Módulo 5.3 Métodos avanzados en clasificación.

Autor:

Por Prof. Alberto Fernández Hilario

Profesor Catedrático de Universidad de Granada. Instituto Andaluz Interuniversitario en Data Science and Computational Intelligence (DasCI)

Breves Instrucciones

Recordatorio: Introducción a NoteBook

El cuaderno de *Jupyter* (Python) es un enfoque que combina bloques de texto (como éste) junto con bloques o celdas de código. La gran ventaja de este tipo de celdas, es su interactividad, ya que pueden ser ejecutadas para comprobar los resultados directamente sobre las mismas.

Muy importante: el orden de las instrucciones (bloques de código) es fundamental, por lo que cada celda de este cuaderno debe ser ejecutada secuencialmente. En caso de omitir alguna, puede que el programa lance un error (se mostrará un bloque salida con un mensaje en inglés de color rojo), así que se deberá comenzar desde el principio en caso de duda. Para hacer este paso más sencillo, se puede acceder al menú “Entorno de Ejecución” y pulsar sobre “Ejecutar anteriores”.

¡Ánimo!

Haga clic en el botón "play" en la parte izquierda de cada celda de código. Las líneas que comienzan con un hashtag (#) son comentarios y no afectan a la ejecución del programa.

También puede pinchar sobre cada celda y hacer "*ctrl+enter*" (*cmd+enter* en Mac).

Cuando se ejecute el primero de los bloques, aparecerá el siguiente mensaje:

"*Advertencia: Este cuaderno no lo ha creado Google.*

El creador de este cuaderno es <autor>@go.ugr.es. Puede que solicite acceso a tus datos almacenados en Google o que lea datos y credenciales de otras sesiones. Revisa el código fuente antes de ejecutar este cuaderno. Si tienes alguna pregunta, ponte en contacto con el creador de este cuaderno enviando un correo electrónico a <autor>@go.ugr.es.”

No se preocupe, deberá confiar en el contenido del cuaderno (Notebook) y pulsar en "*Ejecutar de todos modos*". Todo el código se ejecuta en un servidor de cálculo externo y no afectará en absoluto a su equipo informático. No se pedirá ningún tipo de información o credencial, y por tanto podrá seguir con el curso de forma segura.

Cada vez que ejecute un bloque, verá la salida justo debajo del mismo. La información suele ser siempre la relativa a la última instrucción, junto con todos los `print()` (orden para imprimir) que haya en el código.

ÍNDICE

En este *notebook*:

1. Se presentarán algoritmos de clasificación más sofisticados en aras de un mayor rendimiento predictivo.
2. Se describirán los modelos basados en Máquinas de Vectores Soporte, como una extensión de los modelos lineales.
3. Se comentarán detalles respecto a las soluciones basadas en redes neuronales y su extensión actual hacia el Deep Learning.
4. Se introducirán los algoritmos tipo "ensemble" (múltiples clasificadores) y en concreto se utilizará Random Forest como su mayor exponente.
5. Se analizarán las ventajas e inconvenientes de estos nuevos modelos.
6. Se mostrará cómo utilizar estos modelos avanzados mediante *Scikit-Learn* para Python.
7. Se discutirá sobre la influencia de los hiperparámetros en este tipo de clasificadores.

Contenidos:

1. INTRODUCCIÓN

En esta primera sección, se llevará a cabo una presentación sobre los paradigmas de clasificación avanzados que se describirán a lo largo del presente Módulo. A continuación, se cargarán y almacerán en variables de Python los datos del problema sobre melanoma cutáneo con los que se viene trabajando hasta ahora. Además, este conjunto de datos se transformará reduciendolo a solamente dos dimensiones (variables de entrada) para realizar representaciones gráficas que ilustren el funcionamiento de cada técnica o algoritmo de clasificación en Machine Learning.

1.1 Paradigmas de clasificación: modelos avanzados

Existen casos de estudio en los que se prima la máxima capacidad predictiva o acierto obtenido para la elección del modelo de clasificación. En este apartado, entrarían en juego los denominados "modelos de caja negra". Son algoritmos que obtienen un modelo de alto rendimiento, pero por contrapartida son complejos, es decir, no son directamente comprensibles por el usuario humano. En otras palabras, su comportamiento es robusto frente a problemas difíciles, aunque suelen implicar un mayor coste computacional en su aprendizaje (más recursos de CPU, memoria, más tiempo de ejecución, etc.), y necesitan un mayor nivel de conocimiento para su correcto uso.

Se observa por tanto que las características de los modelos de "caja blanca" descritos en el anterior apartado del curso, como Regresión Logística o Árboles de Decisión, y estas nuevas soluciones, son prácticamente opuestas. Por ese motivo, es importante determinar cuándo resulta más adecuado utilizar cada tipo de técnica de Machine Learning, así como conocer bien sus características para así utilizarlos correctamente.

En concreto, se describirán las Máquinas de Vectores Soporte (SVM) y los modelos tipo Ensemble, en concreto el conocido algoritmo Random Forest.

1.2 Cargar los datos del problema

Con el objetivo de comprobar el comportamiento de los diferentes algoritmos de clasificación, se comienza incorporando los datos del caso de estudio que sirve como hilo conductor del presente curso. La notación o código utilizado es exactamente el mismo que en actividades anteriores.

```
import pandas as pd

#Cargamos los datos ómicos de la matriz de expresión desde un fichero
        compartido en Google Drive
gene_exp_inmune = pd.read_csv('https://drive.google.com/uc?
        id=1PYzEIdmnfjOnBpPDIFBE9hL1Lkj_OBCk',index_col=0)
#Cargamos la variable clínica correspondiente a las etiquetas "inmune"
        vs. "MITF-low"
clinical_info_inmune = pd.read_csv('https://drive.google.com/uc?
        id=1hHQfcvrFa5Jds-9tW_X4sHjKpYKdii9s',index_col=0)


X, y = gene_exp_inmune, clinical_info_inmune

#Imprimimos las 5 primeras muestras del conjunto de datos para
        comprobar que se ha cargado correctamente
X.head()
```

	COL2A1	RXRG	CCL19	SSX1	CST2	PRSS33	CDH2	SCUBE2	TMPRS
0	-1.431141	-7.845756	0.665118	-1.409304	-2.537396	-1.676281	1.529957	-0.895042	-0.29877
1	-0.424374	-8.352423	0.386055	-2.846138	-0.685105	0.339787	-3.488043	-0.584982	5.679815
2	11.014251	0.415549	-1.633781	0.315442	-0.662332	-0.498761	0.535811	-0.467456	-2.81873
3	-1.180446	-8.187415	-1.958023	5.061146	-2.603744	-0.666706	0.456460	-4.609624	-1.71316
4	0.816312	-1.189303	4.837235	4.972176	-2.963715	-2.665721	-0.268042	-1.740607	0.011610

5 rows × 50 columns

Adicionalmente, tal como se realizó en la cápsula anterior ("**Métodos estándar de clasificación**"), para ilustrar el funcionamiento de cada clasificador, se procede a transformar el conjunto inicial de datos en un formato de 2 dimensiones (seleccionando únicamente dos variables de entrada). Es exactamente el mismo código que se incluyó en la cápsula anterior.

```
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
import seaborn as sns

#Transformamos el conjunto de datos inicial para que esté representado
        por solo 2 variables
n_componentes = 2
pca = PCA(n_components=n_componentes)
X_2D = pca.fit_transform(X)

#Se transforma el rango de cada variable a [0, 1]
st = StandardScaler()
X_2D = st.fit_transform(X_2D)

#Transformamos el conjunto resultando a una estructura data_frame
        (pandas) para facilitar la representación gráfica
df = pd.DataFrame(X_2D, columns = ['PCA_V1','PCA_V2'])
df = df.join(y)

#Mostramos en un diagrama de dispersión el nuevo conjunto 2D, junto
        con las funciones de densidad
sns.pairplot(df, hue='RNASEQ-CLUSTER_CONSENHIER', markers=["o", "s"],
        height=3);
```

