes de la recta son 3 filas y 13 columnas. Por simplificar, si hubiese 2 productores y 3 variables los coeficientes podían ser algo así:

```
[[-1  2,  3],
 [4, -5,  6]]
```

Hay que generar la suma del valor absoluto de las pendientes (coeficientes), en el ejemplo de arriba sería (lo calculo para que se comprendan las operaciones que debes hacer):

```
[1 + 4 , 2 + 5, 3 + 6] = [5, 7, 9]
```

Ahora calculamos la importancia relativa para ello calculamos la suma:

```
5 + 7 + 9 = 21
```

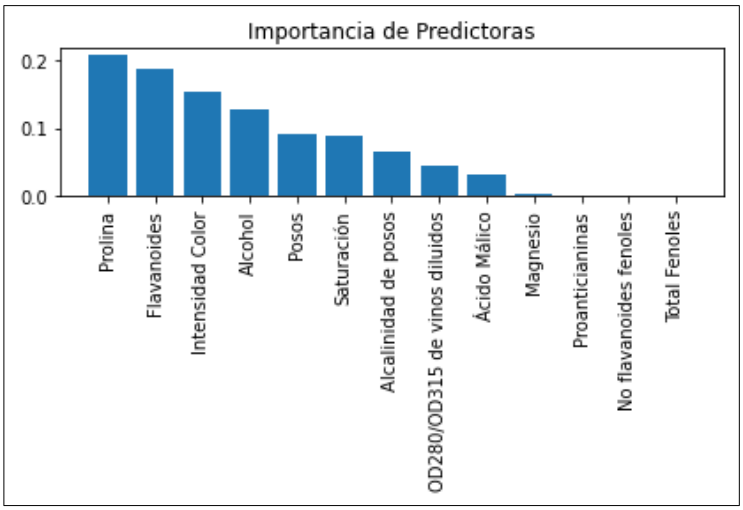
y dividimos cada elemento por el total (21 en el ejemplo):

```
[0.238, 0.333, 0.428]
```

Si ordenamos los valores o sacamos el índice de los elementos ordenados con

```
indice = np.argsort(pesos)  # Obtenemos [0,1,2] de menor a mayor, se invierte: [2, 1, 0]
```

l) Y hacemos un gráfico de barras además de imprimir los valores. El resultado aproximado que se debería obtener:



m) Ahora elimina las características predictoras que se te han quedado con cero influencia y compara el desempeño de los dos modelos. ¿Hay diferencias significativas?

📌ACTIVIDAD 3: CLASIFICADOR K-NN.

PASO 1: CARGA DE DATOS Y CREACIÓN DEL MODELO.

Crea el notebook ***saa_02_p04_a3_<tus_iniciales>.ipynb*** donde realizar esta actividad. Continuamos usando el fichero **`wine.csv`** para implementar ahora un clasificador KNN multiclase y tras construirlo volvemos a intentar mejorarlo eliminando características que no aporten demasiado, pero usando otro enfoque, porque ahora no podemos aplicar restricciones de tipo L1.

a) Copia el fichero ***saa_u02_p04_a2_<tus_iniciales>.ipynb*** en el fichero ***saa_u02_p04_a3_<tus_iniciales>.ipynb*** para cargar los datos de manera similar a como lo hiciste antes y completa el resto de apartados. Cuando particiones los datos y realices otras operaciones aleatorias vuelve a utilizar una semilla aleatoria que coincida con la longitud de tu nombre concatenada a la de tu primer apellido y concatenada a la de tu segundo apellido.

b) Continuamos usando un objeto ***multiclass.OneVsRestClassifier()*** para que utilice el ***método uno contra el resto*** y ahora un modelo de clasificación de tipo ***neighbors.NeighborsClassifier*** con el valor de ***k*** (parámetro ***n_neighbors***) que prefieras. Borra el código que imprime los parámetros de las líneas de los modelos de regresión porque ***k vecinos cercanos*** no usa línea (borra el apartado ***c)*** anterior).

c) Entrena el modelo y muestra los valores de *la matriz de confusión* y las métricas de eficiencia o alternativamente un informe de clasificación donde aparezcan tanto para los datos de *train* como para los datos de *test*.

d) Muestra la ***curva ROC*** y ***el valor AUC*** de cada clase y del modelo en global en los datos de *test*. Para hacerlo ten en cuenta que ahora este modelo no implementa la función ***decision_function()*** así que tendremos que utilizar ***predict_proba()***.

e) Responde a la vista de los resultados de los apartados d) y e) ¿Generaliza bien o tiene *overfitting*?

f) Borra el código del resto de apartados anteriores.

