

APRENDIZAJE SUPERVISADO CON SCIKIT-LEARN

Abraham Jain Jiménez

1 Aprendizaje Supervisado (SL)

Una *learning machine* es un objeto Θ que puede ser conceptualizado como una función que mapea una base de datos D_f a un estimador \hat{f} de la función original f :

$$\Theta : D_f \rightarrow \hat{f}$$

El estimador \hat{f} es producido por el modelo de Machine Learning implementado para el problema.

En aprendizaje supervisado se entrena el modelo a partir de un conjunto de pares de datos $D_{\text{train}} = \{\vec{x}_i, f(\vec{x}_i) = y_i\}_{i=1}^N$, en donde el conjunto de $\vec{x}_i \in \mathbb{R}^d$ son los vectores de características (comúnmente abordados como vectores columna), y y_i es la etiqueta asociada al problema de regresión o clasificación.

En el caso de clasificación, el espectro de y_i es discreto, mientras que en el caso de regresión toma valores continuos.

2 Sintaxis general de scikit-learn para SL

Sintaxis para implementación de un modelo:

```
1 from sklearn.module import Model
2 model = Model()
3 model.fit(X, y)
4 predictions = model.predict(X_new)
5 print(predictions)
```

En donde X representa a la matriz de diseño asociada a los datos (la que concatena los vectores columna \vec{x}_i):

$$X := [\vec{x}_1, \dots, \vec{x}_N] \in \mathbb{R}^{d \times N}$$

Así mismo, y representa al vector de etiquetas:

$$\vec{y} = \begin{pmatrix} y_1 \\ \vdots \\ y_N \end{pmatrix}$$

(De ahí la notación X , y , con mayúscula y minúscula respectivamente).

Módulos de aprendizaje supervisado en `scikit-learn`:

Módulo	Modelos	Ejemplo de clase
<code>sklearn.linear_model</code>	Modelos lineales (regresión, clasificación)	<code>LogisticRegression</code>
<code>sklearn.neighbors</code>	K-NN	<code>KNeighborsClassifier</code>
<code>sklearn.tree</code>	Árboles de decisión	<code>DecisionTreeClassifier</code>
<code>sklearn.ensemble</code>	Ensamblados	<code>RandomForestClassifier</code>
<code>sklearn.svm</code>	SVM	<code>SVC</code>
<code>sklearn.model_selection</code>	Validación cruzada, splits	<code>train_test_split</code>
<code>sklearn.metrics</code>	Métricas de evaluación	<code>confusion_matrix</code>

3 Selección del modelo

La meta de un modelo de ML es lograr un error de generalización tan bajo como sea posible. Esto se hace mediante un proceso llamado *cross-validation* (validación cruzada).

La idea de la validación cruzada es dividir el conjunto de datos original en:

- Training Set: El conjunto de entrenamiento. En donde ajustas el modelo de ML a los datos del problema.
- Development Set: El conjunto donde evalúas el rendimiento del modelo. En donde se toman las decisiones de la selección del modelo y se ajustan los hiperparámetros.

Para este proceso se tienen dos algoritmos populares de partición: *Simple Cross-Validation* y *K-Fold Cross-Validation*.

3.1 Algoritmo *Simple Cross-Validation*

Sean $M = \{M_1, \dots, M_d\}$ un conjunto finito de modelos, S el dataset original, $S_{train} = \{\vec{x}, f(\vec{x}) = y\}$ el training set, y $S_{dev} = \{\vec{x}^*, y^*\}$ el dev set. El algoritmo de validación cruzada simple es:

1. Dividir S aleatoriamente en S_{train} y S_{dev} (una partición común es 70% vs 30% respectivamente).
 2. Entrenar cada modelo M_i únicamente sobre S_{train} para encontrar el estimador \hat{f}_i .
 3. Seleccionar el \hat{f}_i con el menor error de generalización ($Err(\hat{f})$) en el S_{dev} .
-

3.2 Algoritmo *K-Fold Cross-Validation*

Retomando los conjuntos definidos en el algoritmo anterior, en donde S_{train} es el conjunto de entrenamiento con m muestras, el algoritmo de validación cruzada por k -pliegues es:

1. Dividir S_{train} aleatoriamente en k subconjuntos de m/k pares de muestras de entrenamiento $(\vec{x}, f(\vec{x}))$, los cuales serán denotados por S_1, \dots, S_k .
2. Cada $M_i \in M$ se evalúa como sigue:
Para $j = 1, \dots, K$
 - Entrenar el modelo M_i en $S_1 \cup \dots \cup S_{j-1} \cup S_{j+1} \cup \dots \cup S_K$ (es decir, entrenarlo en todas las particiones menos en S_j) para obtener un estimador \hat{f}_{ij} .
 - Evaluar \hat{f}_{ij} en S_j para obtener un estimador del error $\widehat{Err}_{S_j}(\hat{f}_{ij})$.
 - El error de generalización estimado de M_i es calculado como el promedio de los $\{\widehat{Err}_{S_j}(\hat{f}_{ij})\}$ sobre j :

$$\text{Error Promedio} = \frac{1}{K} \sum_{i=1}^K \widehat{Err}_i$$

3. Se toma el modelo M_i con el menor error de generalización estimado y se retiene en S_{train} . El estimador \hat{f} es la respuesta final.

