Módulo 5.3 Métodos avanzados en clasificación.

Autor:

Por Prof. Alberto Fernández Hilario

Profesor Titular 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 introducirán los algoritmos tipo "ensemble" (múltiples clasificadores) y en concreto se utilizará Random Forest como su mayor exponente.
4. Se analizarán las ventajas e inconvenientes de estos nuevos modelos.
5. Se mostrará cómo utilizar estos modelos avanzados mediante *Scikit-Learn* para Python.
6. 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.

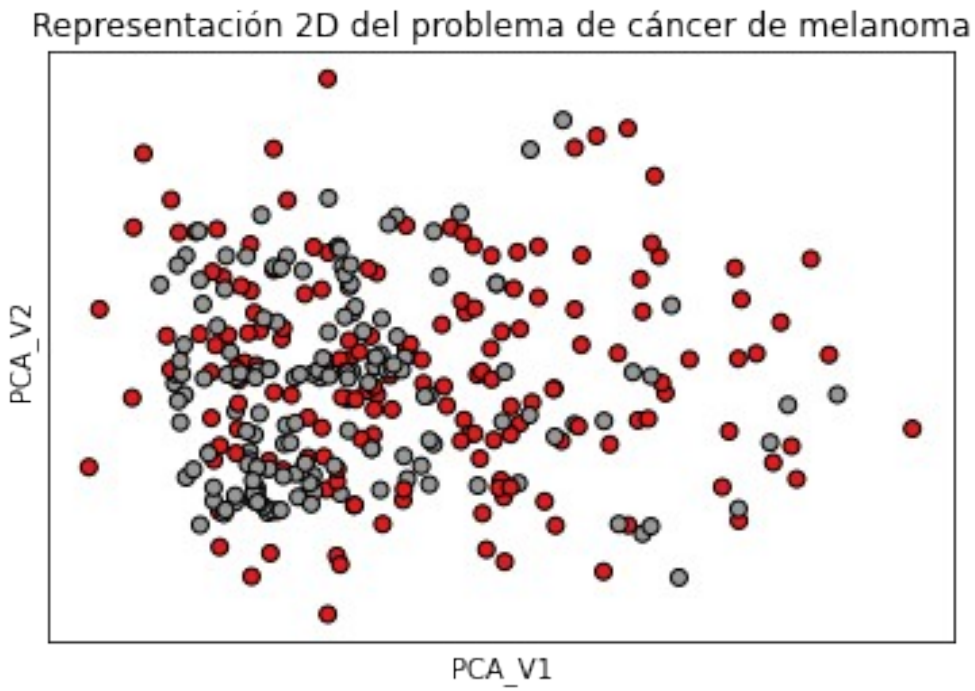
```
import pandas as pd

#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  ...     PLIN1     NCF1C   SLC7A11
0  -1.431141 -7.845756  0.665118  ...  1.285256 -0.901222  2.483020
1  -0.424374 -8.352423  0.386055  ... -1.152801  0.662490 -2.914991
```

```
2  11.014251  0.415549 -1.633781  ... -1.976877 -1.400889 -2.266779
3  -1.180446 -8.187415 -1.958023  ... -1.329148  0.055994 -2.652313
4   0.816312 -1.189303  4.837235  ...  6.284688  1.450729  1.131275

[5 rows x 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

#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)

#Pintamos en un gráfico de puntos (scatterplot) el nuevo conjunto 2D
plt.scatter(X_2D[:, 0], X_2D[:, 1], cmap=plt.cm.Set1,
c=pd.get_dummies(y).iloc[:,0], edgecolor='k')
plt.title("Representación 2D del problema de cáncer de melanoma")
plt.xlabel('PCA_V1')
plt.ylabel('PCA_V2')
plt.xticks(())
plt.yticks(())
plt.show()
```