PASO 2: MEJORA DEL MODELO CON SBS.

La **selección secuencial de características (SBS)** es una familia de algoritmos de búsqueda de tipo *greedy*[1] (sistemáticos o ansiosos) que se utilizan para reducir la dimensionalidad ***d*** de un espacio de datos a una dimensión **`k`** donde ***k < d***. El objetivo es seleccionar las ***k*** características que sean más relevantes para el problema de entre las que hay originalmente. Esta técnica puede ser muy útil sobre todo para aquellos algoritmos que no soportan regularización.

Uno de estos algoritmos es el ***Sequential Backward Selection (SBS)***, que introduce un poco de sobrecarga para seleccionar estas características a cambio de mejorar mucho el rendimiento de su entrenamiento y funcionamiento.

La idea del algoritmo ***SBS*** es bastante simple: elimina características secuencialmente de los datos actuales hasta alcanzar el número de características deseado. Para decidir la característica a eliminar en cada etapa debemos usar una función criterio que llamamos ***J*** y que hay que minimizar.

El criterio calculado por la función ***J*** puede ser simplemente la diferencia de eficiencia del modelo antes y después de eliminar la característica. Así que en cada paso eliminamos la característica que menos pérdida de rendimiento genere. El algoritmo será:

1. Inicializar **`k = d`** donde ***d*** es la dimensionalidad de todo el espacio de características de **`X`**.
2. Encontrar la característica ***x–*** que maximiza el criterio ***x– = argmax J(X$_k$-x)*** donde ***x ϵ X$_k$***

1***Algoritmos de tipo Greedy****: hacen búsquedas escogiendo opciones localmente óptimas y en general consiguen encontrar soluciones no siempre óptimas a los problemas en un tiempo razonable, en contraste con los algoritmos exhaustivos que encuentran algunas de las soluciones óptimas de los problemas pero con un esfuerzo muchísimo mayor.*

3. Eliminar la característica ***x–*** del conjunto de características: $X_{k-1} = X_k - x-;\ k= k-1$
4. Terminar si **K** es el número de características deseadas o sino volver al paso 2.

g) Aunque está implementado en ***scikit-learn***, como es sencillo de hacer lo vamos a progrmar nosotros. Así que manos a la obra, crea el fichero ***sbs_<tus_iniciales>.py*** y copia el siguiente código:

```python
from sklearn.base import clone
from itertools import combinations
import numpy as np
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split

class SBS():
    def __init__(self, estimator, k_features, scoring=accuracy_score, test_size=0.25, random_state=1):
        self.scoring = scoring
        self.estimator = clone(estimator)
        self.k_features = k_features
        self.test_size = test_size
        self.random_state = random_state
```

```python
    def fit(self, X, y):
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=self.test_size,random_state=self.random_state)
        dim = X_train.shape[1]
        self.indices_ = tuple(range(dim))
        self.subsets_ = [self.indices_]
        score = self._calc_score(X_train, y_train, X_test, y_test, self.indices_) # desempeño con todas
        self.scores_ = [score]
        while dim > self.k_features:
            scores = []
            subsets = []
            for p in combinations(self.indices_, r=dim - 1):
                score = self._calc_score(X_train, y_train, X_test, y_test, p)
                scores.append(score)
                subsets.append(p)
            best = np.argmax(scores)
            self.indices_ = subsets[best]
            self.subsets_.append(self.indices_)
            dim -= 1
            self.scores_.append(scores[best])
        self.k_score_ = self.scores_[-1]
        return self
```

```python
    def transform(self, X):
        return X[:, self.indices_]

    def _calc_score(self, X_train, y_train, X_test, y_test, indices):
        self.estimator.fit(X_train[:, indices], y_train)
        y_pred = self.estimator.predict(X_test[:, indices])
        score = self.scoring(y_test, y_pred)
        return score
```

Para utilizarlo, un ejemplo:

```python
import matplotlib.pyplot as plt
from sklearn.neighbors import KneighborsClassifier
from SBS_josrosrod import SBS

knn = KneighborsClassifier(n_neighbors=5)
sbs = SBS(knn, k_features=1)
sbs.fit(X_train_std, y_train)
```