Por simplicidad, para los ejemplos incluidos en este NoteBook se utilizará una validación tipo "*hold-out*" por defecto. Para más detalles consúltese el **Módulo 3** sobre Aprendizaje Supervisado.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y,
        random_state=42)


y_int = pd.get_dummies(y).iloc[:,0]
```

```
X_2D_train, X_2D_test, y_2D_train, y_2D_test = train_test_split(X_2D,
        y_int, random_state=42)

print("Numero de instancias en entrenamiento: {}; y test:
        {}".format(len(X_train),len(X_test)))
```

```
Numero de instancias en entrenamiento: 252; y test: 84
```

2. MÁQUINAS DE VECTORES SOPORTE (SVM)

Los modelos basados en **Máquinas de Vectores Soporte** (en inglés *Support Vector Machine o SVM*) son una de las herramientas preferidas por muchos científicos de datos. La razón es sencilla, ya que obtienen una alta precisión incluso en problemas complejos, utilizando relativamente poca potencia de cálculo. Adicionalmente, pueden ser empleados tanto para tareas de regresión como de clasificación, si bien es en esta última tarea donde mayor relevancia tiene.

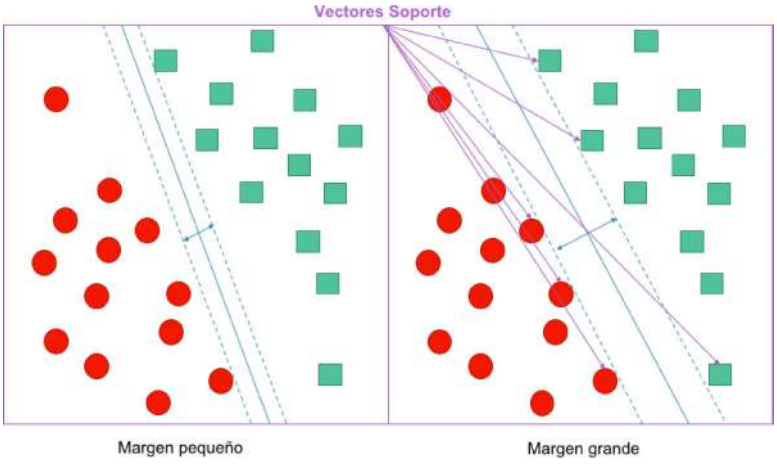
Este apartado, arranca con una introducción completa sobre las principales características y funcionamiento de las SVM. A continuación, se indica cómo hacer uso de este paradigma de clasificación mediante Scikit-Learn. Por último, se enumeran brevemente los pros y contras de este tipo de modelo.

2.1 Introducción a las SVM

El objetivo de una SVM es el de encontrar un hiperplano de separación entre las instancias de dos clases. Un hiperplano es exactamente el mismo tipo de función discriminante que se utilizó para otros clasificadores de tipo lineal, como por ejemplo la *regresión logística*. Recuérdese la ecuación del hiperplano, que no deja de ser un producto escalar de un vector de variables de entrada x con un vector de pesos o importancia de las mismas w $\|x, w\|$:

$$\hat{y}(x,w) = w_0 + w_1 \cdot x_1 + \ldots + w_n \cdot x_n$$

La diferencia de SVM con respecto a otros métodos de separación lineal, es que entre todos los posibles hiperplanos que dividen las instancias en dos partes, se escoge aquél que obtiene un margen máximo. Este margen se calcula como la máxima distancia entre las instancias frontera, tal como se representa en la siguiente figura:



El nombre de esta técnica de aprendizaje viene determinado justamente por estas instancias en las que se "apoya" la frontera de decisión. Son los puntos más cercanos al hiperplano, e influyen directamente en su orientación para alcanzar el citado margen máximo.

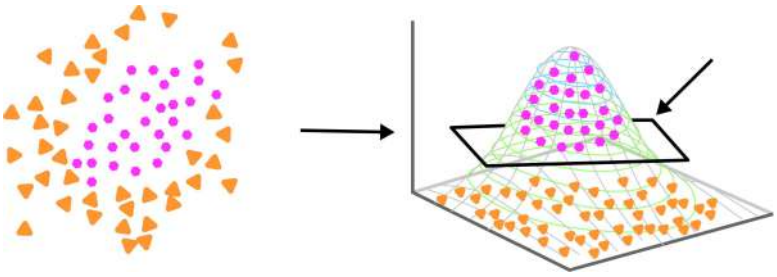
Existen diferentes mecanismos o aproximaciones matemáticas para encontrar la orientación óptima del hiperplano, si bien lo más importante es que dependen principalmente de la distancia de las instancias mal clasificadas (al otro lado de la

"frontera lineal"). Por este motivo, el parámetro más importante de una SVM es el **Coste**, denominado como `C`:

- Un valor bajo aceptaría cometer un cierto número de errores de clasificación, bajando ligeramente la calidad de predicción obtenida en el conjunto de entrenamiento, pero buscando una mejor generalización en test.
- Un valor alto permite ajustar mejor el modelo sobre los datos de entrenamiento, pero implicaría un mayor riesgo de sobreaprendizaje.

A pesar de lo anterior, un simple hiperplano no es la solución adecuada cuando las clases representadas en el problema no son linealmente separables (el problema es difícil), o cuando existe mucho ruido en los datos (se han cometido algunos errores en la captura de la información del problema). Sin embargo, se ha comentado que las SVMs son una herramienta que alcanza grandes resultados predictivos incluso en problemas complejos ¿cómo lo consigue?

La solución empleada se conoce como *kernel trick*, y consiste básicamente en asignar los datos en un espacio más complejo, con variables no lineales, y usar el clasificador SVM lineal (la aproximación básica anteriormente descrita) en este nuevo espacio. Por ejemplo, añadiendo una nueva dimensión o variable, podemos encontrar una separación adecuada de los datos, como se aprecia en la siguiente figura:



Para cambiar el espacio de datos de las variables de entrada, se utilizan las denominadas *funciones kernel* (de ahí el nombre *kernel trick*), que son las incrementan la dimensionalidad del problema ("añaden nuevas variables"), haciendo transformaciones matemáticas no lineales. Los dos ejemplos más comunes de este tipo de funciones kernel son las siguientes:

- Función polinomial: $K(x,w) = \langle x, w\rangle^d$. De este modo, la sumatoria lineal de los productos de pesos w_i y variables x_i se transforma a un polinomio de mayor grado (2, 3, etc.). Así, en lugar de un hiperplano o "línea recta" tendremos una división de los datos más compleja.
- Función *Radial Base Function* (RBF): $K(x,w) = e^{-\frac{\|x-w\|^2}{2\cdot\sigma}}$. Con el uso de funciones RBF las funciones discriminantes no-lineales en este caso se representan como áreas "circulares".

No existe una respuesta universal sobre qué tipo de kernel utilizar, dependiendo mucho de las características del problema. Los kernel polinomiales son más sencillos, con menor tendencia al sobreaprendizaje cuando el grado es pequeño (menor o igual a 5). Los kernel RBF obtienen un mayor acierto en general, puesto que encuentran funciones discriminantes más complejas que permiten separar mejor las clases. Para ello se utiliza un parámetro conocido como "gamma" (γ) que controla cómo se realiza la transformación del conjunto de datos, y por tanto valores altos tenderán al sobreaprendizaje.

Encontrar la combinación ideal entre los parámetros de coste `C` y del kernel `d` o γ es una tarea bastante complicada, para la que se suelen utilizar procedimientos de ajuste de hiperparámetros como Grid Search, cuyos detalles escapan a los objetivos del presente curso.

2.2 Implementación de las SVM en Python

Siguiendo el formato de todos los métodos de aprendizaje en *Scikit-Learn*, para ejecutar una SVM basta con crear un objeto de la clase `SVC` y ajustar utilizando los conjunto `X` e `y`, tal como se muestra a continuación:

