Validación Cruzada en `Scikit-Learn`

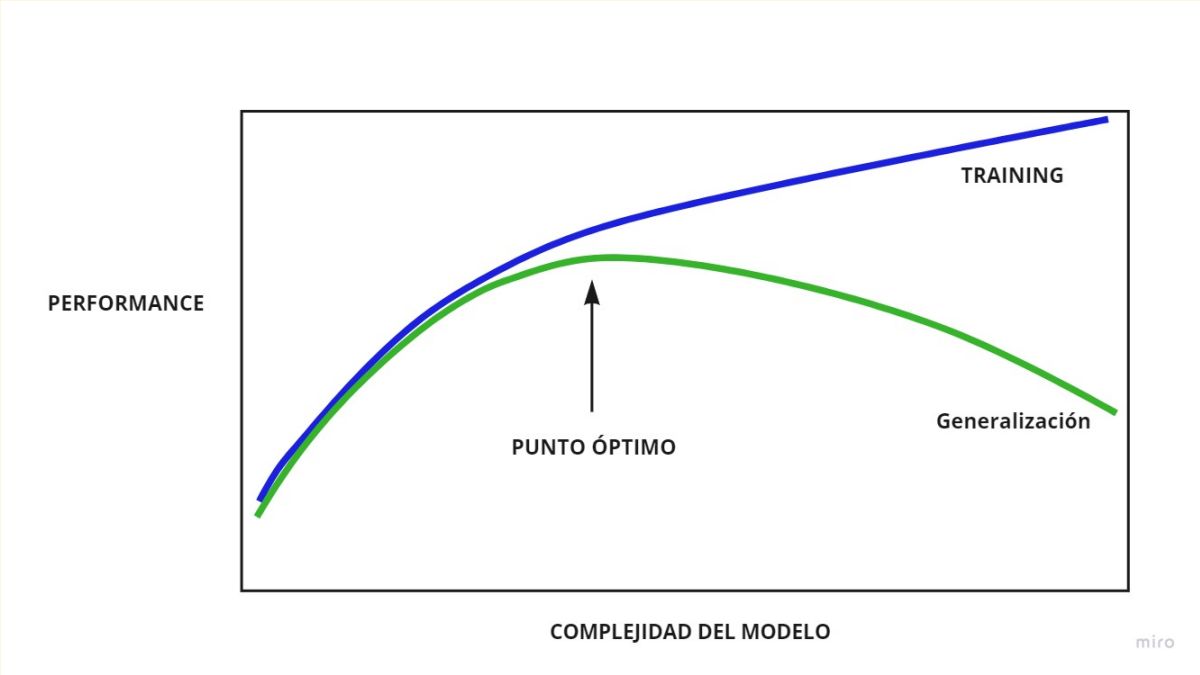
Alfonso Tobar Arancibia

Data Scientist

27-05-2020

github.com/datacubeR/clases-ml/Slides

Cómo Mejorar un Modelo (de nuevo?)