El estimador del error de generalización es una métrica asociada al rendimiento del modelo. Por ejemplo, en regresión multilíneal podría ser R^2 , en tal caso el error promedio es $R^2_{\text{promedio}} = \frac{1}{K} \sum_{i=1}^K R_i^2$.

4 División del conjunto de datos en scikit-learn

Sintaxis:

```
1 from sklearn.model_selection import train_test_split
2 X_train, X_test, y_train, y_test = train_test_split(X,
3                                                     y,
4                                                     test_size,
5                                                     random_state,
6                                                     stratify)
7 model = Model()
8 model.fit(X_train, y_train)
9 print(model.score(X_test, y_test))
```

4.1 Argumentos de la función `train_test_split`

- **test_size**: Proporción del conjunto de test. Por ejemplo, `test_size = 0.2` implica un 20% del dataset para test. También puede ser un entero (número de muestras). Si no se incluye este argumento o se asigna `None`, el valor por defecto es 0.25.
- **random_state**: Fija la semilla generadora de valores aleatorios para que el proceso de división sea reproducible.
- **stratify**: Garantiza que la proporción de clases en los conjuntos de entrenamiento y prueba sea similar a la del conjunto original. Típicamente se utiliza `stratify = y`.

5 K-Fold Cross-Validation en scikit-learn

Sintaxis para K -folds:

```
1 from sklearn.model_selection import cross_val_score, KFold
2 kf = KFold(n_splits, shuffle, random_state)
3 model = Model()
4 cv_results = cross_val_score(model, X, y, cv=kf)
```

5.1 Estructura de KFold()

- `n_splits` define el número de particiones (folds) de la validación.
- `shuffle` indica si se mezclan los datos o no (`True`, `False`).
- `random_state` define la semilla para reproducibilidad.

Función `cross_val_score()`

Devuelve un arreglo NumPy con la puntuación de cada fold. Usa por defecto la métrica default del modelo, `score()`.

6 Optimización de hiperparámetros en scikit-learn

En un modelo de ML, los hiperparámetros son configuraciones del modelo que se definen para el proceso de entrenamiento. Dado que estos precisamente afectan cómo se entrena el modelo, deben optimizarse para que el rendimiento sea el mejor posible de acuerdo a las necesidades de cada proyecto.

Por ejemplo, en los modelos K-NN y K-means (aunque este último es no supervisado), K es un hiperparámetro. En regresión polinomial, el orden M del polinomio (número de parámetros entrenables) es un hiperparámetro. En árboles de decisión, `max_depth` lo es. Etc.

Algunas de las técnicas de optimización de hiperparámetros, es decir, técnicas para encontrar los mejores hiperparámetros de un modelo de ML son *Grid Search Cross-Validation* y *Randomized Search Cross-Validation*.

6.1 Algoritmo Grid Search Cross-Validation

Siendo que los valores de un hiperparámetro pueden tener un espectro continuo o discreto, el algoritmo Grid Search CV es el siguiente:

1. Se define un espacio de búsqueda:

Se toma un conjunto de hiperparámetros y un conjunto de valores posibles para cada uno de ellos.

Sea $H = \{H_i\}_{i=1}^N$ el conjunto de hiperparámetros a ajustar y $V_i = \{a_j^i\}_{j=1}^n$ el conjunto de valores que puede tomar el hiperparámetro H_i (si el espectro de este conjunto es continuo, estamos hablando de valores dentro de un rango).

```

1 param_grid = {
2     "H_1": [lista (o rango continuo) de valores de H_1],
3     .....
4     "H_N": [lista (o rango continuo) de valores de H_N]
5 }

```

2. Se construye la malla:

Se forma una lista de todas las combinaciones posibles de esos hiperparámetros y sus valores, un conjunto de pares (Hiperparámetro, Valor):

$$\text{Malla} = \{(H_i, a_j^i)\}$$

3. Se evalúa cada combinación por K-Fold Cross-Validation:

Se implementa un proceso K-Fold CV en cada par (H_i, a_j^i) para encontrar la mejor combinación de hiperparámetros que optimiza (maximiza o minimiza según el caso) una métrica de rendimiento del modelo.

6.2 Algoritmo Randomized Search Cross-Validation

La desventaja con Grid search es que es costoso computacionalmente, el consumo de tiempo y recursos suele ser alto, por lo que random search es una alternativa menos costosa.

Este algoritmo selecciona hiperparámetros aleatoriamente, crea un conjunto y entrena el modelo con él. Puede que no encuentre el mejor conjunto, pero hay mayores probabilidades de encontrar uno cercano al mejor, lo que ahorra mucho costo computacional.