```
from sklearn import svm
from sklearn.metrics import accuracy_score
import warnings
warnings.filterwarnings("ignore")

clf = svm.SVC(probability=True) #parámetro para que se pueda calcular
        AUC
clf.fit(X_train,y_train)

y_pred = clf.predict(X_test)
acc = accuracy_score(y_test,y_pred)
print("El porcentaje de acierto obtenido es",acc*100)
```

```
El porcentaje de acierto obtenido es 83.33333333333334
```

```
from sklearn import metrics
import matplotlib.pyplot as plt

metrics.ConfusionMatrixDisplay.from_estimator(clf, X_test,
        y_test,cmap='binary')
plt.title("Matriz de confusión obtenida para el clasificador SVM")
plt.show()

print(metrics.classification_report(y_test,y_pred))

f1 = metrics.f1_score(y_test,y_pred,pos_label='immune')
print("La medida F1 para el clasificador %s es %.4f"%
        (clf.__class__.__name__,f1))

y_probs = clf.predict_proba(X_test)
auc = metrics.roc_auc_score(y_test, y_probs[:,1])
print("La medida AUC para el clasificador %s es %.4f"%
        (clf.__class__.__name__,auc))
metrics.RocCurveDisplay.from_estimator(clf, X_test, y_test)
plt.title("Curva ROC obtenida para el clasificador SVM")
plt.show()
```

```
              precision    recall  f1-score   support

    MITF-low       0.84      0.84      0.84        44
      immune       0.82      0.82      0.82        40

    accuracy                           0.83        84
   macro avg       0.83      0.83      0.83        84
weighted avg       0.83      0.83      0.83        84

La medida F1 para el clasificador SVC es 0.8250
La medida AUC para el clasificador SVC es 0.9278
```

Se debe hacer énfasis en la especial importancia que tienen los parámetros de configuración (hiperparámetros) en el algoritmo SVM. A continuación, se recuerdan cuáles son los principales y su notación específica en *Python*:

- `C`: un valor real (`float`) que indica el parámetro de regularización o coste. Debe ser siempre positivo, donde mayores valores buscar un mejor ajuste a los datos de entrenamiento (por defecto=1.0).
- `kernel`: un valor entre {'linear', 'poly', 'rbf', 'sigmoid', 'precomputed'}, que especifica el tipo de kernel usado en el algoritmo (por defecto 'rbf').
- `degree`: un valor entero (`int`) que indica el grado del kernel polinominal (`poly`) por lo que se ignora para el resto (por defecto=3). Es el parámetro anteriormente notado como `d`.

- `gamma`: es un valor a elegir entre {'scale', 'auto'} o un número real (`float`). Es el coeficiente del kernel para los casos 'rbf', 'poly' y 'sigmoid' (por defecto='scale', que equivale a `1 / (n_features * X.var())`; siendo 'auto' `1 / n_features`).

Por último, se muestran las fronteras generadas de acuerdo a diferentes kernel y valores del parámetro. Varios aspectos llaman la atención en este punto:

1. Las fronteras representadas cuando se utiliza el kernel `rbf` suelen ser de tipo elíptico, mientras que para el polinomial (`poly`) presentan una tendencia de tipo "lineal".
2. Al incrementar el parámetro `C` (SVM con RBF) usualmente se observan las fronteras más pegadas a los puntos de entrenamiento.
3. Al incrementar el parámetro `degree` (SVM polinomial) se observan fronteras mucho más complejas.
4. La representación gráfica de las funciones discriminantes en ocasiones parece que no coinciden con la posición de las instancias del conjunto de datos. Esto es totalmente natural, dado que en realidad la frontera se visualiza directamente sobre las nuevas variables que se construyen para poder encontrar una función discriminante lineal de separación entre los datos.

Puesto que son muchos cálculos, puede que tarde unos segundos en mostrar los resultados.

```
#Se importa una biblioteca especial para pintar en 2D
from sklearn.inspection import DecisionBoundaryDisplay
import itertools

#Creamos y entrenamos los clasificadores SVM con los datos 2D
svm_rbf_1 = svm.SVC(kernel='rbf',C=1)
svm_rbf_100 = svm.SVC(kernel='rbf',C=100)
svm_poly_2 = svm.SVC(kernel='poly',degree=2,C=1)
svm_poly_5 = svm.SVC(kernel='poly',degree=5,C=1)
svms = [svm_rbf_1, svm_rbf_100, svm_poly_2, svm_poly_5] #lista

#Parámetros que se utilizarán para visualizar la figura
scatter_kwargs = {'s': 120, 'edgecolor': None, 'alpha': 0.7}
contourf_kwargs = {'alpha': 0.2}
scatter_highlight_kwargs = {'s': 120, 'label': 'Test data', 'alpha':
        0.7}

#Necesario para pintar las 4 gráficas juntas
#gs = gridspec.GridSpec(2, 2)
#fig = plt.figure(figsize=(15,12))

identificador = ['SVM RBF C1', 'SVM RBF C100', 'SVM Poly D2', 'SVM
        Poly D5']
for clf, etq, grd in zip(svms, identificador, itertools.product([0,
        1], repeat=2)):
    clf.fit(X_2D_train, y_2D_train)
    score = clf.score(X_2D_test,y_2D_test)
    #fig = plot_decision_regions(X=X_2D, y=y_int.to_numpy().ravel(),
        clf=clf,
    #                            X_highlight=X_2D_test, legend=2,
    #                            scatter_kwargs=scatter_kwargs,
    #                            contourf_kwargs=contourf_kwargs,
    #
        scatter_highlight_kwargs=scatter_highlight_kwargs)

    fig = DecisionBoundaryDisplay.from_estimator(clf, X_2D,

        response_method="predict",
                                                 alpha=0.5,
                                                 cmap=plt.cm.coolwarm)
    fig.ax_.scatter(X_2D[:, 0], X_2D[:, 1],
                    c=y_int, edgecolor="k",
                    cmap=plt.cm.coolwarm)