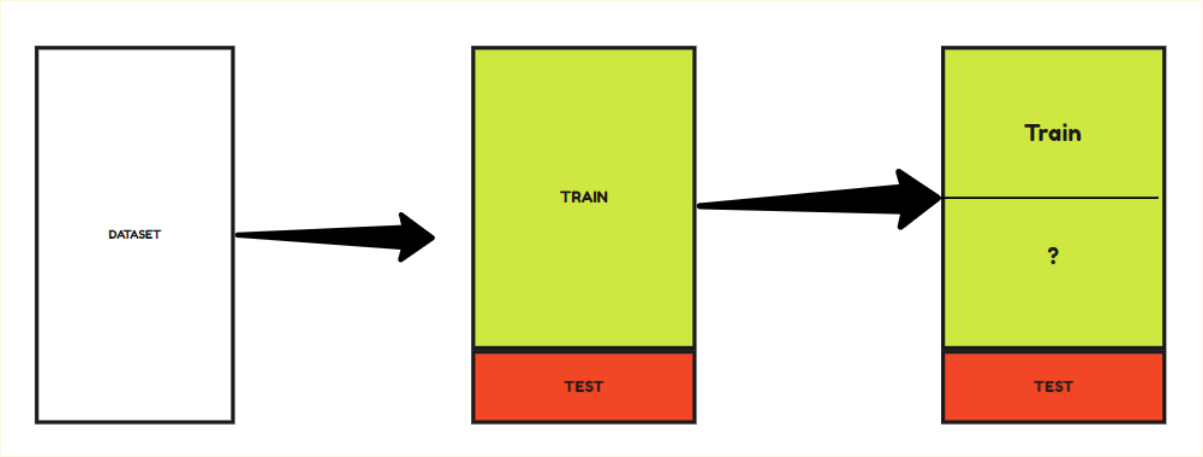


Cuando se prueba un algoritmo no se trata de utilizar un modelo y **si dio bien lo dejamos y si no lo botamos**. Normalmente muchos modelos pueden ser afinados, modificando su complejidad. Esta complejidad se varía mediante los **hiperparámetros de un modelo**. Entender los **hiperpárametros** de un modelo es complejo y es necesario entender otros conceptos como la Regularización.

Problema

El problema que esto suscita es: **¿cómo medimos el uso de distintos hiperparámetros (distintas complejidades) si para probar tenemos sólo el test set? Sería como hacer trampa**. Ya que sabemos que un modelo funciona mejor porque lo probamos en la data que me interesa, no porque de antemano sepamos que generaliza mejor (Sería como generar un overfit al test set).

Generar otro Split



Three Fold Split (Holdout)

En el Three Fold Split el Dataset se divide en 3 partes:

- **Training Set**: Utilizado para entrenar el modelo con distintos Hiperparámetros.
- **Validation Set**: Utilizado para encontrar el mejor set de hiperparámetros.
- **Test Set**: Utilizado para la evaluación final del modelo.


```
X_trainval, X_test, y_trainval, y_test = train_test_split(X,y, test_size = 0.2)
X_train, X_val, y_train, y_val = train_test_split(X_trainval, y_trainval, test_size = 0.2)
```

K-Fold Cross Validation

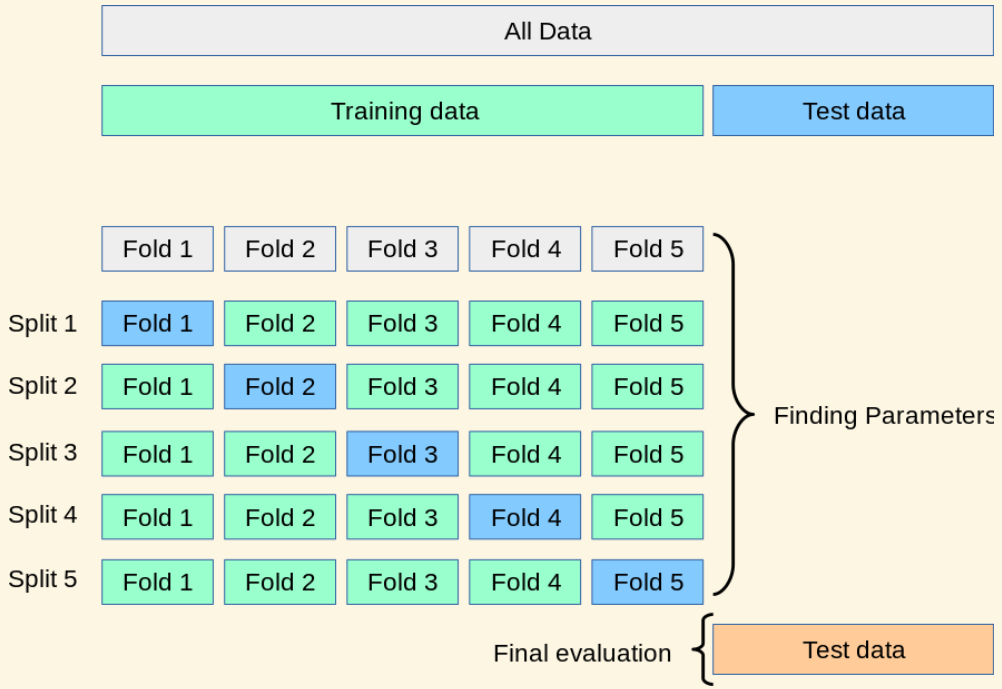
Pros:

- Mecanismo mucho más robusto para validar.
- Todas las muestras se usan en entrenamiento k-1 veces y una vez de validación.