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

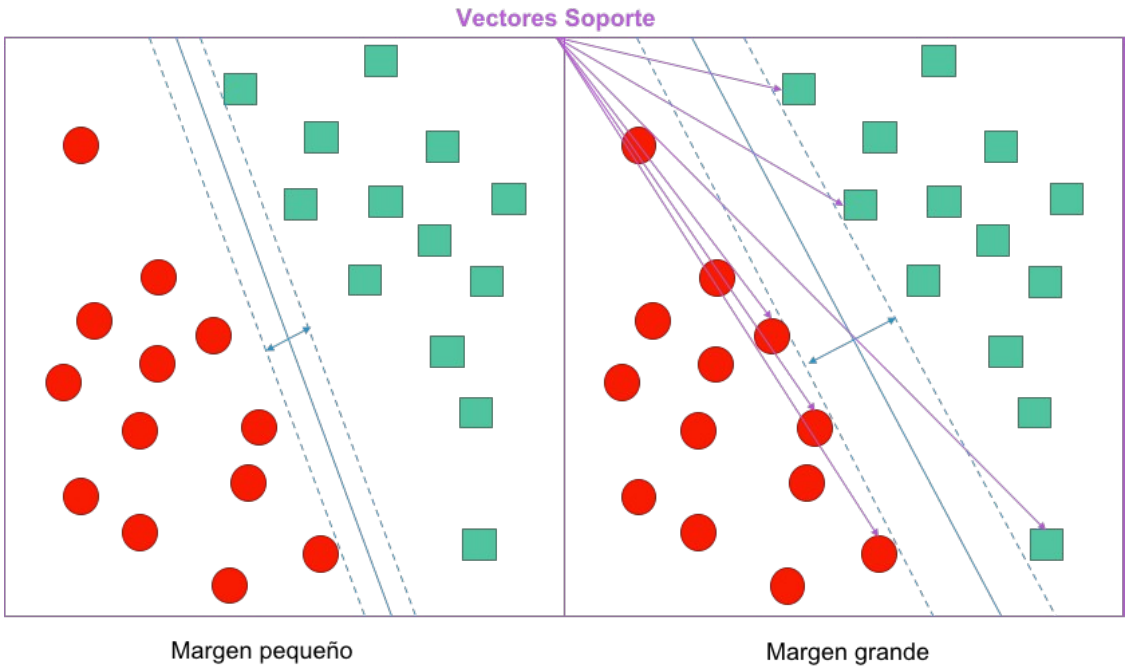
de clasificación mediante Scikit-Learn. Por último, se enumeran brevemente los pros y contras de este tipo de modelo.

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 $\| x, w \|$:

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

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:



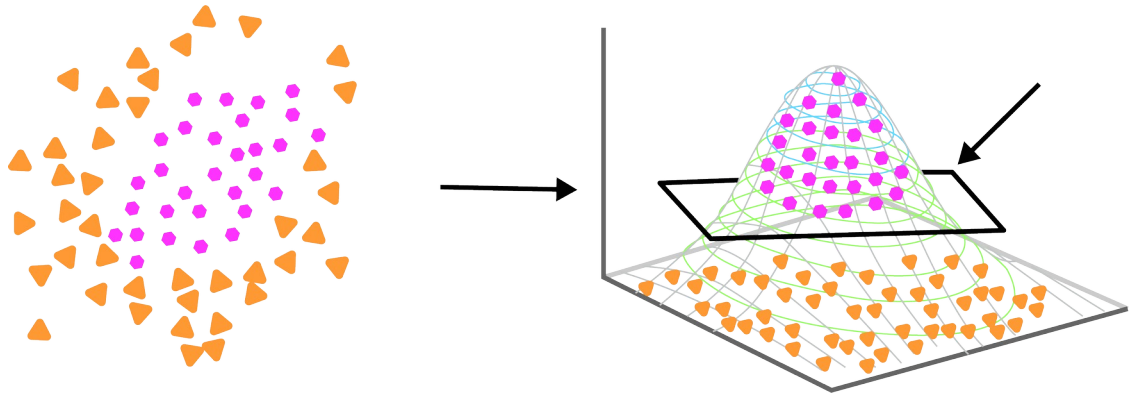
El nombre de esta técnica de aprendizaje viene determinado justamente por estas instancias en las que se "apoya" la frontera de decisión. Son los puntos más cercanos al hiperplano, e influyen directamente en su orientación para alcanzar el citado margen máximo.

