Capítulo 1

Introducción a ciencia de datos

1.1. -

1.2. -

1.3. -

1.4. -

1.5. Mejores prácticas para evaluar modelos y ajuste de *hiperparámetros*

Una vez revisados diversos modelos de aprendizaje automático, es buena idea aprender las mejores prácticas para construir buenos modelos con ayuda del ajuste de los algoritmos y la evaluación de su rendimiento. Para esto, es necesario:

♢ Obtener estimaciones no sesgadas del rendimiento del modelo

♢ Diagnosticar problemas comunes de los algoritmos de aprendizaje automático

♢ *Afinar* los modelos de aprendizaje

♢ Evaluar los modelos predictivos utilizando diversas métricas para el rendimiento

1.5.1. Uso de *pipelines* para automatizar y mejorar los flujos de trabajo

Ya hemos visto que cuando aplicamos diversas transformaciones en el preprocesamiento es necesario reutilizar los parámetros obtenidos durante el ajuste/entrenamiento de cada etapa para la siguiente; *scikit-learn* provee la clase *Pipeline* (*tubería*), que muy util para automatizar el flujo de trabajo, desde los pasos de transformación hasta el ajuste del modelo para hacer predicciones sobre datos nuevos.

1.5.1.1. Conjunto de datos

Para los ejemplos de datos de esta sección, utilizaremos el *Breast Cancer Wisconsin Dataset* que se puede consultar en https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic). Contiene 569 muestras de células tumorales malignas y benignas; la primera columna contiene un identificador único, la segunda corresponde al diagnóstico (M = maligno; B = benigno). las columnas $3 - 32$ contienen 30 características de punto flotante calculadas a partir de imágenes digitalizadas de las células y que pueden ser utilizadas para predecir si un tumor es benigno o maligno.

Una vez identificadas ciertas propiedades del conjunto, podemos leerlo:

```
import pandas as pd
# https://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisco
df = pd.read_csv('https://bit.ly/3gob0mX', header=None)
```

Asignamos las 30 características al arreglo $\mathbf{X}$ y con ayuda de *LabelEncoder* transformamos las etiquetas de clase a enteros:

```
from sklearn.preprocessing import LabelEncoder
X = df.loc[:, 2:].values
y = df.loc[:, 1].values
le = LabelEncoder()
y = le.fit_transform(y)
le.classes_ # muestra las clases del objeto
```

Y podemos probar el objeto *le* usando su método *transform*:

```
le.transform(['M','B'])
```

```
array([1, 0])
```

Antes de construir nuestro *pipeline*, separaremos el conjunto de datos dejando 80 % de los datos para entrenamiento y 20 % para pruebas

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
                                                    stratify=y, random_state=1)
```

1.5.1.2. Transformaciones y estimadores en un *pipeline*

En secciones anteriores hemos visto que para que muchos algoritmos tengan buen rendimiento es recomendable que las características del conjunto de datos se encuentren en la misma escala. Además, supongamos que se desea comprimir las 30 columnas de características a un subespacio bidimensional utilizando PCA para alimentar un clasificador lineal como la regresión logística. En lugar de realizar las operaciones de ajuste y transformación una a una, se pueden *encadenar* las operaciones del *StandardScaler*, *PCA* y *LogisticRegression* dentro de un *pipeline*:

```
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import make_pipeline

pipe_lr = make_pipeline(StandardScaler(),
                        PCA(n_components=2),
                        LogisticRegression(random_state=1))
pipe_lr.fit(X_train, y_train)
y_pred = pipe_lr.predict(X_test)

