Aprendizaje de Máquina II

Contenidos

El cronograma tentativo de clases va a ser:

- Clase 1 : Optimización de Hiperparámetros.Tunning.Implementación. Aplicaciones.
- Clase 2 : Base de Datos SQL y NoSQL.Implementación. Aplicaciones sobre Google Cloud.
- Clase 3 : Conceptos de Arquitectura ML + C.Computing. Infraestructura: Servers (en cloud).
- Clase 4 : MLOps. Deploy de Modelos de ML (Parte I).
- Clase 5 : MLOps. Deploy de Modelos de ML (Parte II).
- Clase 6 : Vertex AI.Implementación. Aplicaciones sobre Google Cloud.
- Clase 7 : MLFlow.Implementación. Aplicaciones.
- Clase 8 : TP Final.

Mecanismos de Evaluación

- Google Forms
- Trabajo práctico integrador con exposición

Clase 1:
Optimización de Hiperparámetros

Agenda

- Explicación: Optimización de Hiperparámetros
- Hands-On
- Break
- Buenas Prácticas de un Data Scientist
- Cierre

Optimización de Hiperparámetros

¿Cómo elegimos los mejores hiperparámetros para nuestro problema?

¿Cómo elegimos los mejores hiperparámetros para nuestro problema?

¿Qué es mejor? ¿Con respecto a exactitud? ¿Área bajo la curva ROC?

¿Cómo elegimos los
mejores hiperparámetros
para nuestro problema?

¿Qué es mejor? ¿Con respecto a exactitud?
¿Área bajo la curva ROC?

Primero, tenemos que definir una métrica a optimizar.

Una vez que sabemos la métrica que queremos optimizar,
¿cómo elegimos los mejores hiperparámetros para nuestro problema?

Solución más natural: probar a mano distintos valores de los hiperparámetros.

Una vez que sabemos la métrica que queremos optimizar,
¿cómo elegimos los mejores hiperparámetros para nuestro problema?

~~**Solución más natural:** probar a mano distintos valores de los hiperparámetros.~~

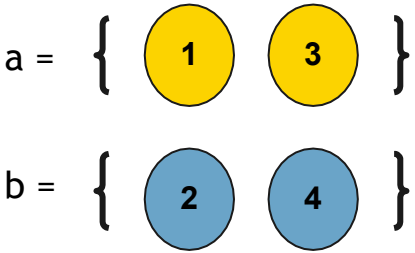
Cansador, tedioso y poco eficiente.

Una vez que sabemos la métrica que queremos optimizar,
¿cómo elegimos los mejores hiperparámetros para nuestro problema?

Solución ma natural: hacer una búsqueda exhaustiva. Es decir probando con todos los valores de los hiperparámetros que podamos y eligiendo la mejor combinación. Este método se llama **Grid Search** (“búsqueda de cuadrícula”).

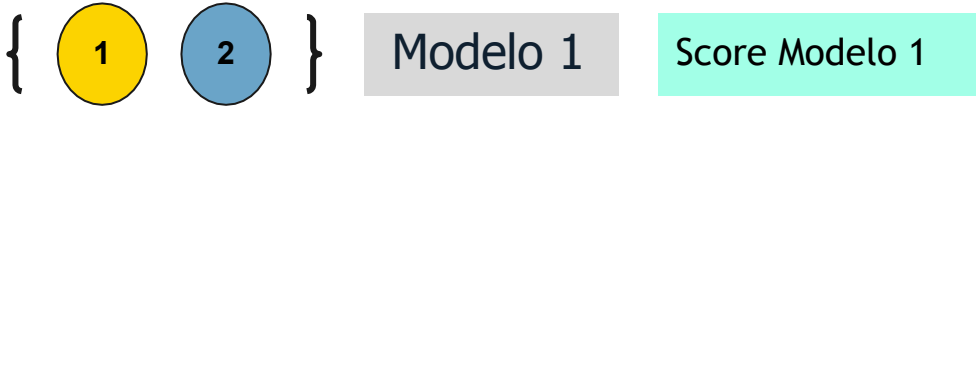
Solución ~~más~~ natural: hacer una búsqueda exhaustiva. Es decir probando con todos los valores de los hiperparámetros que podamos y eligiendo la mejor combinación. Éste método se llama Grid Search (“búsqueda de cuadrícula”).