    plt.show()
```

2.3 Características de las SVM

Del mismo modo que el resto de paradigmas de clasificación, las SVMs poseen distintas propiedades que les hacen preferibles en distintos escenarios, o que pueden suponer complicaciones para su correcto uso.

En primer lugar, se enumeran las principales ventajas de SVM:

- Eficaces en problemas con una *alta dimensionalidad*. Obtienen buenas soluciones incluso cuando el número de variables de entrada es alto.
- Buen comportamiento en aquéllos casos en que el *número de variables es mayor que el número de instancias*. Esto resulta de vital importancia en problemas de carácter bioinformático.
- *Eficiente en memoria*. El modelo se basa únicamente en almacenar los "vectores soporte" encontrados en entrenamiento.
- *Versátil*. Se adapta bien a distintos problemas configurando la función kernel más apropiada en cada caso de estudio.

En segundo lugar, entre las desventajas de las SVM se incluyen:

- *Dificultad en la parametrización*. El comportamiento del modelo SVM es muy dependiente de los parámetros seleccionados para el *coste* y la función *kernel*. No es sencillo encontrar unos valores óptimos, y por tanto supone emplear una etapa denominada como "hiperparametrización" que tampoco garantiza los mejores resultados.
- *Tratamiento de datos nominales*. Igual que el resto de clasificadores lineales, no están adaptados a características que no sean numéricas. La solución es aplicar una transformación que genere nuevas variables mediante una codificación binaria.
- *Exclusividad en problemas binarios*. Por defecto, las SVMs no permiten una clasificación en problemas de más de dos clases, debiendo recurrir a estrategias tipo "divide-y-vencerás" (transformar el problema en múltiple subproblemas de clases binarias).
- *Cálculo poco preciso sobre las probabilidades de salida*. En caso que se necesite obtener valores de probabilidad o confianza para la salida, por ejemplo para calcular la métrica AUC, los valores que obtiene una SVM suelen ser poco fiables.

3. REDES NEURONALES ARTIFICIALES Y DEEP LEARNING

Las Redes Neuronales Artificiales (ANNs por sus siglas en inglés) están inspiradas en la forma en que funciona el cerebro humano. Muy simplificadamente, imagina que una ANN es como una red de pequeñas unidades llamadas neuronas que trabajan juntas para resolver problemas.

En las ANNs las neuronas se organizan de acuerdo a diferentes capas. Las primeras capas suelen ser para recibir información, como datos (variables) de entrada, y las últimas capas proporcionan la respuesta o resultado (ej. etiqueta de clase). Lo interesante sucede entre medias, en denominadas como las capas ocultas, como se estudiará más adelante. A medida que el número de capas ocultas aumenta, entramos en lo que se llama "Deep Learning" (DL). El DL permite que la red aprenda y represente características más complejas y abstractas, en especial para problemas con datos no estructurados como texto, imágenes o señales.

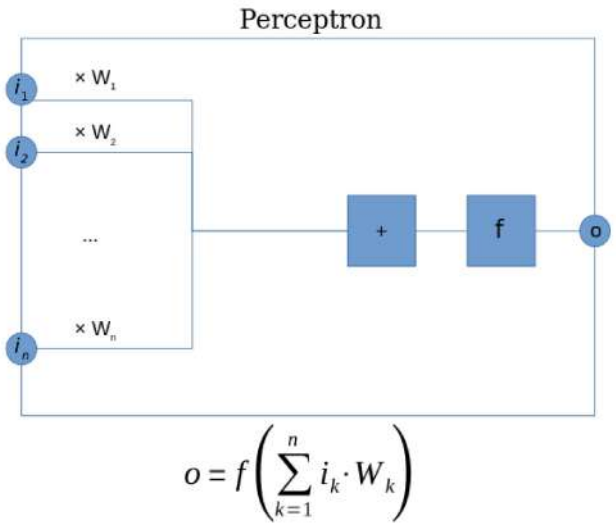
En el resto de esta sección, se introducirá con más detalle el concepto de neurona y capa, y se describirá cómo se realiza el entrenamiento de un modelo basado en ANNs. Posteriormente, se presentará cómo utilizar esta herramienta con Python

(vía `scikit-learn`). A continuación, se indicarán las características correspondientes a su uso. Finalmente, se darán algunos apuntes con respecto a la relación entre ANNs y DL.

3.1 ¿Qué es una Red Neuronal Artificial?

3.1.1 La neurona básica: el Perceptron

Todo arranca a principios de los años 40 con la invención del que será denominado como unidad básica de procesamiento en una red neuronal: el **Perceptrón**. Matemáticamente, el perceptrón toma un conjunto de entradas (i_1, i_2, ..., i_n), aplica pesos a esas entradas y produce una salida (o). Esta salida se obtiene aplicando una función de activación a la suma ponderada de las entradas y los pesos. La función de activación puede ser una simple función escalón, que dispara si la suma ponderada supera un cierto umbral.



$$o = f\left(\sum_{k=1}^{n} i_k \cdot W_k\right)$$

Si esta estructura te recuerda a algo, estás en lo cierto: un perceptrón puede considerarse como un tipo de clasificador lineal básico (como los vistos en el Módulo 5.2). Al igual que el clasificador lineal, su capacidad es limitada cuando se trata de datos más complejos que no son linealmente separables. Es aquí donde entran en juego las ANNs ampliando el número de capas y neuronas utilizadas.

3.1.2 Las capas ocultas

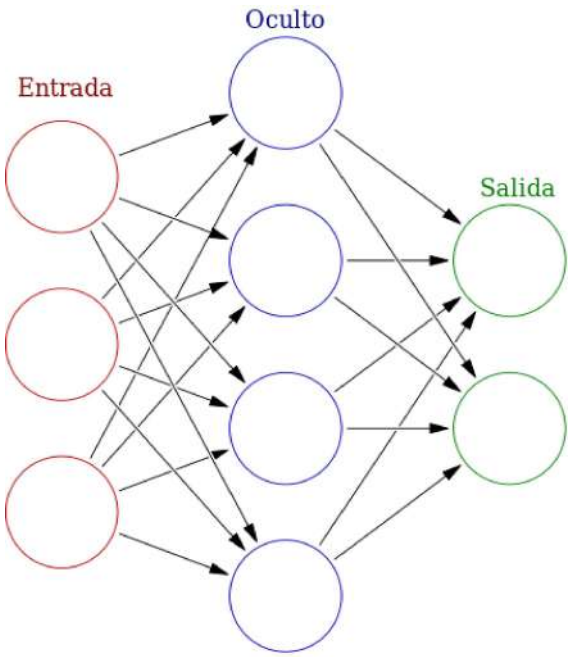
Una red neuronal se puede definir como una colección de perceptrones organizados en capas (bloques de neuronas). Además de la capa de entrada (cuyo número de neuronas coincide con el número de entradas) y la capa de salida (que normalmente es una única neurona para una salida binaria), puede haber capas intermedias llamadas **capas ocultas**.

La capa oculta es donde ocurre la magia en una red neuronal. Cada nodo o "neurona" en esta capa toma las salidas de perceptrones en la capa anterior (puede ser directamente la entrada, u otra capa oculta), aplica sus pesos, agrega los resultados y pasa el resultado a través de lo que se denomina una **función de activación**. Esto permite que la red aprenda y capture características y patrones complejos (no lineales) en los datos. En otras palabras, representan características intermedias o abstracciones de las entradas.

Así, en este caso estaríamos considerando el denominado como Perceptrón Muti-capa o **Multi-Layer Perceptron (MLP)**. En la siguiente figura se ilustra un MLP con tres capas:

- la primera es la capa de entrada con 3 neuronas, lo que corresponde a un problema reprentado por 3 variables descriptivas;
- la segunda corresponde con la cada oculta, con 4 neuronas en total;
- la última corresponde con la capa de salida, en este caso con dos neuronas que pueden codificar hasta 4 valores diferentes.

Nótese que todas las neuronas de una capa se conectan con todas las neuronas de la siguiente capa, siguiendo un esquema denominado como **fully-connected**.



3.1.3 Relación con la Función de Activación

Tal como se ha indicado, cada neurona en la capa oculta aplica una función matemática a su resultado. Esta función de activación determina si la neurona debe "disparar" (enviar una señal) o no, en función de la suma ponderada de sus entradas. Algunas de las funciones de activación más comunes son:

- **ReLU (Rectified Linear Unit)**: $f(x) = max(0,x)$
- **Sigmoid**: $f(x) = \frac{1}{1 + e^{(-x)}}$
- **Tanh (Tangente hiperbólica)**: $f(x) = \frac{e^x - e^{(-x)}}{e^x + e^{(-x)}}$

La elección de la función de activación puede influir en cómo la red aprende y se comporta. Por ejemplo, la función `ReLU` es comúnmente utilizada porque es eficiente y evita ciertos problemas en el aprendizaje. La función `sigmoid` es útil en problemas de clasificación binaria, mientras que la función `tanh` es útil en situaciones en las que las entradas tienen valores negativos y positivos.

3.2 Entrenamiento mediante retroalimentación (backpropagation): Un Viaje de Aprendizaje

Entrenar una ANN es un proceso fascinante que se asemeja al aprendizaje humano basado en ejemplos. Imagina a una ANN como un estudiante que mejora su desempeño a través de la práctica constante. En concreto, lo que realmente se ajusta en la estructura de una ANN son los valores de los pesos en las conexiones entre neuronas. Esto es lo que se denominan los **parámetros de la red**. A mayor número de parámetros, mayor será la complejidad de la red para aprender problemas más difíciles, pero a mayor coste computacional.

A continuación, se explora cómo funciona este proceso de entrenamiento, haciendo hincapié en términos sencillos y conceptos clave.

3.2.1 Paso 1: Inicialización y Pesos Aleatorios

Se comienza con una red neuronal vacía (solo la estructura de neuronas en capas). Al igual que un estudiante en su primer día de clase, no sabe nada. Así, los pesos de las conexiones entre las neuronas se inicializan aleatoriamente.

3.2.2 Paso 2: Alimentación Hacia Adelante (*Forward Propagation*)

Como estudiantes, la ANN recibe datos de entrenamiento (el conjunto de datos completo). La información se propaga a través de la red desde la capa de entrada hasta la capa de salida. Cada neurona realiza cálculos matemáticos en función de los datos de entrada y los pesos de las conexiones, a través de la conocida función de activación. Estos cálculos producen una respuesta de la red (valor decodificado de la(s) neurona(s) de salida), que en principio será la etiqueta predicha de clase.

3.2.3 Paso 3: Comparación con la Respuesta Deseada

La red compara su respuesta con la respuesta deseada (la respuesta correcta). Similar a cómo un estudiante compara sus respuestas en un examen con las respuestas correctas. Esta comparación genera un "error", que mide qué tan cerca o lejos está la red de la respuesta correcta. Este error se mide mediante lo que se denomina como función de pérdida o **loss function**. Un ejemplo trivial de *loss function* podría ser 1 – *accuracy*.

3.2.4 Paso 4: Retropropagación del Error (*Backpropagation*)

Aquí entra en juego la "retroalimentación". La red ajusta sus conexiones (pesos) en función del error. Si la respuesta es incorrecta, la red modifica sus conexiones de manera que la próxima vez se acerque más a la respuesta deseada. Este proceso se llama "backpropagation" y se asemeja al aprendizaje basado en retroalimentación que los estudiantes hacen para mejorar.

3.2.5 Tasa de Aprendizaje (*Learning Rate*):

Imagina que el estudiante tiene una tasa de aprendizaje, que determina cuánto cambia sus respuestas en función de sus errores. Si la tasa de aprendizaje es alta, aprende rápidamente pero puede ser *inestable*. Si es baja, aprende lentamente pero con mayor precisión.

3.2.6 Iteración y Práctica Constante:

Este proceso se repite muchas veces. La red sigue viendo ejemplos, cometiendo errores y ajustándose; los ejemplos de entrada van incorporándose a la red en lotes, y este proceso se repite haciendo varias "pasadas" al conjunto de datos. Es como un estudiante que sigue practicando los mismos ejercicios una y otra vez para mejorar sus habilidades.

Con cada ciclo de entrenamiento, la ANN se vuelve más y más competente, como un estudiante que se convierte en un experto en una materia. A través del backpropagation, la tasa de aprendizaje y la regularización, la red neuronal aprende a resolver problemas de manera efectiva. Así, podemos identificar la denominada *convergencia de la red*, es decir su efectividad en el correcto ajuste al problema en cada ciclo.

3.3 Cálculo de la salida de una red neuronal y Extensión a Múltiples Clases

Hasta este punto, hemos hablado principalmente de redes neuronales con una sola neurona de salida, lo que permite codificar un valor binario {0, 1} según supere o no un umbral de activación, por defecto mediante la función sigmoide. Esto las hace ideales para problemas de decisión dicotómicas, como diagnosticar enfermedades como presentes o ausentes.

En estos casos, la red neuronal ajusta su respuesta para acercarse a 0 o 1, lo que equivale a decir "sí" o "no". Sin embargo, como bien sabemos, en el mundo real a menudo nos enfrentamos a problemas más complejos donde las respuestas no son simplemente "sí" o "no", sino que pueden pertenecer a varias categorías.

En tales casos, utilizamos un número más alto de neuronas junto con funciones de activación especiales para extender la salida de una red neuronal a múltiples clases. Una función común es la función **"Softmax"**. Esta función calcula la

probabilidad de que una muestra pertenezca a cada clase, y la suma de estas probabilidades es igual a 1.

Así que, la salida de la red se extiende para acomodar diferentes clases, y las funciones de activación como `Softmax` permiten asignar probabilidades a cada clase. Esta es la magia que permite a las redes neuronales manejar problemas de múltiples clases, como reconocimiento de dígitos escritos a mano o clasificación de imágenes en categorías diversas.

3.2 Implementación de las redes neuronales en Python

En esta sección se dan los detalles necesarios para construir un modelo basado en el perceptrón multi-capa (`MLPClassifier`), que se encuentra en la biblioteca `neural_network`.

En este ejemplo, se creará una red neuronal con una única capa oculta de 10 neuronas, y se entrenará la red durante 1000 iteraciones. Se pueden ajustar estos valores según las necesidades de cada problema, tal como se describirá más abajo.