print('Exactitud en test = %.3f' % pipe_lr.score(X_test, y_test))
```

La función *make_pipeline*, recibe un número arbitrario de transformadores de *scikit-learn* (objetos que implementan los métodos *fit* y *transform*), en nuestro ejemplo: *StandardScaler* y *PCA*; seguidos por un estimador de *scikit-learn* que implemente los métodos *fit* y *transform*: *LogisticRegression*. Con esta información la función *make_pipeline* crea un objeto de tipo *Pipeline*.

Un objeto tipo *Pipeline* puede verse como un *meta-estimador* que envuelve a los transformadores y el estimador. Al llamar al método *fit* del *Pipeline*, el conjunto de datos de entrenamiento viaja a través de los transformadores vía sus propias operaciones de ajuste y transformación en todos los pasos intermedios hasta llegar al objeto estimador final; es estimador será ajustado con el conjunto de datos transformado en el paso inmediato anterior.

Al llamar al *predict* del *Pipeline*, los datos que recibe viajan por todos los pasos de transformación y el estimador final obtendrá una predicción de los datos trasformados. Los *pipelines* son muy útiles cuando se busca automatizar diversas tareas en la construcción de modelos de aprendizaje automático.

1.5.2. Evaluación del rendimiento de un modelo con *k-fold cross-validation*

Un punto clave en el desarrollo de modelos de aprendizaje automático es estimar su rendimiento sobre datos que el modelo no haya visto antes. Si utilizamos un conjunto de datos para entrenamiento del modelo y el mismo conjunto para estimar su comportamiento, entonces puede obtenerse un modelo *subajustado* (alto sesgo, *bias*) si el modelo es muy simple o *sobreajustado* (alta varianza, *variance*) si es muy complejo para los datos utilizados. Para encontrar un balance aceptable entre sesgo-varianza (*bias-variance trade-off*) es necesario evaluar nuestros modelos de

forma cuidadosa; una forma de realizar esta evaluación es utilizar la *evaluación cruzada* (*cross-validation*).

La versión más simple de está técnica es la que ya hemos usado en la mayoría de los ejemplos, llamado *holdout cross-validation* (validación cruzada *dura*): partimos el conjunto de datos en datos de entrenamiento (*training*) y datos de prueba (*test*); con el primero se ajusta/entrena el modelo y con el segundo se hace una estimación de su rendimiento de *generalización*, es decir, sobre datos no usados en la etapa de ajuste.

Sin embargo, en una aplicación típica de aprendizaje autmático, también nos interesa comparar y ajustar (*tunning*) diversas valores para los parámetros del modelo para poder mejorar el rendimiento al trabajar con datos desconocidos. Este proceso se conoce como *selección de modelo* (*model selection*) que se refiere a seleccionar los valores *óptimos* de los parámetros del modelo (llamados *hiperparámetros*).

Una desventaja del método *holdout cross-validation* es que la estimación del rendimiento puede resultar muy sensible a la forma en que se obtienen los subconjuntos de entrenamiento y prueba: la estimación variará para diferentes muestras de los datos. Una técnica más robusta para estimar el comportamiento y selección de hiperparámetros es la llamada *k-fold cross-validation* (validación cruzada con *k-pliegues*) en la que se repite el método *holdout* k veces sobre k diferentes subconjuntos de entrenamiento.

1.5.2.1. *k-fold cross-validation*

En la validación cruzada *k-fold* se divide aleatoriamente el conjunto de entrenamiento en k pliegues (*folds*) sin reemplazo, donde $k-1$ pliegues se utiliza para entrenamiento y el pliegue restante se usa para evaluar el rendimiento. Este proceso se repite k veces de forma que se obtienen k modelos y estimaciones del rendimiento.

Posteriormente calculamos la media de los rendimientos de los k modelos independientes para obtener una estimación del rendimientp que es menos sensible a las subparticiones de los datos de entrenamiento que el método *holdout*. Generalmente la validación por k-pliegues se utliza para *afinar* el modelo, es decir, encontrar los valores óptimos de los hiperparámetros que entreguen un rendimiento satisfactorio en la generalización. La figura 1.1 muestra la idea detrás de este proceso.

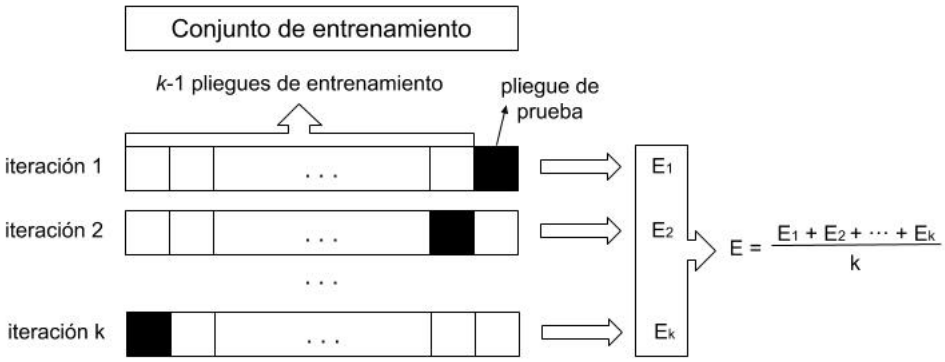


Figura 1.1: Método de validación cruzada de *k-pliegues*

Una vez obtenidos los valores óptimos para los hiperparámetros se puede reentrenar el modelo con el conjunto de entrenamiento completo para obtener una estimación final del rendimiento

usando el conjunto de prueba independiente. La razón detrás de este proceso es que ajustar el modelo con más muestras de entrenamiento usualmente resulta en un modelo más robusto y con mayor exactitud (*accuracy*).

Experimentalmente se ha observado que 10 es un buen valor para el número de pliegues; sin embargo, cuando se tienen conjuntos de entrenamiento relativamente pequeños, se recomienda incrementar el número de pliegues: si se incrementa el valor de k, se utilizará una mayor cantidad de datos de entenamiento en cada iteración lo que resulta en menor sesgo para estimar el rendimiento en la generalización al promediar las estimaciones individuales; pero hay que recordar que al incrementar el valor de k, también se incrementará el tiempo de ejecución. Por otro lado, si se tiene gran cantidad de datos, se puede elegir un valor más pequeño para k, por ejemplo 5 sin perder mucha exactitud en la estimación del rendimiento promedio del modelo mientras se reduce el costo computacional de reajustar y evaluar los diferente pliegues.

Una mejora a este algoritmo es el llamado *stratified k-fold cross-validation* que entrega mejores resultados cuando la proporción de las clases no son equitativas. En la *validación cruzada estratificada*, la proporción de clases se preserva dentro de cada pliegue para asegurar que cada uno es representativo de la proporción de las clases en el conjunto de entrenamiento. Ese último lo usaremos para el ejemplo:

```
import numpy as np
from sklearn.model_selection import StratifiedKFold

kfold = StratifiedKFold(n_splits=10).split(X_train,y_train)
scores = []
for k, (train, test) in enumerate(kfold):
  pipe_lr.fit(X_train[train], y_train[train])
  score = pipe_lr.score(X_train[test], y_train[test])
  scores.append(score)
  print('Pliegue : %2d Class dist : %s, Acc : %.4f' % (k+1,
        np.bincount(y_train[train]),score))
```

```
Pliegue  1 : Dist por clase : [256 153], Acc : 0.935
Pliegue  2 : Dist por clase : [256 153], Acc : 0.935
Pliegue  3 : Dist por clase : [256 153], Acc : 0.957
Pliegue  4 : Dist por clase : [256 153], Acc : 0.957
Pliegue  5 : Dist por clase : [256 153], Acc : 0.935
Pliegue  6 : Dist por clase : [257 153], Acc : 0.956
Pliegue  7 : Dist por clase : [257 153], Acc : 0.978
Pliegue  8 : Dist por clase : [257 153], Acc : 0.933
Pliegue  9 : Dist por clase : [257 153], Acc : 0.956
Pliegue 10 : Dist por clase : [257 153], Acc : 0.956
```

```
print('Exactitud de validación cruzada : %.3f +/- %.3f'%(np.mean(scores),
      np.std(scores)))
```

```
Exactitud de validación cruzada : 0.950 +/- 0.014
```

El ejemplo anterior es útil para ilustrar cómo funciona la validación cruzada, *scikit-learn* también implementa su evaluador, esto permite tener el mismo ejemplo de forma menos verbosa:

```
from sklearn.model_selection import cross_val_score

scores = cross_val_score(estimator=pipe_lr,
                         X=X_train, y=y_train,
                         cv=10, n_jobs=1)

print('Puntajes de exactitud de validación cruzada : %s'%(scores))

print('Exactitud de validación cruzada : %.3f +/- %.3f'%(np.mean(scores),
      np.std(scores)))
```