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

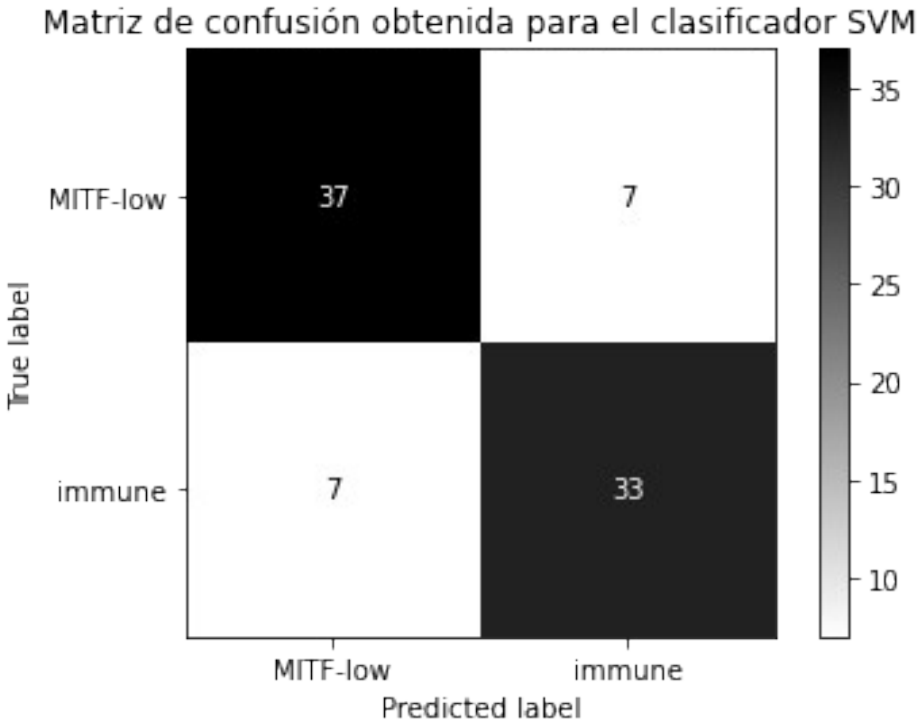
Siguiendo el formato de todos los métodos de aprendizaje en *Scikit-Learn*, para ejecutar una SVM basta con crear un objeto de la clase SVC y ajustar utilizando los conjunto X e y, tal como se muestra a continuación:

```
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)
```

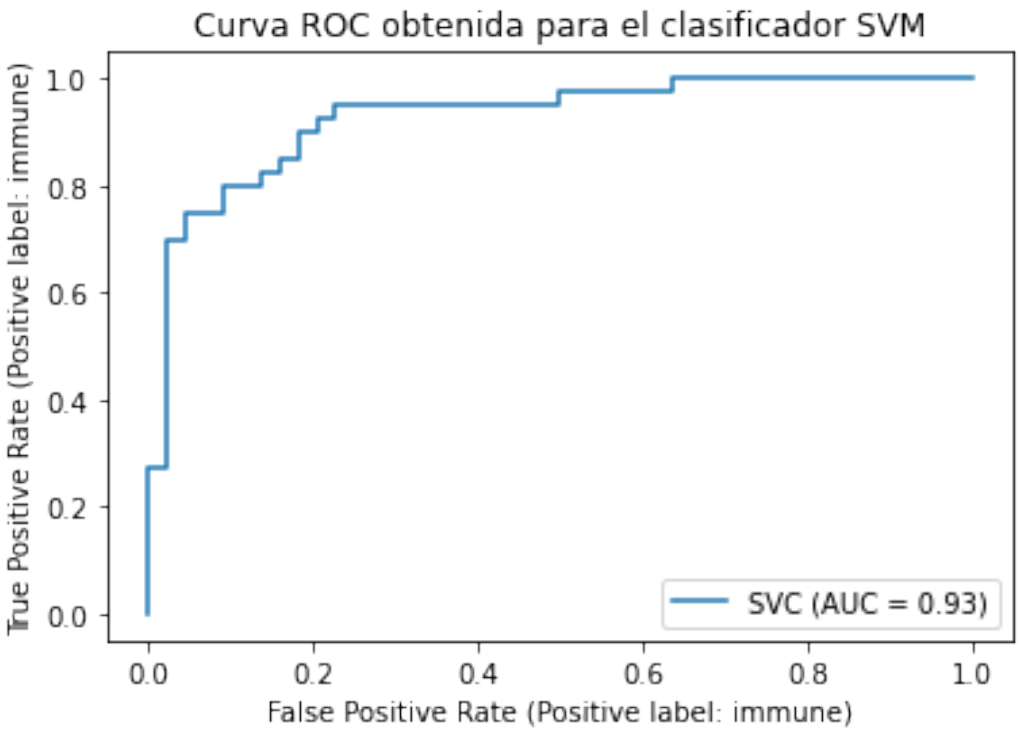
```
El porcentaje de acierto obtenido es 83.33333333333334
```

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

metrics.plot_confusion_matrix(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.plot_roc_curve(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 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:

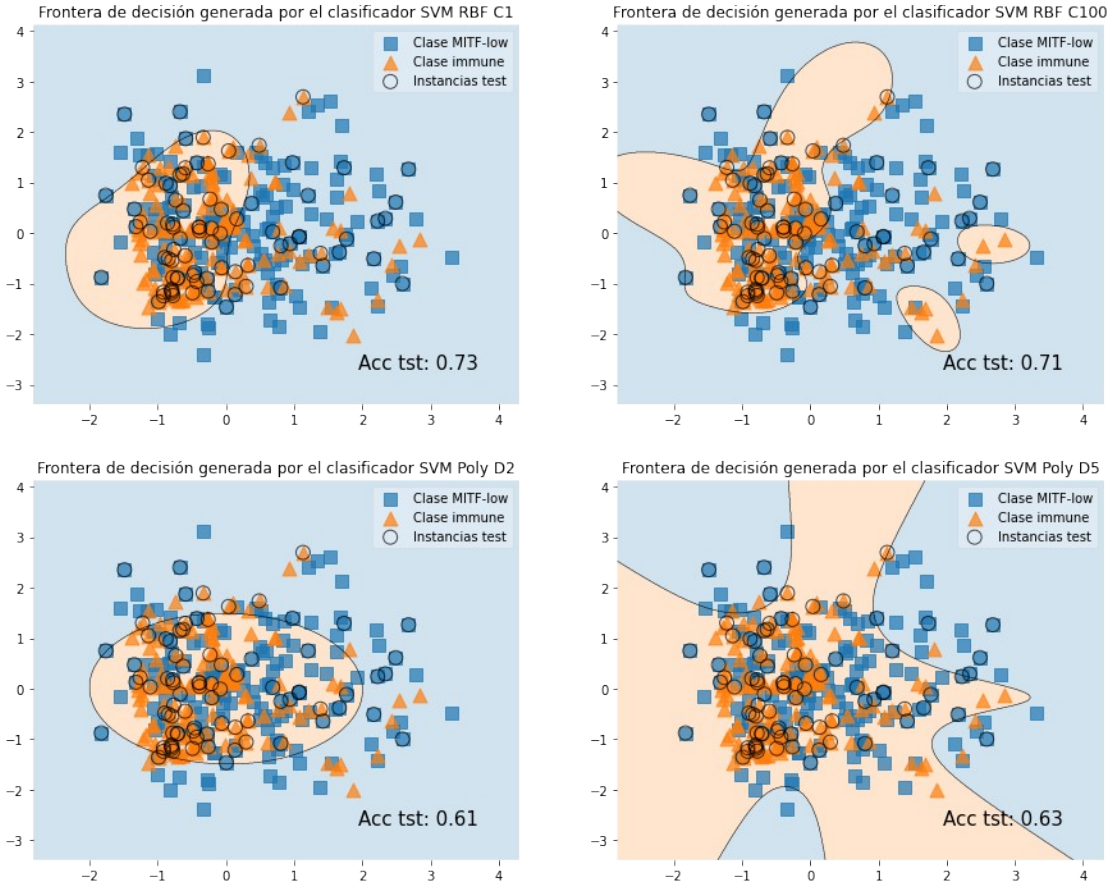
1. Las fronteras representadas cuando se utiliza el kernel `rbf` suelen ser de tipo elíptico, mientras que para el polinomial (`poly`) presentan una tendencia de tipo "lineal".
2. Al incrementar el parámetro `C` (SVM con RBF) usualmente se observan las fronteras más pegadas a los puntos de entrenamiento.

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 mlxtend.plotting import plot_decision_regions
import matplotlib.gridspec as gridspec
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)
    ax = plt.subplot(gs[grd[0], grd[1]])
    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)
    plt.title('Frontera de decisión generada por el clasificador
'+etq)
    plt.text(4 - .3, -3 + .3, ('Acc tst: %.2f' % score).lstrip('0'),
```

```
size=15, horizontalalignment='right')
    handles, labels = fig.get_legend_handles_labels()
    fig.legend(handles, ['Clase MITF-low', 'Clase immune', 'Instancias
test'],
               framealpha=0.3, scatterpoints=1)

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. 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.

3.1 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**.

