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:

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

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:

```
from sklearn.module import Model
model = Model()
model.fit(X, y)
predictions = model.predict(X_new)
print(predictions)
```

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

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

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**
`sklearn.linear_model`	Modelos lineales (regresión, clasificación)	`LogisticRegression`
`sklearn.neighbors`	K-NN	`KNeighborsClassifier`
`sklearn.tree`	Árboles de decisión	`DecisionTreeClassifier`
`sklearn.ensemble`	Ensambles	`RandomForestClassifier`
`sklearn.svm`	SVM	`SVC`
`sklearn.model_selection`	Validación cruzada, splits	`train_test_split`
`sklearn.metrics`	Métricas de evaluación	`confusion_matrix`

---

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 $M = \{M_1, \ldots, M_d\}$ 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 $S_1, \ldots, S_k$.
2. Cada $M_i \in M$ se evalúa como sigue:

   Para $j = 1, \ldots, K$

   - Entrenar el modelo M_i en $S_1 \cup \cdots \cup S_{j-1} \cup S_{j+1} \cup \cdots \cup S_K$ (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 multilineal 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:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    test_size,
                                                    random_state,
                                                    stratify)
model = Model()
model.fit(X_train, y_train)
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`: Se usa en problemas de clasificación. Este argumento 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`. Es decir, este argumento mantiene la proporción de clases en el conjunto de entrenamiento y prueba, es muy útil en desbalance de clases.

4.2 Desbalance de clases en un problema de clasificación

El argumento `stratify = y` genera lo que se conoce como muestras estratificadas (*stratified samples*). Una muestra estratificada es una muestra que se selecciona de manera que las proporciones de ciertos grupos, en este caso las divisiones del conjunto original, se mantengan igual que en la población original.

Por ejemplo, si en un conjunto de datos el 70% de las observaciones pertenece a la clase A y el 30% a la clase B, una muestra estratificada mantendrá esa misma proporción.

Evaluando si hay desbalance de clases antes de dividir el conjunto:

```
#Balance de clases del dataframe original
print(y["columna etiqueta"].value_counts())

#Balance de clases del conjunto train
print(y_train["columna etiqueta"].value_counts())

#Balance de clases del conjunto test
print(y_test["columna etiqueta"].value_counts())
```

---

5 K-Fold Cross-Validation en `scikit-learn`

Sintaxis para K-folds:

```
from sklearn.model_selection import cross_val_score, KFold
kf = KFold(n_splits, shuffle, random_state)
model = Model()
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 de 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).

```
param_grid = {
    "H_1": [lista (o rango continuo) de valores de H_1],
                .....................................
    "H_N": [lista (o rango continuo) de valores de H_N]
}
```

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 encontre 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
```

```
#Datos
X, y = DataSet

# Definimos el modelo
model = Model()

# Definimos el grid de hiperparámetros
param_grid = {
    'Hiperparámetro i-ésimo': [conjunto de valores para el hiperparámetro]
}

# Se implementa Grid Search CV. scoring recibe la métrica del problema
grid_search = GridSearchCV(estimator=model,
                           param_grid=param_grid,
                           cv, scoring)

# Entrenamiento
grid_search.fit(X, y)

# Resultados
print("Mejores parámetros:", grid_search.best_params_)
print("Mejor score promedio:", grid_search.best_score_)
```

8 Randomized Search CV en `scikit-learn`

```
from sklearn.model_selection import RandomizedSearchCV
from sklearn.module import Model
from scipy.stats import randint

# Datos
X, y = DataSet

# Modelo
model = Modelo()

# Espacio de búsqueda (distribuciones)
param_distributions = {
    'Hiperparámetro': randint(x_inicial, x_final)
```

```
}

# Búsqueda aleatoria
random_search = RandomizedSearchCV(
    estimator=model,
    param_distributions=param_distributions,
    n_iter,            # Número entero de combinaciones a tomar
    cv,                 # Número de folds
    scoring,          # Métrica
    random_state
)

# Se efectúa el algoritmo Randomized Search
random_search.fit(X, y)

# Resultados
print("Mejores hiperparámetros:", random_search.best_params_)
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 preprocesamiento 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`

```
from sklearn.pipeline import Pipeline

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.

```
steps = [("transformación 1", Transformador1()),
         ..................................
         ("transformación N-1", TransformadorN-1()),
         ("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.

```
steps = [('scaler', StandardScaler()),
         ('knn', KNeighborsClassifier(n_neighbors=6))]

pipeline = Pipeline(steps)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
                                                    random_state=21)
knn_scaled = pipeline.fit(X_train, y_train)
y_pred = knn_scaled.predict(X_test)
print(knn_scaled.score(X_test, y_test))
```