```
from sklearn.neural_network import MLPClassifier

# Creamos y entrenamos la red neuronal (random_state se usa para
        ofrecer los mismos resultados cada vez que se ejecuta)
clf = MLPClassifier(hidden_layer_sizes=(10,), max_iter=1000,
        random_state=42)
clf.fit(X_train, y_train)


#Probamos su calidad predictiva
y_pred = clf.predict(X_test)
acc = accuracy_score(y_test,y_pred)
print("El porcentaje de acierto obtenido es",acc*100)
```

```
El porcentaje de acierto obtenido es 80.95238095238095
```

```
metrics.ConfusionMatrixDisplay.from_estimator(clf, X_test,
        y_test,cmap='binary')
plt.title("Matriz de confusión obtenida para el clasificador ANN")
plt.show()


print(metrics.classification_report(y_test,y_pred))


f1 = metrics.f1_score(y_test,y_pred,pos_label='immune')
print("La medida F1 para el clasificador %s es %.4f"%
        (clf.__class__.__name__,f1))


y_probs = clf.predict_proba(X_test)
auc = metrics.roc_auc_score(y_test, y_probs[:,1])
print("La medida AUC para el clasificador %s es %.4f"%
        (clf.__class__.__name__,auc))
metrics.RocCurveDisplay.from_estimator(clf, X_test, y_test)
plt.title("Curva ROC obtenida para el clasificador ANN")
plt.show()
```

```
              precision    recall  f1-score   support

    MITF-low       0.75      0.95      0.84        44
      immune       0.93      0.65      0.76        40

    accuracy                           0.81        84
   macro avg       0.84      0.80      0.80        84
weighted avg       0.84      0.81      0.80        84

La medida F1 para el clasificador MLPClassifier es 0.7647
La medida AUC para el clasificador MLPClassifier es 0.8330
```

```
#Creamos y entrenamos el clasificador con los datos 2D
clf = MLPClassifier(hidden_layer_sizes=(10,), max_iter=1000,
        random_state=42)
clf.fit(X_2D_train, y_2D_train)
score = clf.score(X_2D_test,y_2D_test)

fig = plt.figure(figsize=(12,9))
fig = DecisionBoundaryDisplay.from_estimator(clf, X_2D,

        response_method="predict",
                                                alpha=0.5,
                                                cmap=plt.cm.coolwarm)

fig.ax_.scatter(X_2D[:, 0], X_2D[:, 1],
                c=y_int, edgecolor="k",
                cmap=plt.cm.coolwarm)

plt.show()
```

```
<Figure size 1200x900 with 0 Axes>
```

Al igual que pasaba en las SVMs, los hiperparámetros de las ANNs tienen una influencia significativa en el comportamiento y capacidad predictiva del modelo, es decir, cuán bien o mal se ajustará a los datos para reproducir fielmente la información del problema. De este modo, encontrar un buen equilibrio entre infra-entrenamiento y sobre-entrenamiento es bastante complejo, y se suelen utilizar procedimientos de optimización tipo `GridSearch` como en el caso de las SVMs.

Cuando se utiliza una ANN, existe un alto número de variables de configuración (hiperparámetros) que ajustar, así que a continuación se muestran únicamente aquellos más importantes:

- `hidden_layer_sizes`: Esta es una configuración que representa el número de neuronas en cada capa oculta de una red neuronal. Por ejemplo, si se desea aplicar una sola capa oculta con 100 neuronas, se configuraría como `hidden_layer_sizes=(100,)` (este es el valor que se toma por defecto).
- `activation`: Como se indicó anteriormente, la función de activación es una parte importante de una red neuronal. Es la función matemática que determina si una neurona debe disparar o no. Hay varias opciones comunes, como `'identity'`, que no hace nada (es decir, no activa ni desactiva la neurona), `'logistic'`, que utiliza la función sigmoide, `'tanh'`, que utiliza la tangente hiperbólica, y `'relu'`, que es la función de unidad lineal rectificada (que es la que se toma por defecto).
- `learning_rate_init`: Esta es la tasa de aprendizaje inicial. Controla el tamaño de los pasos que la red toma al ajustar sus pesos (por defecto `learning_rate_init=0.001`).
- `max_iter`: Es el número máximo de iteraciones que la red realizará durante el entrenamiento. Si no converge antes de alcanzar este número, el entrenamiento se detendrá (por defecto `max_iter = 200`).