Retomando los términos vistos en Grid Search CV, el algoritmo Randomized Search CV es el siguiente:

1. Se define un espacio de búsqueda:

A diferencia de Grid Search CV, Randomized Search CV no necesita recibir una lista exacta de valores para los hiperparámetros, se pueden dar distribuciones.

2. Se toman N combinaciones de hiperparámetros:

El algoritmo elige aleatoriamente N combinaciones de hiperparámetros y sus valores, y es el usuario el que define cuántas serán.

3. Se evalúa cada combinación por K-Fold Cross-Validation:

Con esto se calcula el promedio del score, o sea la métrica de rendimiento.

4. Elección de la mejor combinación de hiperparámetros:

Se elige la mejor combinación según el score promedio en el proceso K-Fold CV.

7 Grid Search CV en scikit-learn

```
1 from sklearn.model_selection import GridSearchCV
2 from sklearn.module import Model
3
4 #Datos
5 X, y = DataSet
6
7 # Definimos el modelo
8 model = Model()
9
10 # Definimos el grid de hiperparámetros
11 param_grid = {
12     'Hiperparámetro i-ésimo': [conjunto de valores para el hiperparámetro]
13 }
14
15 # Se implementa Grid Search CV. scoring recibe la métrica del problema
16 grid_search = GridSearchCV(estimator=model,
17                             param_grid=param_grid,
18                             cv, scoring)
19
20 # Entrenamiento
21 grid_search.fit(X, y)
```

```

22
23 # Resultados
24 print("Mejores parámetros:", grid_search.best_params_)
25 print("Mejor score promedio:", grid_search.best_score_)

```

8 Randomized Search CV en scikit-learn

```

1  from sklearn.model_selection import RandomizedSearchCV
2  from sklearn.model import Model
3  from scipy.stats import randint
4
5  # Datos
6  X, y = DataSet
7
8  # Modelo
9  model = Modelo()
10
11 # Espacio de búsqueda (distribuciones)
12 param_distributions = {
13     'Hiperparámetro': randint(x_inicial, x_final)
14 }
15
16 # Búsqueda aleatoria
17 random_search = RandomizedSearchCV(
18     estimator=model,
19     param_distributions=param_distributions,
20     n_iter,           # Número entero de combinaciones a tomar
21     cv,              # Número de folds
22     scoring,         # Métrica
23     random_state
24 )
25
26 # Se efectúa el algoritmo Randomized Search
27 random_search.fit(X, y)
28
29 # Resultados
30 print("Mejores hiperparámetros:", random_search.best_params_)

```



```
31 print("Mejor score promedio:", random_search.best_score_)
```

9 Pipelines en scikit-learn

Una pipeline, comunmente traducido como canalización, es un objeto que sirve para ejecutar una serie de transformaciones y construir un modelo en un único flujo de trabajo.

Las pipelines son útiles para organizar el código (por ejemplo, en el caso en el que todo un equipo trabaja sobre un mismo notebook), para evitar fuga de datos (*data leakage*) y facilitar tareas como cross-validation y la búsqueda de hiperparámetros con `GridSearchCV` o `RandomizedSearchCV`.

9.1 Fuga de datos

La fuga de datos ocurre cuando se utiliza información del conjunto de prueba en procesos en los que únicamente debe usarse el conjunto de entrenamiento, ya sea durante el propio entrenamiento del modelo o durante las etapas de pre-procesamiento que afectan al entrenamiento. Esto genera un modelo que no generaliza bien, un modelo poco robusto.

Por las razones anteriores es que se suele dividir el conjunto de datos en training y test, por ejemplo, antes de la imputación de valores faltantes.

9.2 Sintaxis de las pipelines en scikit-learn

```
1 from sklearn.pipeline import Pipeline
2
3 pipe = Pipeline(steps)
```

En donde `steps` es una secuencia de N pasos donde cada paso desde el primero hasta el N-1 debe ser un transformador, es decir, de la clase `transformers` de `scikit-learn`. El último paso, N, es un estimador, un modelo de ML.

```
1 steps = [("transformación 1", Transformador1()),
2         .....
3         ("transformación N-1", TransformadorN-1()),
4         ("modelo", Model())]
```

9.3 Ejemplo: Escalando datos para aplicar un modelo K-NN mediante una pipeline

Mediante una pipeline, se van a escalar los datos mediante la transformación `StandardScaler()` como parte del preprocesamiento de datos para un modelo K-NN.

```
1 steps = [('scaler', StandardScaler()),
2          ('knn', KNeighborsClassifier(n_neighbors=6))]
3
4 pipeline = Pipeline(steps)
5
6 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
7                                                    random_state=21)
8 knn_scaled = pipeline.fit(X_train, y_train)
9 y_pred = knn_scaled.predict(X_test)
10 print(knn_scaled.score(X_test, y_test))
```