Contras:

- Más costoso computacionalmente
- Mayores tiempos de entrenamiento

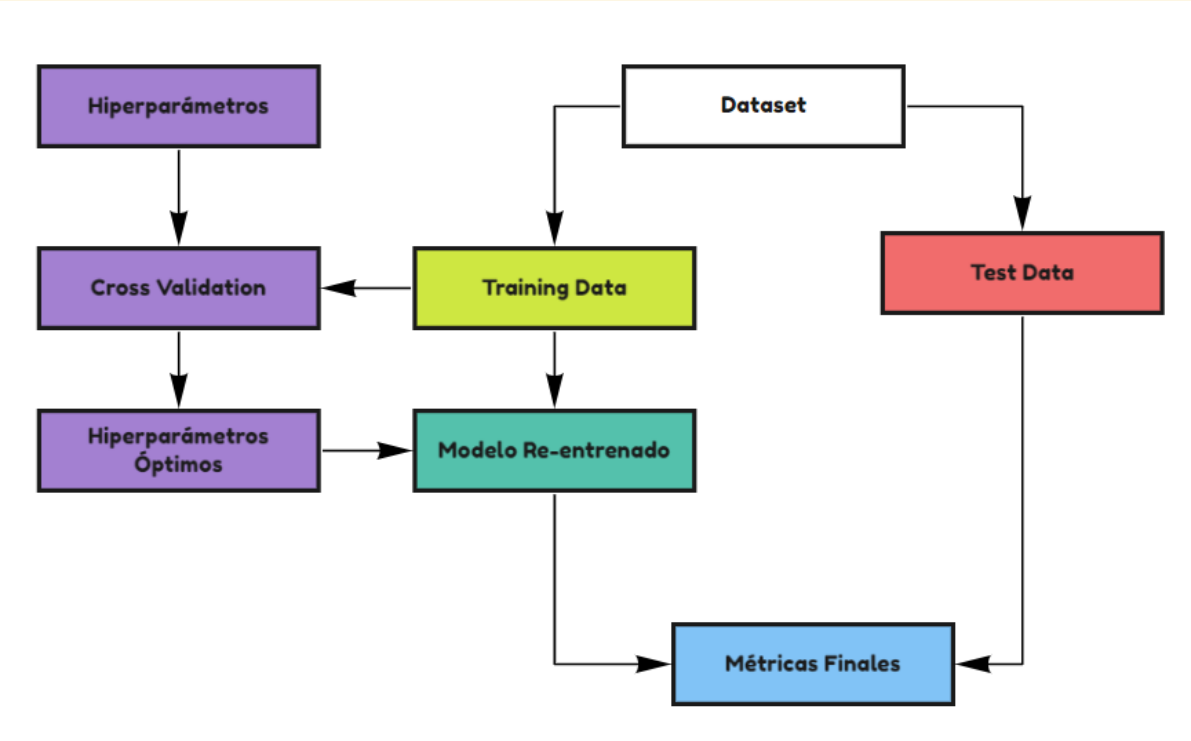
> NOTA: Normalmente K = 5 o 10 son los valores mayormente utilizados.



* Imágen obtenida del sitio oficial de `Scikit-Learn`.

Esquema de Validación

* Esquema adaptado del sitio oficial de `Scikit-Learn`



> NOTA: **Cross Validation** indica qué modelo usar mientras que el **evaluar en el Test Set** indica qué tan bueno es el modelo elegido.