```
#Veamos la influencia de algunos de estos parámetros en los resultados
        (tardará algunos segundos en ejecutar)
import numpy as np

# Parámetros que vamos a variar
num_layers = [1, 2, 3]
num_neurons = [5, 10, 15]
learning_rates = [0.001, 0.01, 0.1]
max_iterations = [100, 500, 1000]

# Almacenaremos los resultados del accuracy
accuracies = []
```

```
# Iteramos a través de diferentes configuraciones
for layers in num_layers:
    for neurons in num_neurons:
        for rate in learning_rates:
            for iterations in max_iterations:
                clf = MLPClassifier(hidden_layer_sizes=(neurons,) *
        layers, max_iter=iterations, learning_rate_init=rate,
        random_state=42)
                clf.fit(X_train, y_train)
                accuracy = clf.score(X_test, y_test)
                accuracies.append((layers, neurons, rate, iterations,
        accuracy))

# Extraemos los valores de precisión
accuracies = np.array(accuracies)

# Mostramos los resultados gráficamente
plt.figure(figsize=(10, 6))
plt.scatter(accuracies[:, 0] * accuracies[:, 1], accuracies[:, 4],
        c=accuracies[:, 2], cmap='viridis')
plt.xlabel('Número de Neuronas (Total)')
plt.ylabel('Accuracy')
plt.title('Influencia de los Hiperparámetros en MLP (Neuronas vs LR)')
plt.colorbar(label='Tasa de Aprendizaje')
plt.show()