```
Puntajes de exactitud de validación cruzada : [0.93478261 0.93478261
   0.95652174 0.95652174 0.93478261 0.95555556 0.97777778 0.93333333
   0.95555556 0.95555556]
Exactitud de validación cruzada : 0.950 +/- 0.014
```

Una característica muy útil de la función *cross_val_score* es que la evaluación de los pliegues puede distribuirse en diferentes procesadores. Si se establece el parámetro *n_jobs* a 1, todo el trabajo se realizará en un sólo CPU; si se asigna *n_jobs*= 2 el trabajo se reparte en dos procesadores y si se establece *n_jobs*= -1, se utilizar todos los procesadores disponibles para realizar el cómputo en paralelo.

1.5.3. Depuración de algoritmos usando curvas de aprendizaje y de validación

En esta sección revisaremos dos herramientas de diagnóstico simples, pero muy poderosas que pueden ayudar a mejorar el rendimiento de un algoritmo de aprendizaje. Las curvas de aprendizaje se pueden usar para diagnosticar si el algoritmo tiene un problema de sobreajuste (alta varianza) o subajuste (alto sesgo). Además, revisaremos las curvas de validación que pueden ayudar a encontrar problemas comunes en un algoritmo de aprendizaje.

1.5.3.1. Diagnosticando problemas de sesgo y varianza con curvas de aprendizaje

Si un modelo es muy complejo para un conjunto de datos de entrenamiento, el modelo tiende a sobreajustarse a los datos de entrenamiento y no generaliza correctamente a datos desconocidos. Graficando las precisiones de entrenamiento y validacion como funciones del número de muestras,

es posible detectar si el modelo sufre de varianza alta o sesgo alto y si puede mejorar recolectando más muestras.

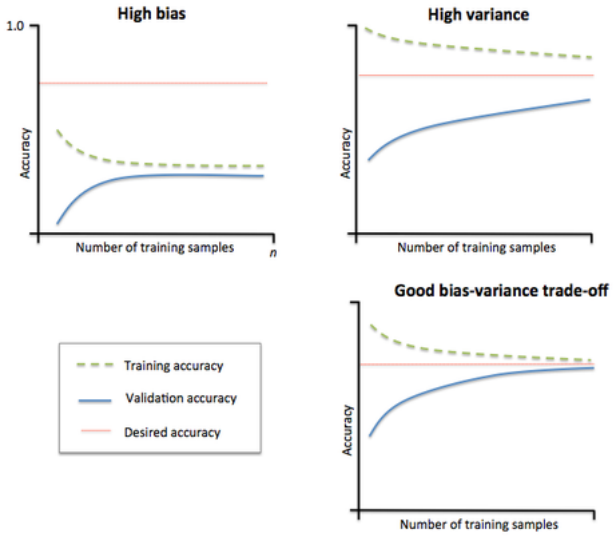


Figura 1.2: Balance sesgo-varianza (*bias-variance trade-off*)

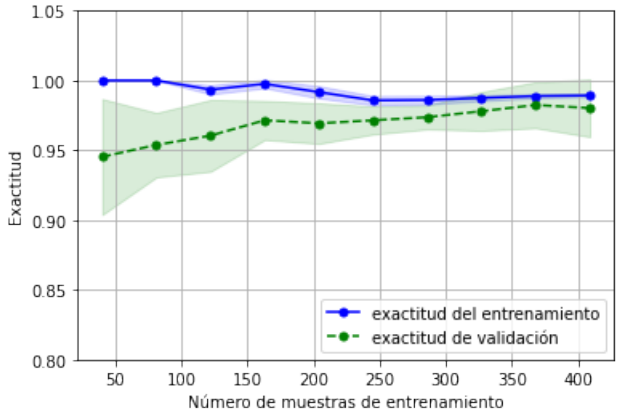
En la figura 1.2 se observan tres relaciones sesgo-varianza:

♢ La gráfica superior izquierda muestra un modelo con sesgo alto: este modelo tiene poco entrenamiento y baja exactitud, indicadores de un subajuste a los datos de entrenamiento. Formas comunes de enfrentar este problema, es incrementar el número de parámetros, por ejemplo, recolectando o creando características adicionales; o disminuyendo el grado de regularización en clasificadores como SVM o regresión logística.

♢ La gráfica superior derecha presenta un modelo con alta varianza, indicado por una diferencia grande entre la exactitud de entrenamiento y de validación cruzada. Para enfrentar el problema de sobreajuste, se pueden recolectar más datos de entenamiento, reducir la complejidad del modelo o incrementar el parámetro de regularización. Para modelos sin regularización, puede ser de ayuda disminuir el número de columnas con selección (SBS) o extracción (PCA) de características para minimizar el grado de sobreajuste. Si bien recolectar más datos de entrenamiento puede reducir la posibilidad de sobreajustes, no siempre es garantía, por ejemplo, si los datos de entrenamiento son extremadamente ruidosos o el modelo está muy cerca del óptimo.

Veamos el uso de la función *learning_curve* de *scikit-learn* para evaluar el modelo:

```
import matplotlib.pyplot as plt
from sklearn.model_selection import learning_curve

pipe_lr = make_pipeline(StandardScaler(),
                        LogisticRegression(penalty='l2', random_state=1))

train_sizes,train_scores,test_scores= learning_curve(estimator=pipe_lr,
                                                     X=X_train, y=y_train,
                                                     train_sizes=np.linspace(0.1,1,10),
                                                     cv=10, n_jobs=1)
train_mean = np.mean(train_scores, axis=1)
train_std = np.std(train_scores, axis=1)
test_mean = np.mean(test_scores, axis=1)
test_std = np.std(test_scores, axis=1)

plt.plot(train_sizes, train_mean, color='blue', marker='o', markersize=5,
         label='exactitud del entrenamiento')
plt.fill_between(train_sizes, train_mean+train_std, train_mean-train_std,
                 alpha=0.15, color='blue')

plt.plot(train_sizes, test_mean, color='green', marker='o', markersize=5,
         linestyle='--', label='exactitud de validación')
plt.fill_between(train_sizes, test_mean+test_std, test_mean-test_std,
                 alpha=0.15, color='green')