GridSearchCV / Implementación por Fuerza Bruta

`GridSearchCV` es un meta-estimator, es decir, un Estimator que toma como argumento otro Estimator, por lo tanto hereda las propiedades del Estimator inicial tales como: `.fit()`, `.predict()`, etc.

```
from sklearn.model_selection import GridSearchCV

params = {'n_neighbors': np.arange(1,17,2)}
knn = KNeighborsClassifier()
search = GridSearchCV(knn, params, cv = 5, return_train_score = True, n_jobs = -1)
search.fit(X_train, y_train)

# Resultados
print('Mejor Cross Val Score (Mean)', search.best_score_) # en Cross Validation
print('Mejor K:', search.best_params_)# Hiperparámetro óptimo

print('Mejor Score en el Test Set:', search.score(X_test, y_test)) # en el modelo reentrenado
```

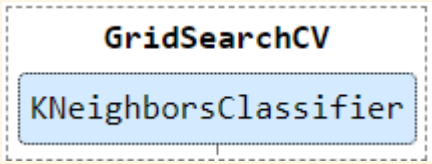
```
Mejor Cross Val Score (Mean) 0.6769427755343249
Mejor K: {'n_neighbors': 7}
Mejor Score en el Test Set: 0.7039106145251397
```

GridSearchCV / Fuerza Bruta

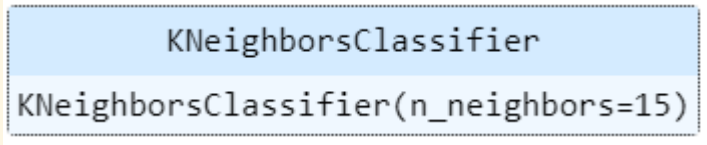
```
search.cv_results_
```

mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_n_neighbors	params	split0_test_score	split1_test_score	split2_test_score	mean_test_score	std_test_score	rank_test_score	split0_train_score	split1_train_score	split2_train_score	mean_train_score	std_train_score
0.004001	5.947204e-07	0.015665	0.000944	1	{'n_neighbors': 1}	0.621849	0.662447	0.616034	0.633443	0.020646	8	0.962025	0.955789	0.962105	0.959973	0.002959
0.005332	4.722135e-04	0.015999	0.001413	3	{'n_neighbors': 3}	0.642857	0.696203	0.632911	0.657324	0.027790	3	0.833333	0.795789	0.795789	0.808304	0.017698
0.004668	4.710899e-04	0.019007	0.001418	5	{'n_neighbors': 5}	0.680672	0.658228	0.624473	0.654458	0.023098	6	0.778481	0.755789	0.762105	0.765459	0.009562
0.004665	4.771033e-04	0.018664	0.003091	7	{'n_neighbors': 7}	0.651261	0.645570	0.658228	0.651686	0.005176	7	0.759494	0.738947	0.732632	0.743691	0.011468
0.005662	9.461827e-04	0.028335	0.012658	9	{'n_neighbors': 9}	0.630252	0.713080	0.666667	0.670000	0.033896	2	0.740506	0.715789	0.720000	0.725432	0.010797
0.005000	8.177028e-04	0.039000	0.010198	11	{'n_neighbors': 11}	0.617647	0.691983	0.658228	0.655953	0.030390	4	0.725738	0.709474	0.707368	0.714194	0.008209
0.008332	6.127815e-03	0.020667	0.003399	13	{'n_neighbors': 13}	0.613445	0.675105	0.675105	0.654552	0.029067	5	0.734177	0.698947	0.711579	0.714901	0.014573
0.006333	1.886157e-03	0.019666	0.002867	15	{'n_neighbors': 15}	0.630252	0.683544	0.696203	0.670000	0.028577	1	0.732068	0.703158	0.696842	0.710689	0.015335

```
search
```




```
search.best_estimator_
```



Otras metodologías de Cross Validation.

Todas estas metodologías pueden utilizarse en conjunto con GridSearchCV e la siguiente forma:

```
from sklearn.model_selection import LeaveOneOut, ShuffleSplit
search = GridSearchCV(knn, param_grid = params, cv = LeaveOneOut(), n_jobs = -1)
```

- **Leave One Out**: Deja una muestra cómo validación. Extremadamente lento, por lo que se recomienda sólo cuando hay muy poca data disponible.

```
LeaveOneOut()
```

- **ShuffleSplit**: También se conoce como MonteCarlo y genera nuevas muestras sintéticas. Estas muestran permiten que haya repetición.

```
ShuffleSplit(n_splits = 5)
```