3.2 Random Forest: Fundamentos e implementación en Python

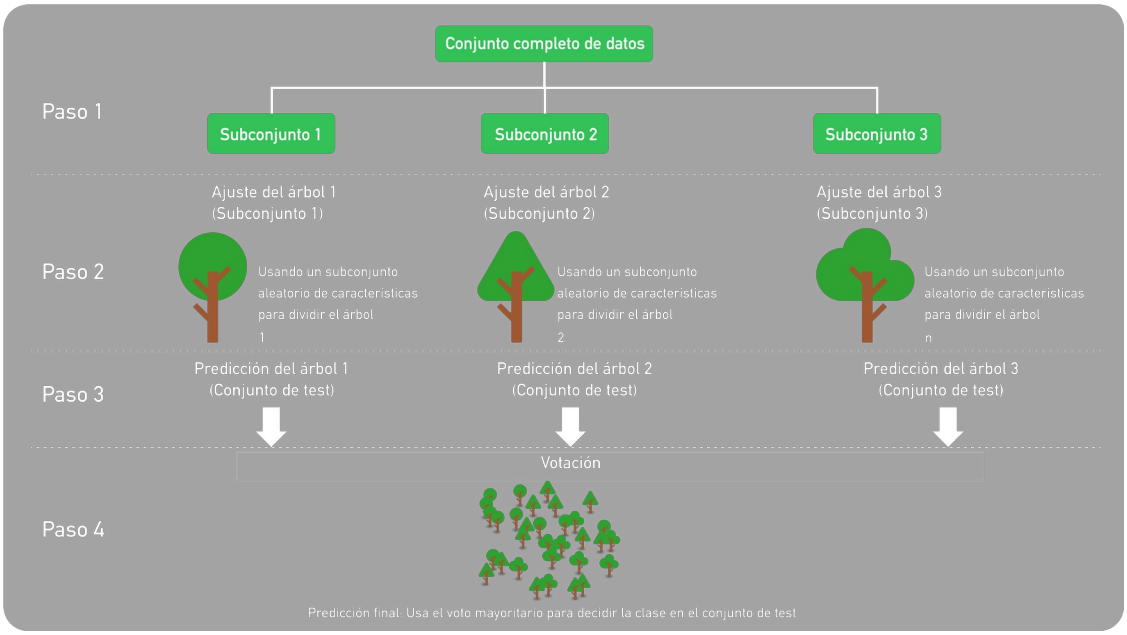
Para comprender correctamente en qué se fundamenta el algoritmo Random Forest, hay que definir con detalle la metodología de **bagging**. Si se tiene un conjunto de datos X de tamaño N, se podría generar un nuevo conjunto tomando N instancias de manera aleatoria, con **reemplazamiento**. Si este procedimiento 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.



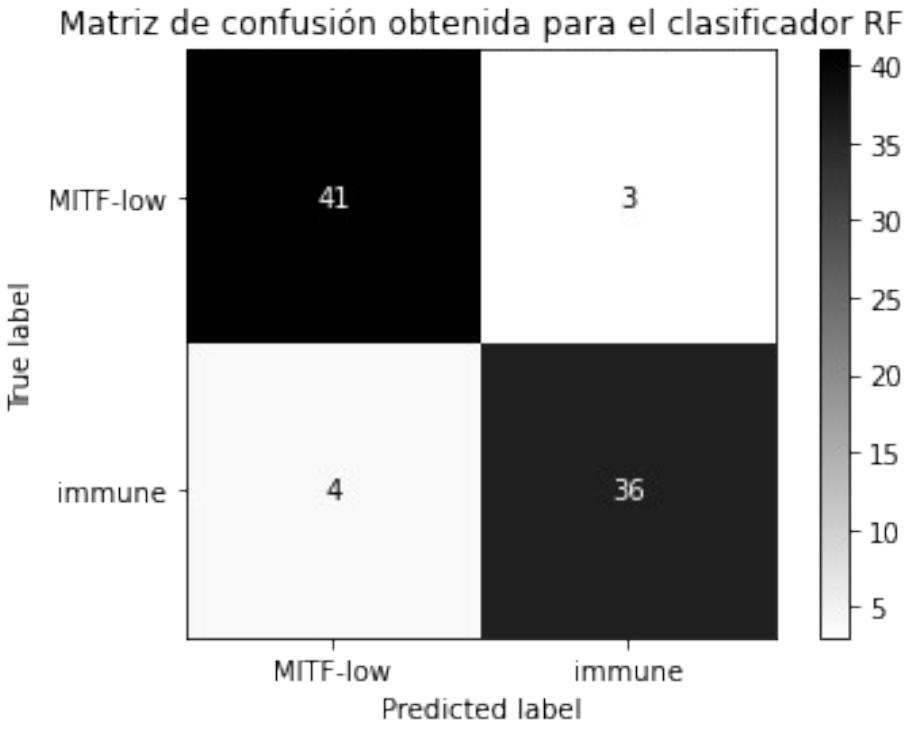
Retomando las dos características principales a cumplir para la metodología tipo Ensemble, en Random Forest se obtiene lo siguiente:

- 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.