# Mostramos los resultados gráficamente
plt.figure(figsize=(10, 6))
plt.scatter(accuracies[:, 0] * accuracies[:, 1], accuracies[:, 4],
        c=accuracies[:, 3], cmap='viridis')
plt.xlabel('Número de Neuronas (Total)')
plt.ylabel('Accuracy')
plt.title('Influencia de los Hiperparámetros en MLP (Neuronas vs
        Iteraciones)')
plt.colorbar(label='#Iteraciones')
plt.show()
```

3.3 Características de las Redes Neuronales Artificiales

Las ANNs tienen sus propias ventajas y desventajas que las hacen adecuadas para diferentes situaciones en biología y otras disciplinas. A continuación, se exponen una serie de propiedades de interés, distinguiendo entre potencialmente positivas y negativas.

3.3.1 Ventajas de las ANNs

1. **Capacidad para Aprender Modelos No Lineales:** Una de las ventajas clave de las RNA es su habilidad para aprender y modelar relaciones no lineales en los datos. Esto las hace efectivas para tareas de clasificación en las que las fronteras de decisión son complejas o irregulares.

2. **Flexibilidad en Tareas de Clasificación y Regresión:** Las ANN pueden aplicarse a una variedad de tareas, desde la clasificación de especies de plantas hasta la predicción de tasas de crecimiento. Son versátiles y pueden adaptarse a diferentes tipos de problemas.

3. **Capacidad para Extraer Características Relevantes:** Las redes neuronales pueden aprender características útiles directamente de los datos, lo que reduce la necesidad de realizar una extracción manual de características.

4. **Lidiar con Datos Ruidosos:** Las redes neuronales son robustas y pueden manejar datos ruidosos o con errores, lo que es común en aplicaciones del mundo real.

5. **Aplicación en Diversos Dominios y Problemas Multimodales:** Las redes neuronales se han utilizado con éxito en una amplia variedad de dominios, desde reconocimiento de voz y visión por computadora hasta biología y finanzas. Además, para problemas que involucran múltiples modos de entrada (por ejemplo, imágenes y texto), las redes neuronales pueden fusionar y modelar estas modalidades de manera efectiva.

6. **Adaptación en Tiempo Real:** Las redes neuronales pueden aprender y ajustarse continuamente a medida que llegan nuevos datos. Esto las hace adecuadas para aplicaciones en las que los patrones cambian con el tiempo (aprendizaje continuo).

3.3.2 Desventajas de las ANNs

1. **Función de Pérdida No Convexa:** Las ANN con capas ocultas tienen una función de pérdida que no es convexa. Esto, en sentido matemático, significa que existen varios mínimos locales en esta función. Como resultado, diferentes inicializaciones aleatorias de pesos pueden llevar a diferentes niveles de precisión en la validación. De ahí que encontremos un comportamiento relativamente inestable y muy dependiente de la selección de la semilla de inicio.

2. **Ajuste de Hiperparámetros Necesario:** Para obtener buenos resultados con ANN, es necesario ajustar varios hiperparámetros, como el número de neuronas en capas ocultas y el número de iteraciones. Encontrar la configuración óptima puede requerir una búsqueda exhaustiva, lo que se denomina hiperparametrización.

3. **Sensibilidad a la Escala de Características:** Las ANN son sensibles a la escala de las características. Por lo tanto, se recomienda encarecidamente escalar los datos antes de usar una ANN. La escala adecuada puede variar según el conjunto de datos y debe aplicarse tanto al entrenamiento como a los datos de prueba para obtener resultados precisos. Cualquier técnica de escalado (`Min-Max` o `StandardScaler`) puede ser útil en este caso.

4. **Problemas con Datos No Numéricos:** Al igual que otros modelos de Machine Learning, las ANN no están diseñadas para manejar directamente características no numéricas, como nombres o etiquetas (categóricos). Estas características generalmente requieren transformación antes de usar una ANN (por ejemplo mediante codificación `One-Hot`).

5. **Requiere Tamaño de Muestra Grande:** Aunque las ANN son capaces de aprender patrones complejos, su desempeño mejora con un gran conjunto de datos. En casos con conjuntos de datos muy pequeños, pueden no ser la mejor opción.

6. **Potencialmente Lento en Grandes Conjuntos de Datos:** En comparación con otros algoritmos, las ANN pueden ser relativamente lentas en conjuntos de datos muy grandes, especialmente cuando se incrementa el número de iteraciones necesario para asegurar una convergencia adecuada (con una baja tasa de aprendizaje).

3.4 El Vínculo Entre Redes Neuronales y Deep Learning**

El estudio de las ANN y, en particular, de los MLP, sirve como una base sólida para comprender el paradigma más amplio del DL. Aunque existen diferencias específicas entre estos dos enfoques, el conocimiento adquirido en los primeros es altamente extensible al segundo, con algunas consideraciones importantes.

1. Jerarquía de Capas: Tanto las ANN como el DL se basan en la idea de organizar capas de neuronas o nodos para procesar datos. En las ANN, esto a menudo se limita a una o dos capas ocultas, mientras que en el DL, se emplean múltiples capas ocultas.

2. Representación Jerárquica: El aprendizaje jerárquico de características es un concepto fundamental en DL. A medida que avanzamos en capas más profundas de una red, se espera que las características aprendidas sean cada vez más abstractas y representen conceptos de nivel superior.

3. Funciones de Activación: Las funciones de activación, como la sigmoide y ReLU, son elementos comunes tanto en las ANN como en DL.

4. Backpropagation: El algoritmo de retropropagación de errores es la base del entrenamiento en las ANN. En DL, este algoritmo se mantiene, pero a menudo se combina con técnicas más avanzadas, como el descenso de gradiente estocástico (SGD) con variantes y ajustes más sofisticados.

5. Redes Convolucionales (CNN) y Redes Recurrentes (RNN): En el DL, se extiende el concepto de redes neuronales más allá de las capas completamente conectadas. Las CNN se utilizan para datos con estructura espacial, como imágenes, mientras que las RNN se emplean para datos secuenciales, como texto y series temporales.

6. Problemas de Gran Escala: Las ANN y MLP son efectivas en una variedad de aplicaciones, pero el DL ha demostrado ser especialmente exitoso en problemas de gran escala, como el reconocimiento de imágenes, el procesamiento de lenguaje natural y el aprendizaje por refuerzo.

4. ENSEMBLES Y RANDOM FOREST

Los algoritmos posiblemente más utilizados en Ciencia de Datos son aquéllos basados en múltiples clasificadores, generalmente conocidos como "ensembles".

En esta sección, se introduce en qué consiste esta tipología de sistemas de aprendizaje. Posteriormente, el foco se centra en uno de las aproximaciones más conocidas de este tipo de métodos: el algoritmo Random Forest. Por último, se indican algunas de las ventajas e inconvenientes relativos a este paradigma.

4.1 Introducción al paradigma de ensembles

Hasta el momento, se ha construido un modelo único a partir de los datos del problema, y éste ha permitido realizar una predicción más o menos acertada con respecto al conjunto de test. No obstante, en la actividad cotidiana es frecuente contar con el criterio de distintas fuentes a la hora de tomar decisiones. Agregar o combinar las "opiniones" de un grupo de expertos parece ser la clave para incrementar la confianza en una predicción concreta.

Para que el anterior escenario sea realmente válido, existen dos premisas muy importantes:

- Las decisiones/predicciones deben ser realizadas a partir de fuentes o expertos que tengan cierta credibilidad. De esta forma, se garantiza una calidad adecuada en la respuesta.
- El grupo de expertos o colección de fuentes debería ser diversa entre sí. Así, se promueve cierto nivel de objetividad.

En definitiva, en caso de poder contar con la opinión de especialistas que presenten puntos de vista complementarios, ayudará a aumentar la calidad y fiabilidad de la decisión tomada.

Esta premisa también tiene su aplicación directa en el Machine Learning, tomando la denominación de "*modelos tipo Ensemble*". En esta familia de algoritmos, se construye un número relativamente grande de clasificadores, utilizando para ello dos enfoques:

- **Bagging**: Usar un subconjunto diferente de los datos de entrenamiento. En este caso, se entrenan `M` clasificadores de manera independiente.
- **Boosting**: Usar pesos o costes para los ejemplos más difíciles de identificar correctamente. En este caso, se realizan `M` iteraciones, en cada una generando un clasificador dependiente del resultado de la etapa anterior.

Este apartado del curso se centra exclusivamente en la primera de las estrategias, es decir, **bagging**.

4.2 Random Forest: Fundamentos e implementación en

Python

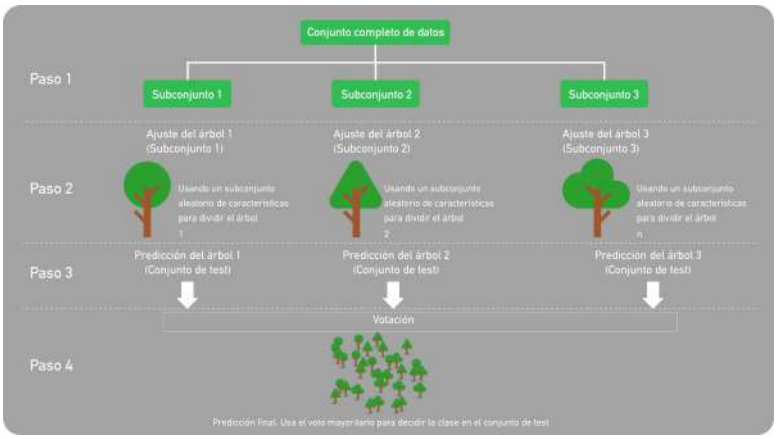
Para comprender correctamente en qué se fundamenta el algoritmo Random Forest, hay que definir con detalle la metodología de **bagging**. Si se tiene un conjunto de datos *X* de tamaño *N*, se podría generar un nuevo conjunto tomando *N* instancias de manera aleatoria, con **reemplazamiento**. Si este procedimiento se repite *M* veces, se obtiene un número suficiente de conjuntos para que su "combinación" represente la distribución original de los datos.

En el caso de la tarea de aprendizaje supervisado, se crearán *M* nuevos conjuntos de entrenamiento, todos del tamaño original, pero con algunas de las instancias repetidas en cada nuevo conjunto. De este modo, se pueden aprender *M* clasificadores independientes, cada uno especializado en las instancias que usadas en su entrenamiento.

Al ser cada modelo relativamente independiente entre sí, y generando cada uno un valor de salida que puede diferir, la pregunta principal que surge en este momento es ¿cómo se unifica la respuesta del modelo global? En el caso de *clasificación* la más común es utilizar un voto por mayoría.

Random Forest es un algoritmo de aprendizaje utilizado indistintamente para clasificación o regresión, y que se basa en el anterior principio de **bagging**. En este caso, el nombre *forest* (bosque en inglés) permite deducir cuál es la familia de modelos que se entrenan en cada submuestra del conjunto de entrenamiento: los **árboles de decisión**.

Pero este algoritmo añade una nueva componente a la metodología **bagging**, y es la de realizar también una selección de variables de entrada en cada subconjunto de entrenamiento. De esta forma, se obtiene una doble ganancia. Por un lado, se incrementan las diferencias entre cada conjunto, y por tanto entre cada modelo generado. Por otro, el tiempo de aprendizaje y la complejidad de cada modelo se ve reducida.



Retomando las dos características principales a cumplir para la metodología tipo Ensemble, en Random Forest se obtiene lo siguiente:

- Los clasificadores utilizados se basan en árboles de decisión, y por tanto resultan muy apropiados a la hora de realizar el aprendizaje.
- La selección de instancias, junto con la selección de variables, permite que la construcción de cada árbol individual sea relativamente distinto a la de los demás modelos, donde la unión hace la fuerza.

Utilizar el algoritmo Random Forest en *Scikit-Learn* es tan sencillo como cualquier otro clasificador, como se muestra el siguiente ejemplo:

```
from sklearn.ensemble import RandomForestClassifier

rf = RandomForestClassifier(random_state=42)
rf.fit(X_train, y_train)

y_pred = rf.predict(X_test)
acc = accuracy_score(y_test,y_pred)
print("El porcentaje de acierto obtenido por RF es",acc*100)
```

```
El porcentaje de acierto obtenido por RF es 91.66666666666666
```

```
metrics.ConfusionMatrixDisplay.from_estimator(rf, X_test,
        y_test,cmap='binary')
plt.title("Matriz de confusión obtenida para el clasificador RF")
plt.show()
```

```
print(metrics.classification_report(y_test,y_pred))
```

```
f1 = metrics.f1_score(y_test,y_pred,pos_label="immune")
print("La medida F1 para el clasificador %s es %.4f"%
        (rf.__class__.__name__,f1))
```

```
y_probs = rf.predict_proba(X_test)
auc = metrics.roc_auc_score(y_test, y_probs[:,1])
print("La medida AUC para el clasificador %s es %.4f"%
        (rf.__class__.__name__,auc))
metrics.RocCurveDisplay.from_estimator(rf, X_test, y_test)
plt.title("Curva ROC obtenida para el clasificador RF")
plt.show()
```