GridSearchCV / Fuerza Bruta

- **RepeatedKFold**: Corresponde a un procedimiento que se repite N veces. Entre cada repetición la data se revuelve aleatoriamente. Es uno de los procedimientos más robustos.

```
from sklearn.model_selection import RepeatedKFold, RepeatedStratifiedKFold
RepeatedKFold(n_splits = 5, n_repeats = 10)
RepeatedStratifiedKFold(n_splits = 5, n_repeats = 10)
```

- **TimeSeriesSplit**: Corresponde a un procedimiento divide data, tal que siempre se entrene con el pasado y se prediga/evalúe en el futuro.

```
from sklearn.model_selection import TimeSeriesSplit
TimeSeriesSplit(n_splits = 5)
```

- Para más información de todos los tipos de CV existentes Ver docs

RandomizedSearchCV

`RandomizedSearchCV()` es otro meta-estimator que permite generar una búsqueda de hiperparámetros de manera randomizada. Es decir, en vez de realizar cada combinación de hiperparámetros aleatoriamente elige un número dado.

Pros: Mucho más rápido que GridSearchCV, además uno sabe a priori cuantos modelosentrenará y aproximadamente cuánto tiempo demorará.

Contra: No es exhaustiva y podría no encontrar el valor óptimo.

```
from sklearn.model_selection import RandomizedSearchCV
RandomizedSearchCV(knn, params, n_iter = 10, random_state = 123, n_jobs = -1, cv = 5)
```

> Existe otro tipo de optimización llamada `Optimización Bayesiana`, la cual elige el siguiente set de hiperparámetros a probar dependiendo de los resultados de la iteración anterior. Para aprender más al respecto ver acá.

Scoring

El CV tiene como objetivo último el entregar el mejor de set de hiperparámetros que entreguen el mejor modelo. Pero ¿qué significa el **MEJOR MODELO**?. El **MEJOR MODELO** depende de la métrica que escojamos, por ende, es un parámetro que deberíamos utilizar en Grid/RandomizedSearch.

El parámetro **scoring** es un string que indicará al CV bajo qué criterio encontrar el mejor modelo.

Scoring	Function
Classification	
'accuracy'	metrics.accuracy_score
'balanced_accuracy'	metrics.balanced_accuracy_score
'average_precision'	metrics.average_precision_score
'neg_brier_score'	metrics.brier_score_loss
'f1'	metrics.f1_score
'f1_micro'	metrics.f1_score
'f1_macro'	metrics.f1_score
'f1_weighted'	metrics.f1_score
'f1_samples'	metrics.f1_score
'neg_log_loss'	metrics.log_loss
'precision' etc.	metrics.precision_score
'recall' etc.	metrics.recall_score
'jaccard' etc.	metrics.jaccard_score
'roc_auc'	metrics.roc_auc_score

Regression	
'explained_variance'	metrics.explained_variance_score
'max_error'	metrics.max_error
'neg_mean_absolute_error'	metrics.mean_absolute_error
'neg_mean_squared_error'	metrics.mean_squared_error
'neg_root_mean_squared_error'	metrics.mean_squared_error
'neg_mean_squared_log_error'	metrics.mean_squared_log_error
'neg_median_absolute_error'	metrics.median_absolute_error
'r2'	metrics.r2_score
'neg_mean_poisson_deviance'	metrics.mean_poisson_deviance
'neg_mean_gamma_deviance'	metrics.mean_gamma_deviance

* Valores sacados de la página oficial de `Scikit-learn`. Para más información entrar acá.

Tarea

- Utilizar KNN para resolver un problema de Regresión utilizando el dataset Diabetes.
    - Encontrar el número del K óptimo utilizando las siguientes metodologías:
    - Holdout
    - KFold
    - ShuffleSplit
    - RepeatedKFold
- Utilizar KNN para resolver un problema de Clasificación utilizando el dataset Iris.
    - Encontrar el número del K óptimo utilizando las siguientes metodologías:
    - Holdout
    - StratifiedKFold
    - ShuffleSplit
    - RepeatedStratifiedKFold
- Pruebe sólo `25 iteraciones` utilizando RandomizedSearchCV con k desde 1 a 100. Utilice `random_state = 123`.
- Para los parámetros `n_splits` y `n_repeats` utilice el valor de 5.