plt.grid()
plt.xlabel('Número de muestras de entrenamiento')
plt.ylabel('Exactitud')
plt.legend(loc='lower right')
plt.ylim([0.8, 1.05])
plt.show()
```



Con el parámetro *train_ sizes*, se puede controlar el número (absoluto o relativo) de muestras de entenamiento usadas para generar las curvas de aprendizaje. En nuestro ejemplo, usamos 10 intervalos relativos igualmente espaciados para los tamaños de los conjuntos de enternamiento. Por omisión, se utiliza validación cruzada de $k-$pliegues estratificada, establecemos $k = 10$ vía el parámetro *cv*. Después obtenemos la exactitud promedio tanto de los entrenamientos como de las pruebas para los diferentes tamaños de las muestras y se muestran en una gráfica con la función *plot* y agregamos la desviación estándar de la exactitud con ayuda de la función *fill_ between*.

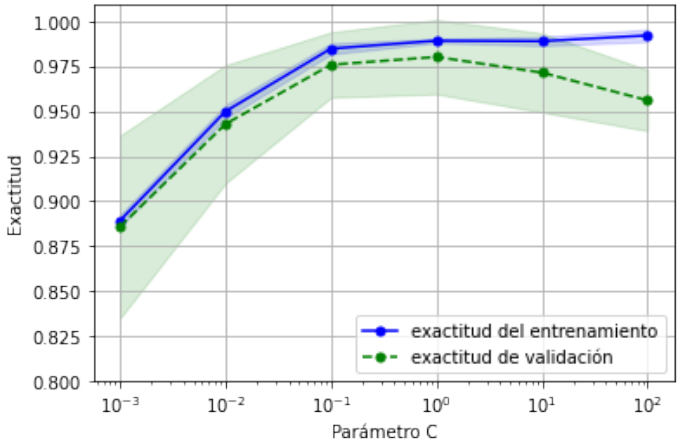
La curva de aprendizaje de la figura muestra que nuestro modelo tiene buen rendimiento tanto en los conjuntos de entrenamiento como en los de validación si se utilizan más de 250 muestras para enternamiento. También podemos ver que la exactitud de entrenamiento es mayor para conjuntos menores a 250 muestras y que la diferencia entre la validación y el entrenamiento se incrementa (un indicador de sobreajuste).

1.5.3.2. Enfrentando sub y sobreajuste con curvas de validación

Las curvas de validación son similares a las de aprendizaje, pero en lugar de graficar las precisiones de entrenamiento y validacion como funciones del número de muestras, se varían los valores de los parámetros del modelo; por ejemplo, el parámetro de regularización inversa C en la regresión logística. Veamos cómo se crea una curva de validación con *scikit-learn*:

```
from sklearn.model_selection import validation_curve
param_range = [0.001, 0.01, 0.1, 1.0, 10.0, 100.0]

train_scores, test_scores = validation_curve(estimator=pipe_lr,
                                             X=X_train, y=y_train,
                                             param_name='logisticregression__C'
                                             param_range=param_range,
                                             cv=10)
```

```
train_mean = np.mean(train_scores, axis=1)
train_std = np.std(train_scores, axis=1)
test_mean = np.mean(test_scores, axis=1)
test_std = np.std(test_scores, axis=1)

plt.plot(param_range, train_mean, color='blue', marker='o', markersize=5,
         label='exactitud del entrenamiento')
plt.fill_between(param_range, train_mean+train_std, train_mean-train_std,
                 alpha=0.15, color='blue')

plt.plot(param_range, test_mean, color='green', marker='o', markersize=5,
         linestyle='--', label='exactitud de validación')
plt.fill_between(param_range, test_mean+test_std, test_mean-test_std,
                 alpha=0.15, color='green')

plt.grid()
plt.xscale('log')
plt.xlabel('Parámetro C')
plt.ylabel('Exactitud')
plt.legend(loc='lower right')
plt.ylim([0.8, 1.01])
plt.show()
```



Al igual que las curvas de aprendizaje, las de validación utilizan validación cruzada de $k-$pliegues estratificada para estimar el rendimiento del clasificador. Dentro de la función *validation_curve*, se especifica el parámetro de regularización a evaluar: *logisticregression__C* y los valores que tomará mediante *param_range*.

Aún cuando las variaciones con respecto al parámetro C tienen una variación muy sutil, podemos ver que el modelo presenta subajuste cuando incrementamos la fuerza de la regularización (valores pequeños de C). Sin embargo, para valores grandes de C, es decir, al reducir la fuerza del

parámetro de regularización, el modelo tiene a sobreajustarse a los datos de entrenamiento. Para nuestro ejemplo el *punto justo* del parámetro C parece encontrarse cerca de 0.1.

1.5.4. Ajuste *fino* de modelos de aprendizaje automático

Existen dos tipos de parámetros en los modelos de aprendizaje automático: (1) aquellos que se aprenden de los datos de entrenamiento, por ejemplo, los pesos de la regresión logística y (2) los parámetros del algoritmo que se optimizan por separado. Los últimos son parámetros de ajuste, tambien llamados *hiperparámetros* del modelo, por ejemplo, el parámetro de regularización de la regresion logística.

En la sección anterior usamos las curvas de validación para mejorar el rendimiento de un algoritmo optimizando uno de sus hiperparámetros. Ahora revisaremos una técnica de optimización de hiperparámetros llamada *búsqueda de malla* (*grid seach*) que puede ayudar a mejorar aún más el rendimiento de un modelo al buscar la combinación *óptima* para los valores de los hiperparámetros.

1.5.4.1. Ajustando hiperparámetros con *grid search*

El enfoque de la búsqueda de malla es sencillo: es un paradigma de búsqueda de fuerza bruta exhaustiva en el que especificamos una lista de valores para diferentes hiperparámetros y la computadora evalua el rendimiento del modelo para cada combinación para obtener la combinación óptima de los valores:

```
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC

pipe_svc = make_pipeline(StandardScaler(),
                         SVC(random_state=1))

param_range = [0.0001,0.001,0.01,0.1,1,10,100,1000]
param_grid = [{'svc__C':param_range,'svc__kernel':['linear']},
              {'svc__C':param_range,'svc__gamma':param_range,
               'svc__kernel':['rbf']}]

gs = GridSearchCV(estimator=pipe_svc, param_grid=param_grid,
                  scoring='accuracy', cv=10, n_jobs=-1)
gs = gs.fit(X_train, y_train)
```

En el código anterior inicializamos un objeto *GridSearchCV* para entrenar y ajustar un *pipeline* para una *máquina de soporte vectorial* (*Support Vector Machine SVM*). Establecemos el parámetro *param_grid* para la búsqueda de malla como una lista de diccionarios para especificar los parámetros que queremos ajustar: para la SVM lineal, solo evaluamos el parámetro de regularización inverso C; para la SVM con *kernel* RBF ajustamos tanto *svc__C* como *svc__gamma*.

Después de realizar la búsqueda de malla con los datos de entrenamiento, podemos obtener resultado del modelo con el mejor rendimiento con el atributo *best_score_* y los hiperparámetros

asociados se encuentra en el atributo *best_params_*.

```
print(gs.best_score_)
print(gs.best_params_)
```

```
0.9846859903381642
{'svc__C': 100, 'svc__gamma': 0.001, 'svc__kernel': 'rbf'}
```

Para nuestro ejemplo, la SVM con *kernel* RBF con $svc\_\_C = 100$ y $svc\_\_gamma = 0.001$ entregan la mejor exactitud de validación cruzada: 98.47 %.

Finalmente, utilizamos el conjunto de prueba independiente para estimar el rendimiento de generalización del mejor modelo seleccionado, accesible vía el atributo *best_estimator_*:

```
clf = gs.best_estimator_
clf.fit(X_train, y_train)
print('Exactitud en test : %.3f' % clf.score(X_test, y_test))
```

```
Exactitud en test : 0.974
```

La búsqueda de malla es un enfoque poderoso para encontrar el conjunto óptimo de parámetros; sin embargo, la evaluación de todas las combinaciones posibles es computacionalmente cara. Un enfoque alternativo es muestrear diferentes combinaciones usando la búsqueda aleatoria dentro de *scikit-learn*:

https://scikit-learn.org/stable/modules/grid_search.html#randomized-parameter-optimization

1.5.5. Selección de algoritmos con validación cruzada anidada

Combinando validación cruzada con búsqueda de malla es un enfoque útil para el ajuste fino del rendimiento de un modelo de aprendizaje automático al probar diversos valores para sus hiperparámetros. Si lo que se desea es elegir entre diversos algoritmos de aprendizaje automático una técnica recomendada es la *validación cruzada anidada* (*nested cross-validation*).

En la validación cruzada anidada tenemos un ciclo externo de validación cruzada de $k-$pliegues que divide el conjunto original en pliegues de entrenamiento y prueba como la revisada anteriormente y un ciclo interno usado para seleccionar el modelo usando validación cruzada de $k-$pliegues sobre el conjunto de entrenamiento.

En la figura 1.3 presenta el concepto con cinco pliegues externos y dos internos para validación cruzada; estos valores son útiles cuando se cuenta con grandes conjuntos de datos y el rendimiento computacional es importante. Esta configuración particular de validación cruzada anidada se conoce como 5×2 *cross-validation*.

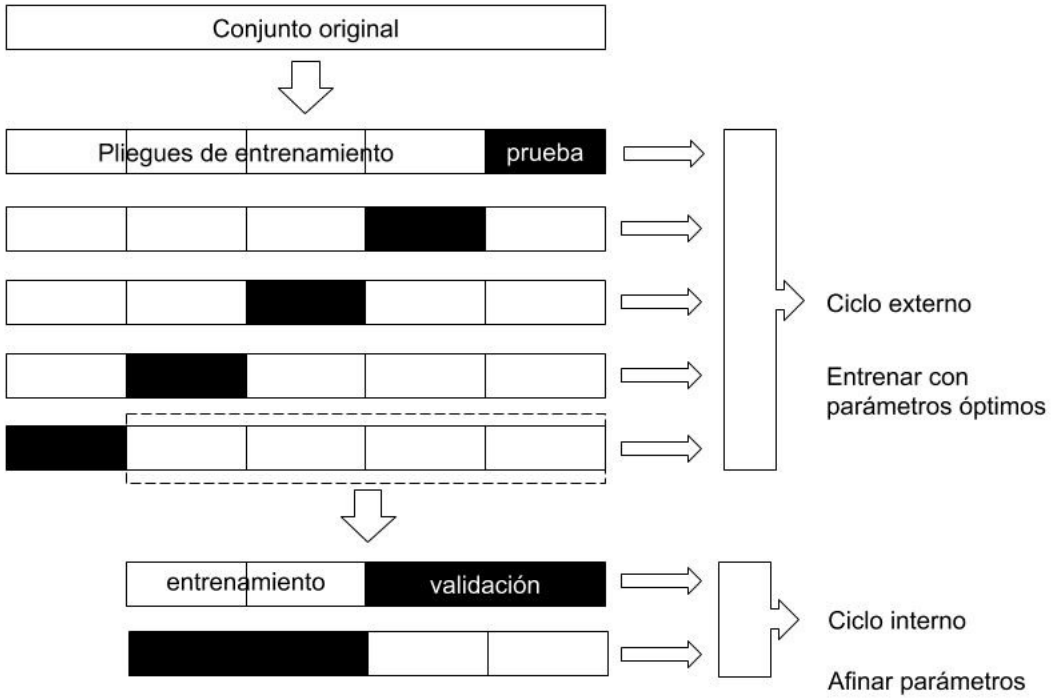


Figura 1.3: 5×2 *nested cross-validation*

En scikit-learn la validacióncruzada anidada puede realizarse de la siguiente manera:

```
gs = GridSearchCV(estimator=pipe_svc, param_grid=param_grid,
                  scoring='accuracy', cv=2)

scores = cross_val_score(gs, X_train, y_train,
                         scoring='accuracy', cv=5)

print('Exactitud de NCV : %.3f +/- %.3f'%(np.mean(scores),np.std(scores)))
```

```
Exactitud de NCV : 0.974 +/- 0.015
```

La exactitud de validación cruzada promedio nos da una buena estimación sobre lo que se esperará si afinamos los hiperparámetros de un modelo y se aplican sobre datos desconocidos. Por ejemplo, podemos usar esta estrategia para comparar el rendimiento de un modelo de SVM con un clasificador de árbol de decisión simple; por simplicidad sólo se afinará el parámetro de profundidad:

```
from sklearn.tree import DecisionTreeClassifier

gs = GridSearchCV(estimator=DecisionTreeClassifier(random_state=0),
                  param_grid=[{'max_depth':[1,2,3,4,5,6,7,None]},
                  scoring='accuracy', cv=2)

scores = cross_val_score(gs, X_train, y_train,
                         scoring='accuracy', cv=5)

print('Exactitud de NCV : %.3f +/- %.3f'%(np.mean(scores),np.std(scores)))
```

```
Exactitud de NCV : 0.934 +/- 0.016
```

Podemos ver que el rendimiento calculado por validación cruzada anidada del modelo SVM (97.4 %) es notablemente mejor que el del árbol de decisión (93.4 %); es de espararse que sea una mejor opción para clasificar datos nuevos que vengan de la misma población del conjunto de datos de entrenamiento.

1.5.6. Revisión de varias métricas de evaluación de rendimiento

En las secciones anteriores, hemos evaluado el rendimiento de los modelos con ayuda de su *exactitud* (*accuracy*) que es una métrica útil con la que se puede cuantificar el rendimiento general de un modelo. Sin embargo, existen otras métricas del rendimiento que pueden usarse para determinar la relevancia de un modelo, tales como *precision* (precisión), sensitividad (*recall*) y *F1-score*.

1.5.6.1. Lectura de una matriz de confusión

Antes de entrar a los detalles de las métricas, revisemos la *matriz de confusión*, es una matriz que despliega el rendimiento de un algoritmo de aprendizaje: es una matriz cuadrada donde se reportan los totales de las predicciones *verdaderas positivas* (TP), *verdaderas negativas* (TN), *falsas positivas* (FP) y *falsas negativas* (FN) de un clasificador, como se muestra en la figura 1.4.

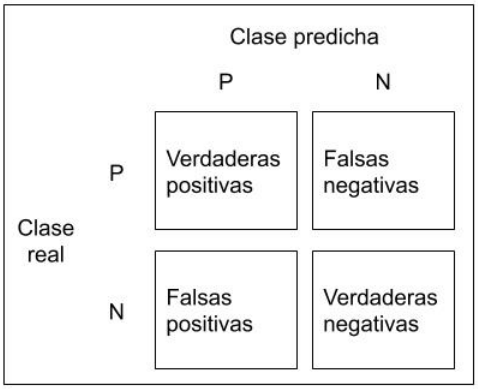


Figura 1.4: Posibles casos para las predicciones

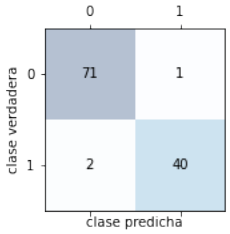
Dentro de *scikit-learn* existe la función *confusion_ matrix*:

```
from sklearn.metrics import confusion_matrix

pipe_svc.fit(X_train, y_train)
y_pred = pipe_svc.predict(X_test)
confmat = confusion_matrix(y_true=y_test, y_pred=y_pred)
print(confmat)
```

```
[[71  1]
 [ 2 40]]
```

Y también es posible producir una imagen con esta información con ayuda de *matshow* incluída dentro de *matplotlib*:

```
fig, ax = plt.subplots(figsize=(2.5, 2.5))
ax.matshow(confmat, cmap=plt.cm.Blues, alpha=0.3)
for i in range(confmat.shape[0]):
  for j in range(confmat.shape[1]):
    ax.text(x=j, y=i, s=confmat[i,j])
```



Puede resultar más fácil interpretar los valores obtenidos y con ellos calcular varias métricas de error.

1.5.6.2. Métricas de *precision* y *recall* de un modelo de clasificación

Tanto el *error* (*ERR*) como la exactitud (*accuracy ACC*) de la predicción proveen información general sobre cuantas muestras fueron clasificadas erróneamente. El error se define como la suma de predicciones falsas entre el número total de predicciones:

$$ERR = \frac{FP + FN}{FP + FN + TP + TN}$$

La exactitud (*accuracy*) de la predicción puede calcularse directamente a partir del error:

$$ACC = \frac{TP + TN}{FP + FN + TP + TN} = 1 - ERR$$

La *proporción de verdaderas positivas* (TPR) y la *proporción de falsas positivas* (FPR) son métricas de rendimiento especialmente útiles en problemas de clases desbalanceadas:

$$FPR = \frac{FP}{N} = \frac{FP}{FP + TN}; \quad TPR = \frac{TP}{P} = \frac{TP}{FN + TP}$$

Por ejemplo, al diagnosticar tumores nos interesa más detectar tumores malignos para poder ayudar con el tratamiento apropiado para el paciente. Sin embargo, también es importante disminuir el número de tumores benignos que fueron erróneamente clasificados como malignos (FP) para evitar preocupar innecesariamente a algún paciente. En contraste a la FPR, la TPR provee información útil sobre la fracción de muestras positivas que fueron identificadas correctamente de entre el total de positivas (P).

Las métricas precisión (*precision PRE*) y sensitividad (*recall REC*) se relacionan con las proporciones positivas y negativas; de hecho, REC es un sinónimo de TPR:

$$PRE = \frac{TP}{TP + FP}; \quad REC = TPR = \frac{TP}{P} = \frac{TP}{FN + TP}$$

En la práctica, la combinación de PRE y REC para obtener la métrica llamada $F1 - score$:

$$F1 = 2\frac{PRE \times REC}{PRE + REC}$$

Estas métricas se encuentran implementadas en *scikit-learn* dentro del módulo *sklearn.metrics*:

```
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score, f1_score

print('Precisión : %.3f' % precision_score(y_true=y_test, y_pred=y_pred))
print('   Recall : %.3f' % recall_score(y_true=y_test, y_pred=y_pred))
print('       F1 : %.3f' % f1_score(y_true=y_test, y_pred=y_pred))
```