```
              precision    recall  f1-score   support

    MITF-low       0.91      0.93      0.92        44
      immune       0.92      0.90      0.91        40

    accuracy                           0.92        84
   macro avg       0.92      0.92      0.92        84
weighted avg       0.92      0.92      0.92        84

La medida F1 para el clasificador RandomForestClassifier es 0.9114
La medida AUC para el clasificador RandomForestClassifier es 0.9716
```

La implementación de *Scikit-Learn* de Random Forest tiene una doble ventaja muy interesante.

1. Por un lado, permite identificar cuáles son las variables de entrada más importantes utilizados para predecir la salida. Para ello, se debe consultar una de las propiedades del modelo creado, `feature_importances_`, que determina el peso de cada variable, normalizado a suma 1.
2. Por otro lado, se puede transformar el conjunto de datos de entrada de acuerdo a la información anterior, para de este modo simplificar y mejorar un posterior proceso de aprendizaje. En concreto, el método `transform()` reducirá el dataset dejando sólo las variables seleccionadas.

A continuación se muestra un ejemplo de cómo llevar a cabo este proceso:

```
#Para realizar selección de características, usamos lo siguiente:
from sklearn.feature_selection import SelectFromModel

#En primer lugar, se capturan los valores del ranking de importancia
importancia = rf.feature_importances_
#Se representan las 5 más importantes
(pd.Series(importancia,
        index=X_train.columns).nlargest(5).plot(kind='barh'))
plt.show()

#A continuación, se transforma el conjunto de entrenamiento y test
#Para ello, se utilizarán únicamente las variables más importantes
        según RF
fs = SelectFromModel(rf,prefit=True)  #instrucción que define las
        variables
X_train_fs = fs.transform(X_train)    #Transformación del conjunto de
        train
X_test_fs = fs.transform(X_test)      #Transformación del conjunto de
        test
```

```
rf_fs = RandomForestClassifier(random_state=42)      #Reentrenamos
        modelo específico de menos variables
rf_fs.fit(X_train_fs, y_train)


y_pred = rf_fs.predict(X_test_fs)
acc = accuracy_score(y_test,y_pred)
print()
print("El acierto con el modelo simplificado a %d variables es %.4f"%
        (X_test_fs.shape[1],acc*100))
```

```
El acierto con el modelo simplificado a 16 variables es 88.0952
```

El algoritmo Random Forest presenta tres parámetros principales:

- `n_estimators`: un valor entero (`int`) que indica el número de árboles de decisión utilizados (por defecto=100).
- `max_depth`: un valor entero (`int`) que establece máxima profundidad de cada árbol. Si no hay ninguna, entonces los nodos se expanden hasta que todas las hojas sean puras o hasta que todas las hojas contengan menos de `min_samples_split` instancias (por defecto=`None`).
- `max_features`: un valor a elegir entre {"auto", "sqrt", "log2"}, o bien un valor entero o real (`int` o `float`). Se refiere al número de variables a considerar cuando se busca la mejor división (por defecto="`auto`", que equivale a `sqrt` o raiz cuadrada del número de variables).

En general, la preferencia es tener un alto número de árboles (entre 100 y 500) con una gran profundidad (dejar como `None`), dejando también por defecto el número de variables. Sin embargo, hay que considerar que incrementar los dos primeros parámetros implicará un mayor coste computacional, alargando quizá inncesariamente el tiempo de entrenamiento. Es siempre conveniente validar y ajustar los valores de los parámetros de manera independiente para cada caso de estudio.

En el siguiente ejemplo, se muestra la frontera de decisión obtenida por Random Forest con los parámetros por defecto determinados por *Scikit-Learn*. Hay que tener en cuenta que en este problema transformado a dos únicas variables, la potencia de Random Forest con respecto a la diversidad se pierde, y la potencia predictiva tenderá a reducirse.

```
#Creamos y entrenamos el clasificador con los datos 2D
clf = RandomForestClassifier(random_state=42)
clf.fit(X_2D_train, y_2D_train)
score = clf.score(X_2D_test,y_2D_test)

fig = plt.figure(figsize=(12,9))
fig = DecisionBoundaryDisplay.from_estimator(clf, X_2D,

        response_method="predict",
                                              alpha=0.5,
                                              cmap=plt.cm.coolwarm)

fig.ax_.scatter(X_2D[:, 0], X_2D[:, 1],
                c=y_int, edgecolor="k",
                cmap=plt.cm.coolwarm)

plt.show()

<Figure size 1200x900 with 0 Axes>
```

En el siguiente trozo de código, puede observarse la estructura de uno de los árboles contenidos dentro de RandomForest:

```
#Bibliotecas necesarias para una mejor visualización
from sklearn import tree
from graphviz import Source
```

```
rf_full = RandomForestClassifier()
rf_full.fit(X, y)
dt = rf_full.estimators_[0]

#se pinta el árbol:
tree_graph = tree.export_graphviz(dt, out_file=None,
                                  feature_names=X.columns,
        class_names=pd.unique(y[y.columns[0]]),
                                  filled = True)
graph = Source(tree_graph)
graph
```

4.3 Características de Random Forest

Tal como se indicó al comienzo de esta sección, el algoritmo Random Forest es quizá uno de los más utilizados en tareas de clasificación, tanto por usuarios expertos como por aquéllos que se inician en el análisis de datos. Los principales motivos se asocian a las siguientes características:

- Robusto frente al *overfitting*, en contraposición con el comportamiento de los árboles de decisión individuales.
- La configuración de sus parámetros es bastante simple e intuitiva, y su habilidad predictiva es buena incluso utilizando los valores por defecto.
- Funciona muy bien cuando el número de variables de entrada del problema es grande y para una gran cantidad de datos.
- Permite realizar una selección de variables de alta calidad.

Sin embargo, existen algunos aspectos también negativos relativos al método Random Forest a tener en cuenta:

- El aprendizaje puede ser lento dependiendo de la parametrización, es decir, número de árboles y profundidad.
- En contraposición con los árboles de decisión simples, el modelo global Random Forest no resulta directamente interpretable por el usuario. La razón es evidente, y es que el clasificador global tiene un elevado número de árboles, que habría que revisar individualmente.
- No capta las posibles correlaciones entre las variables de entrada.

REFERENCIAS BIBLIOGRÁFICAS

- Han, J., Kamber, M., Pei, J. (2011). Data Mining: Concepts and Techniques. San Francisco, CA, USA: Morgan Kaufmann Publishers. ISBN: 0123814790, 9780123814791
- Scikit-Learn: Supervised Learning https://scikit-learn.org/stable/supervised_learning.html (visitado el 25 de Junio de 2020).
- Open Machine Learning Course: Topic 5. Ensembles of algorithms and random forest. Part 1. Bagging https://mlcourse.ai/articles/topic5-part1-bagging/ (visitado el 25 de Junio de 2020).
- Open Machine Learning Course: Topic 5. Ensembles of algorithms and random forest. Part 2. Random Forest https://mlcourse.ai/articles/topic5-part2-rf/ (visitado el 25 de Junio de 2020).
- Open Machine Learning Course: Topic 5. Ensembles of algorithms and random forest. Part 3. Feature Importances https://mlcourse.ai/articles/topic5-part3-feature-importance/ (visitado el 25 de Junio de 2020).

Referencias adicionales

- Alpaydin, E. (2016). Machine Learning: The New AI. MIT Press. ISBN: 9780262529518
- Witten, I. H., Frank, E., Hall, M. A., Pal, C. J. (2017). Data mining: practical machine learning tools and techniques. Amsterdam; London: Morgan

Kaufmann. ISBN: 9780128042915 0128042915

- Towards Data Science: Support Vector Machines(SVM) — An Overview https://towardsdatascience.com/https-medium-com-pupalerushikesh-svm-f4b42800e989 (visitado el 25 de Junio de 2020).