Por ejemplo, si tenemos dos hiperparámetros, a y b, que pueden tomar valores a = {1,3} y b = {2,4}



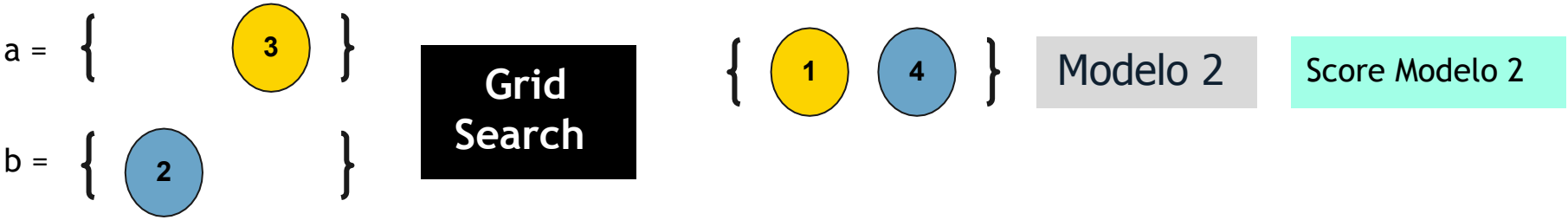
Solución ~~más~~ natural: hacer una búsqueda exhaustiva. Es decir probando con todos los valores de los hiperparámetros que podamos y eligiendo la mejor combinación. Éste método se llama Grid Search (“búsqueda de cuadrícula”).

Por ejemplo, si tenemos dos hiperparámetros, a y b, que pueden tomar valores a = {1,3} y b = {2,4}



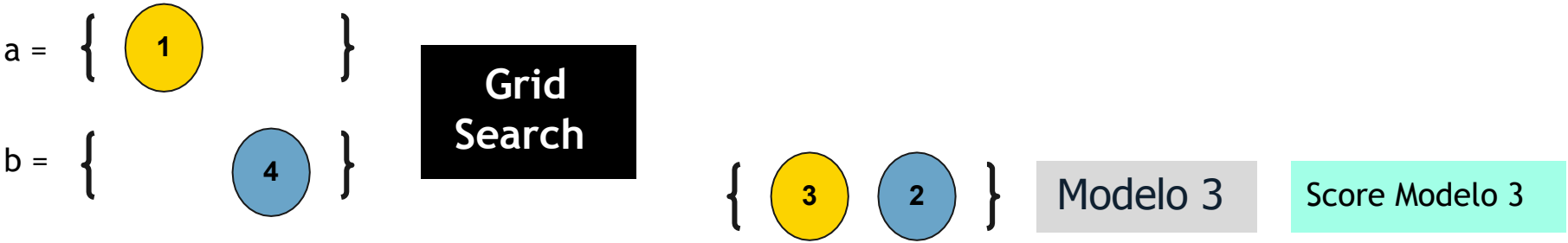
Solución ~~más~~ natural: hacer una búsqueda exhaustiva. Es decir probando con todos los valores de los hiperparámetros que podamos y eligiendo la mejor combinación. Éste método se llama Grid Search (“búsqueda de cuadrícula”).

Por ejemplo, si tenemos dos hiperparámetros, a y b, que pueden tomar valores a = {1,3} y b = {2,4}