```
Precisión : 0.97
   Recall : 0.952
       F1 : 0.964
```

Incluso la clase *GridSearchCV* incluye otras métricas además de la exactitud (*accuracy*) vía su parámetro *scoring*. La lista completa puede consultarse en:

https://scikit-learn.org/stable/modules/model_evaluation.html

Es importante recordar que la *clase positiva* para *scikit-learn* es la que tiene la etiqueta 1; si deseamos especificar una *etiqueta positiva* diferente, podemos contruir un marcador (*scorer*) propio con ayuda de la función *make_scorer*, que puede pasarse directamente como parámetro de *GridSearchCV* (en nuestro ejemplo, *f1_score*):

```
from sklearn.metrics import make_scorer, f1_score
scorer = make_scorer(f1_score, pos_label=0)
gs = GridSearchCV(estimator=pipe_svc,param_grid=param_grid,
                  scoring=scorer, cv=10)
gs = gs.fit(X_train, y_train)
print(gs.best_score_)
print(gs.best_params_)
```

```
0.9880771478667446
{'svc__C': 100, 'svc__gamma': 0.001, 'svc__kernel': 'rbf'}
```

1.5.6.3. Curva ROC (*Receiver Operating Characteristic*, Característica Operativa del Receptor)

Este tipo de gráficas son muy útiles para elegir modelos de clasificación basados en su rendimiento con respecto a las métricas FPR y TPR calculadas moviendo el umbral del clasificador. La diagonal de una gráfica ROC puede interpretarse como *una estimación aleatoria*, de forma que los clasificadores que caigan por debajo de esta diagonal se consideran peor que un modelo aleatorio. Un clasificador *perfecto* alcanza la esquina superior izquierda con $TPR = 1$ y $FPR = 0$. Basados en la curva ROC, es posible calcular la llamada *área bajo la curva ROC* (*ROC Area Under the Curve ROC*, *AUC*) para caracterizar el rendimiento del modelo.

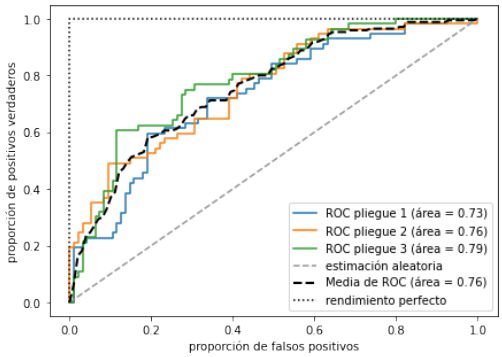
También es posible obtener *precision-recall curves* para diferentes umbrales de un clasificador; se puede consultar en:

https://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_recall_curve.html

Con el código del ejemplo siguiente obtenemos la curva ROC de un clasificador que utiliza solamente dos características del conjunto de datos *Breast Cancer Wisconsin Dataset* para predecir si un tumor es benigno o maligno. Usaremos el mismo *pipeline* de regresión logística definido previamente pero la tarea de clasificación se hace más retadora para obtener una curva que sea visualmente más interesante; por la misma razón reducimos el número de pliegues en el *StratifiedKFold* a tres:

```
from sklearn.metrics import roc_curve, auc
from numpy import interp

pipe_lr = make_pipeline(StandardScaler(), PCA(n_components=2),
                        LogisticRegression(penalty='l2',random_state=1,
                                           C=100.0))

X_train2 = X_train[:, [4, 14]]
cv = list(StratifiedKFold(n_splits=3).split(X_train, y_train))
```

```
fig = plt.figure(figsize=(7, 5))

mean_tpr = 0.0
mean_fpr = np.linspace(0, 1, 100)
all_tpr = []

for i, (train, test) in enumerate(cv):
  # probabilities
  probas = pipe_lr.fit(X_train2[train],
                       y_train[train]).predict_proba(X_train2[test])
  fpr, tpr, threshold = roc_curve(y_train[test], probas[:,1], pos_label=1)
  mean_tpr += interp(mean_fpr, fpr, tpr)
  mean_tpr[0] = 0.0
  roc_auc = auc(fpr, tpr)
  plt.plot(fpr, tpr, label='ROC pliegue %d (área = %0.2f)' % (i+1, roc_auc))

plt.plot([0,1], [0,1], linestyle='--', color=[0.6,0.6,0.6],
         label='estimación aleatoria')

mean_tpr /= len(cv)
mean_tpr[-1] = 1.0
mean_auc = auc(mean_fpr, mean_tpr)
plt.plot(mean_fpr, mean_tpr, 'k--',
         label='Media de ROC (área = %.2f)'%mean_auc, lw=2)
plt.plot([0,0,1], [0,1,1], linestyle=':', color='black',
         label='rendimiento perfecto')

plt.xlim([-0.05, 1.05])
plt.ylim([-0.05, 1.05])
plt.xlabel('proporción de falsos positivos')
plt.ylabel('proporción de positivos verdaderos')
plt.legend(loc='lower right')
plt.show()
```



En el código utilizamos la clase *StratifiedKFold* y calculamos la curva ROC del clasificador de regresión logística en nuestro *pipe_lr* con ayuda de la función *roc_curve* incluida en *sklearn.metrics* para cada iteración. Además, interpolamos la curva ROC media de los tres pliegues con la función *interp* de *numpy*, finalmente obtenemos el área bajo la curva con la función *auc*. La curva ROC resultante indica cierto grado de varianza entre los pliegues y que el promedio de ROC AUC (0.76) cae entre el rendimiento perfecto y la estimación aleatoria.

La documentación de incluye más ejemplos de esta curva:
https://scikit-learn.org/stable/modules/generated/sklearn.metrics.roc_curve.html
https://scikit-learn.org/stable/auto_examples/model_selection/plot_roc.html

1.5.6.4. Métricas para problemas de multiclase

Las métricas presentadas hasta este momento son específicas para clasificacion binaria; sin embargo, *scikit-learn* tiene implementadas métodos de promedio *macro* y *micro* que extienden esas medidas para problemas multiclase con ayuda de la clasificación *One-versus-All* (OvA). La *micro-media* se calcula a partir de los valores individuales; por ejemplo, la *micro-media* para la precisión de un sistema de $k-$clases se calcula como:

$$PRE_{micro} = \frac{TP_1 + \cdots + TP_k}{TP_1 + \cdots + TP_k + FP_1 + \cdots + FP_k}$$

La *macro-media* es simplemente la media de las medidas de los diferentes sistemas:

$$PRE_{macro} = \frac{PRE_1 + \cdots + PRE_k}{k}$$

La *micro-media* es útil si deseamos ponderar equitativamente cada predicción, mientras que la *macro-media* sirve para evaluar el rendimiento global de un clasificador con respecto a las etiquetas de clase más frecuentes.

La *macro-media* es la opción por omisión en *scikit-learn* para problemas multiclase, pero podemos especificar el método con el parámetro *average* dentro de la función de medida; por ejemplo, en la función *make-scorer*:

```
my_scorer = make_scorer(score_func=precision_score, pos_label=1,
                        greater_is_better=True, average='micro')
```

1.5.7. Tratando con datos desbalanceados

Un problema común cuando se trabaja con problemas *reales* es que los conjuntos de datos estén desbalanceados respecto a las etiquetas de clase; es decir, que alguna clase esté *sobrerrepresentada* en el conjunto. Por ejemplo, esto puede ocurrir al intentar detectar *spam* o fraudes.

Imaginemos que el conjunto de datos de cáncer que hemos usado en esta sección consite en 90 % de pacientes saludables: podemos obtener 90 % de exactitud (*accuracy*) sin necesidad de usar un algoritmo de aprendizaje automático, simplemente al elegir *siempre* la clases mayoritaria (tumor benigno). Por tanto, si tenemos un algoritmo que obtiene un valor de la exactitud cercano al 90 % podemos afrimar que el no aprendió algo útil a partir del conjunto de datos de entrenamiento.

En esta sección revisamos algunas técnicas que pueden ayudar a enfrentar conjuntos de datos desbalanceados. Primero, crearemos un conjunto desbalanceado; nuestro conjunto de datos original consta de 357 tumores benignos (clase 0) y 212 tumores malignos (clase 1).

Para nuestro conjunto desbalanceado tomamos las 357 muestras de tumores benignos y apliamos (*vstack*, *hstack*) 40 de tumores malignos; si calculamos la exactitud de un modelo que predice la clase mayoritaría, obtenemos un valor cercano a 90 %:

```
X_imb = np.vstack((X[y==0], X[y==1][:40]))
y_imb = np.hstack((y[y==0], y[y==1][:40]))

y_pred = np.zeros(y_imb.shape[0])
np.mean(y_pred==y_imb) * 100
```

```
89.92443324937027
```

Por lo tanto, cuando trabajamos con conjuntos de datos desbalanceados es recomendable concentrarnos en otras métricas. Por ejemplo si nos interesa identificar pacientes con tumores malignos para recomendar revisiones adicionales, entonces la métrica *recall* es la mejor elección. Para filtrado de *spam*, donde no queremos que un mensaje sea marcado como *spam* si el sistema no es muy certero, la *precision* sería una métrica más apropiada.

Por otro lado, un conjunto de datos desbalanceado influye en el entrenamiento de los modelos. Típicamente un algoritmo de aprendizaje automático optimizan una *función de recompensa* (o costo) que se calcula con una suma sobre las muestras de entrenamiento que observa durante su ajuste: es muy probable que la regla de decisión estará sesgada hacia la clase mayoritaria.

Una forma de enfrentar proporciones de clase desbalanceadas durante el ajuste del modelo es asignar mayor *penalización* a las predicciones erróneas de la clase mayoritaria. En *scikit-learn* ajustar esta penalización se puede realizar estableciendo el parámetro *class_ weight='balanced'*, que está disponible en la mayoría de los clasificadores.

Otras estrategias muy usadas para tratar las clases desbalanceadas son *sobremuestrear* (*upsamplig*) la clase minoritaria, *submuestrar* (*downsamplig*) la clase mayoritaria y la generación de muestras de entrenamiento *sintéticas*. Desafortunadamente no existe una solución universal, por esto es recomendable probar diferentes estrategias para un problema dado, evaluar los resultados y elegir aquella que entregue el mejor comportamiento.

Dentro de *scikit-learn* existe la función *resample* que ayuda a sobremuestrear la clase minoritaria seleccionando nuevas muestras del conjunto de datos con reemplazo. El código siguiente toma la clase minoritaria del conjunto de cáncer de pecho (clase 1) e iterativamente toma nuevas muestras hasta que contiene el mismo número de muestras que la clase 0:

```
from sklearn.utils import resample

print('Número de muestras de la clase 1 (antes) : ',X_imb[y_imb==1].shape[0])
```

```
Número de muestras de la clase 1 (antes) :  40
```

```
X_upsampled, y_upsampled = resample(X_imb[y_imb==1], y_imb[y_imb==1],
                                    n_samples=X_imb[y_imb==0].shape[0],
                                    replace=True, random_state=123)

print('Número de muestras de la clase 1 (después) : ',X_upsampled.shape[0])
```

```
Número de muestras de la clase 1 (después) :  357
```

Después de volver a muestrear, podemos volver a apilar la clase 0 original con la clase 1 sobremuestrada para obtener un conjunto de datos balanceado; de forma que la predicción sobre el conjunto antes mayoritario sólo obtenga el 50 % de exactitud:

```
X_bal = np.vstack((X[y==0], X_upsampled))
y_bal = np.hstack((y[y==0], y_upsampled))

y_pred = np.zeros(y_bal.shape[0])
np.mean(y_pred==y_bal) * 100
```

```
50.0
```

Para submuestrar la clase mayoritaria, se puede usar la función *resample* invirtiendo las etiquetas de clase en el código anterior.

Existe una biblioteca de Python llamada *imbalanced-learn* enfocada en el problema de conjuntos desbalanceados; aún se encuentra en desarrollo y se puede consultar en:

https://github.com/scikit-learn-contrib/imbalanced-learn