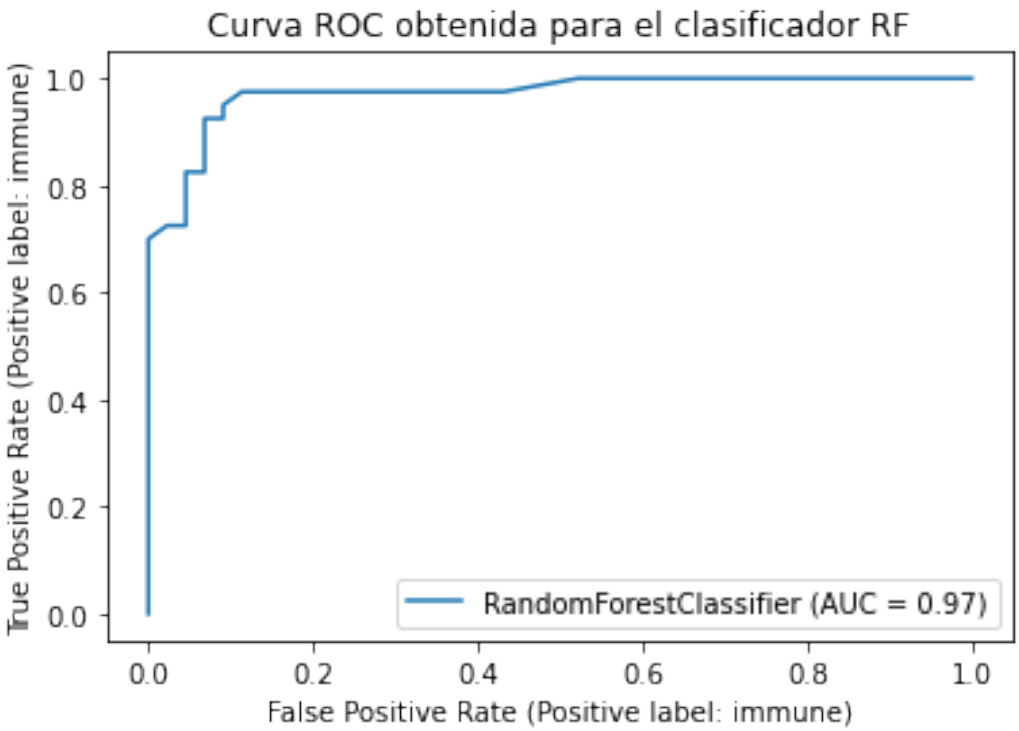
Utilizar el algoritmo Random Forest en *Scikit-Learn* es tan sencillo como cualquier otro clasificador, como se muestra el siguiente ejemplo:

```
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.plot_confusion_matrix(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.plot_roc_curve(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.

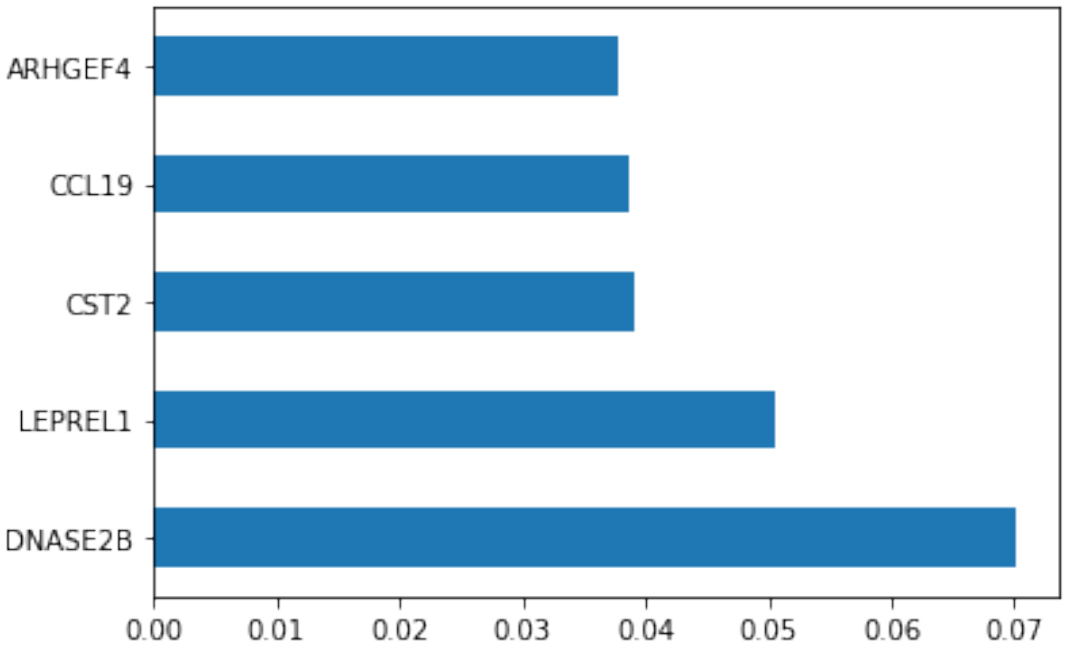
1. Por un lado, permite identificar cuáles son las variables de entrada más importantes utilizados para predecir la salida. Para ello, se debe consultar una de las propiedades del modelo creado, `feature_importances_`, que determina el peso de cada variable, normalizado a suma 1.

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:

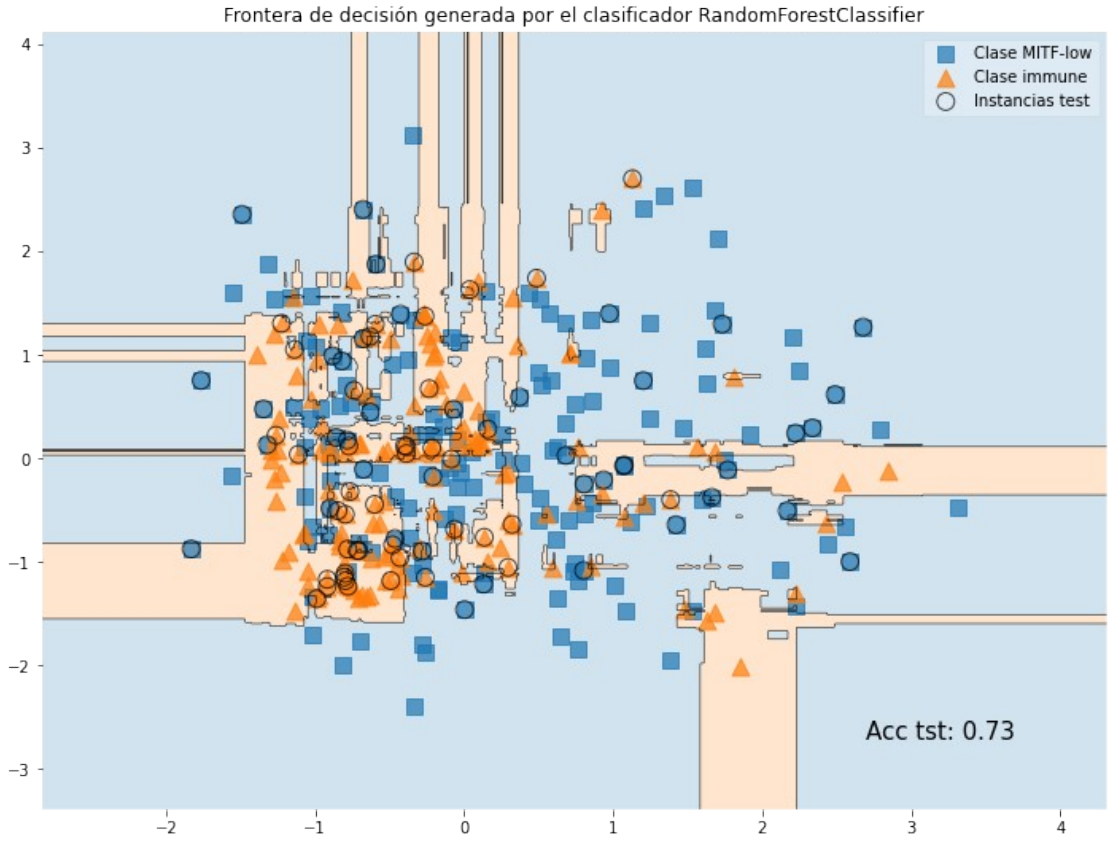
- `n_estimators`: un valor entero (`int`) que indica el número de árboles de decisión utilizados (por defecto=100).
- `max_depth`: un valor entero (`int`) que establece máxima profundidad de cada árbol. Si no hay ninguna, entonces los nodos se expanden hasta que todas las hojas sean puras o hasta que 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.

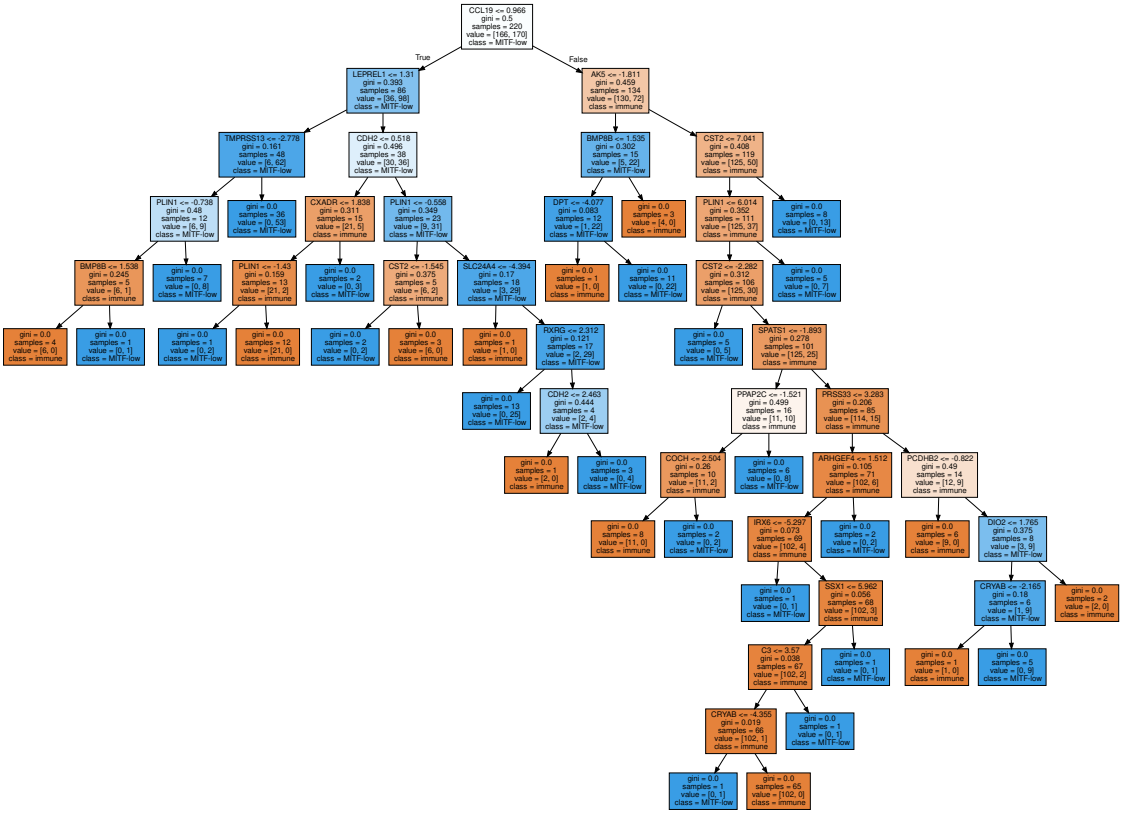
En el siguiente ejemplo, se muestra la frontera de decisión obtenida por Random Forest con los parámetros por defecto determinados por *Scikit-Learn*. Hay que tener en cuenta que en este problema transformado a dos únicas variables, la potencia de Random Forest con respecto a la diversidad se pierde, 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 = plot_decision_regions(clf=clf,X=X_2D,y=y_int.to_numpy().ravel(),
                            X_highlight=X_2D_test, legend=2,
                            scatter_kwargs=scatter_kwargs,
                            contourf_kwargs=contourf_kwargs,

scatter_highlight_kwargs=scatter_highlight_kwargs)
plt.title('Frontera de decisión generada por el clasificador
'+clf.__class__.__name__)
plt.text(4 - .3, -3 + .3, ('Acc tst: %.2f' % score).lstrip('0'),
size=15, horizontalalignment='right')
handles, labels = fig.get_legend_handles_labels()
fig.legend(handles, ['Clase MITF-low', 'Clase immune', 'Instancias
test'],
           framealpha=0.3, scatterpoints=1)

plt.show()
```



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
```



3.3 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).