APRENDIZAJE SUPERVISADO CON SCIKIT-LEARN

Abraham Jain Jiménez

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.

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`

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*.

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}.

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 de 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.
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.

Simple Cross-Validation en `scikit-learn`

Sintaxis para la selección de modelos:

```
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))
```

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`.

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)
```

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()`.