Aunque esta implementación de ***SBS*** ya divide internamente el dataset en *train* y test dentro de su función ***fit()***, externamente le proporcionamos como datos de trabajo el dataset *X_train*. Así es como si internamente la clase ***SBS*** crease una división *train* + *val* + *test* para prevenir usar los datos de test durante el entrenamiento o la búsqueda de parámetros, ya que ***SBS*** calcula *scores* de las mejores

características y si usamos el dataset test original podemos crear *overfitting* a esos datos de *test* y generar una medida de desempeño engañosa cuando usemos sus datos para comprobar si el modelo generaliza bien.

h) Como el objeto ***sbs*** ha ido coleccionando los ***scores*** de cada etapa, podemos graficar la evolución a medida que va quitando características. Adapta el siguiente código y lo añades a tu fichero (ten en cuenta que estás usando el objeto multiclase) y cuando lo ejecutes, a la vista del gráfico, ¿En qué rango de características funciona bien el modelo [desde, hasta]?

```
import matplotlib.pyplot as plt
from sklearn.neighbors import KneighborsClassifier
from sbs_josrodrod import SBS

knn = KNeighborsClassifier(n_neighbors=5)
sbs = SBS(knn, k_features=1)    # Le pedimos que pruebe hasta dejas solo 1
sbs.fit(X_train, y_train)
# Dibujamos como cambia el desempeño al cambiar el nº de características
k_carac = [len(k) for k in sbs.subsets_]

plt.plot(k_carac, sbs.scores_, marker='o')
plt.ylim([0.7, 1.02])
plt.ylabel('Accuracy')
plt.xlabel('Nº de características')
plt.grid()
plt.tight_layout()
plt.title("Desempeño vs nº características")
plt.show()
```

i) Si al clasificador ***KNN*** le afectan cosas, intenta minimizar estas cosas antes de realizar la división en *train* + *test*. Además, por prudencia (porque nos está dando una medida del desempeño agrupado en las 3 clases) vamos a quedarnos con la cantidad de características mayor que tenga un mejor desempeño, es decir, si entre [1, 4] tiene buen desempeño, nos quedamos con 4 en vez de con 1 por si a alguna clase le afecta demasiado no tener alguna de las que descartemos, pero esa pérdida se enmascara con buenos resultados en las otra clases. Así que nos quedaríamos con 4. Pero es necesario saber qué características son esas. Si adaptas este código en el que veríamos qué puedes saberlo suponiendo que hay 13 y queremos quedarnos con 3:

```
k3 = list(sbs.subsets_[10])
print(df_vinos.columns[1:][k3])
```

j) Ahora crea un nuevo clasificador pero descartando las categorías que el método anterior te indique que son descartables. Puedes adaptar el ejemplo. Calcula matriz, e informe de clasificación para *train* y test (mira si generaliza) y la curva *ROC*. Compara el desempeño de los dos modelos. ¿Hay diferencias significativas?

```
knn.fit(X_train[:, k3], y_train)
```

PASO 3: MEJORA DEL MODELO CON ENSEMBLES.

Otra aproximación para hacer el mismo trabajo de encontrar características que se puedan descartar es usar ***random forest***, un método *ensemble* que ya hemos comentado en esta unidad y que veremos con más detalle en la siguiente unidad. Al usar *random forest* suponemos que la importancia media de las características decrece cuando los cálculos se hacen a partir de todos los árboles del bosque.

En *scikit-learn* la implementación de ***random forest*** que hay ya recopila información sobre la importancia de cada característica cuando se construye el bosque y se almacena en la propiedad ***`feature_importances_`*** de un ***`RandomForestClassifier`***.

k) Adapta el siguiente código que utiliza un *random forest* de 500 árboles para averiguar la importancia de cada característica y haz un listado de mayor a menor importancia. ¿Coincide con alguno de los métodos anteriores que hemos usado?

```
from sklearn.ensemble import RandomForestClassifier

carac_labels = df_vinos.columns[1:]
forest = RandomForestClassifier(n_estimators=500, random_state=123)
forest.fit(X_train, y_train)
importancias = forest.feature_importances_
indices = np.argsort(importancias)[::-1]
```

PASO 4: BUSCAR EL MEJOR MODELO

Entrena estos clasificadores a ver si consigues mejorar el accuracy y el overfitting. No uses GridSearch, prefiero que uses manuealmente los modelos. Si quieres, si puedes usar validación cruzada.

- Regresión logística.
- SoftMax.
- Perceptrón.
- Bagging
- Boosting
- Voting de 3 modelos
- Stacking de 3 modelos (los que quieras)

ENTREGAR:

e) Código de entrenamiento de modelos y captura de ejecución de test, matriz de confusión, informe, curva ROC y AUC de cada uno. Guarda el mejor modelo a un fichero.