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 *hiper-parámetros*

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-wisc
df = pd.read_csv('https://bit.ly/3gob0mX', header=None)
```

Asignamos las 30 características al arreglo **X** y con ayuda de *LabelEncoder* transformamos las etiquetas de clase a enteros:

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

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
```

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.

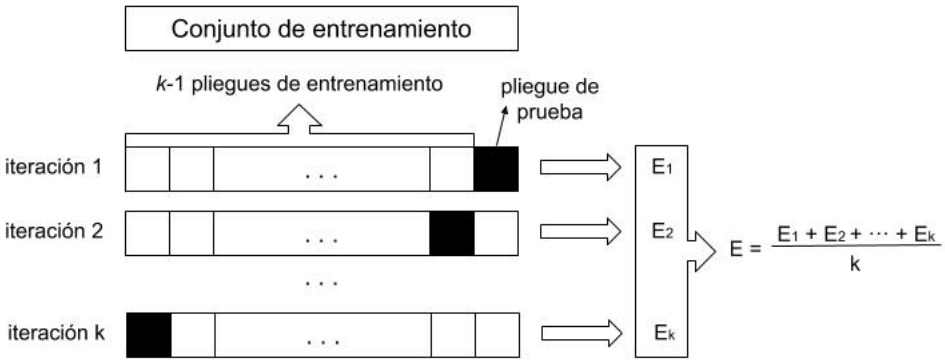


Figura 1.1: Método de validación cruzada de *k-pliegues*

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:

```
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:

```
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.

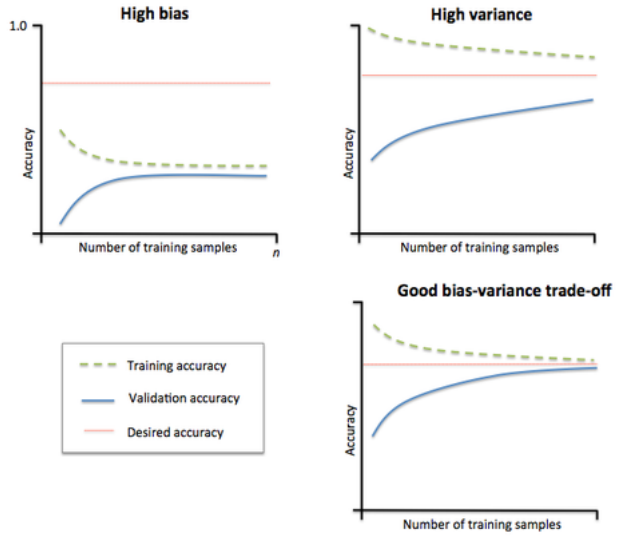


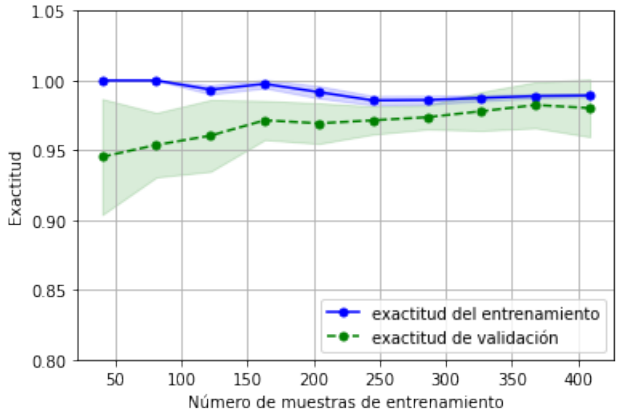
Figura 1.2: Balance sesgo-varianza (*bias-variance trade-off*)

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 entrenamiento, 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:

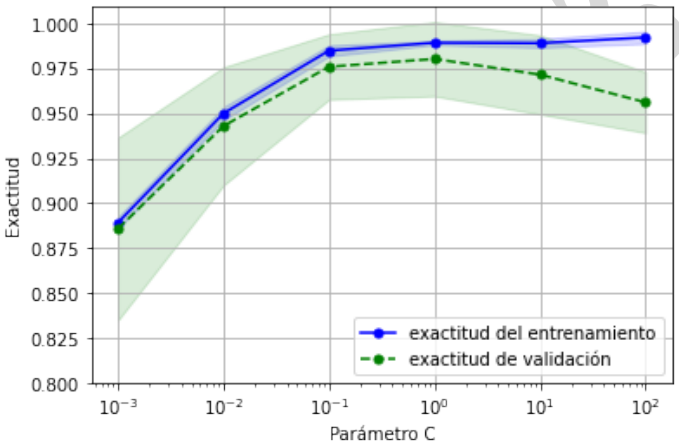


Con el parámetro *train_ sizes*, se puede controlar el número (absoluto o relativo) de muestras de entrenamiento usadas para generar las curvas de aprendizaje. En nuestro ejemplo, usamos 10 intervalos relativos igualmente espaciados para los tamaños de los conjuntos de entrenamiento. Por omisión, se 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 entrenamiento. 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 validación 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*:



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, también 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
```

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