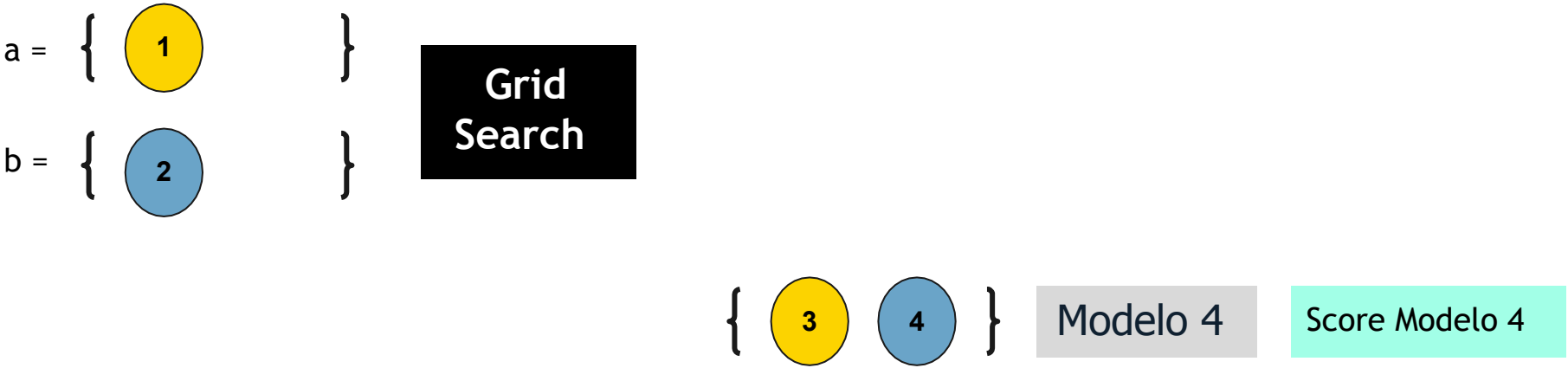
Solución ~~más~~ natural: hacer una búsqueda exhaustiva. Es decir probando con todos los valores de los hiperparámetros que podamos y eligiendo la mejor combinación. Éste método se llama Grid Search (“búsqueda de cuadrícula”).

Por ejemplo, si tenemos dos hiperparámetros, a y b, que pueden tomar valores a = {1,3} y b = {2,4}

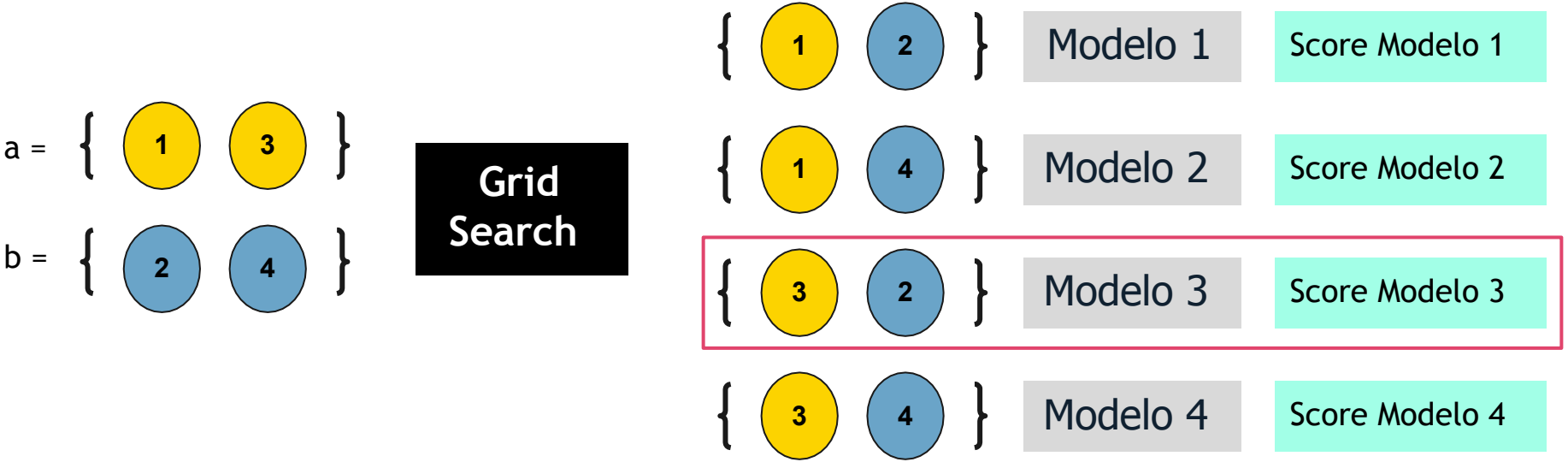


Solución ~~más~~ natural: hacer una búsqueda exhaustiva. Es decir probando con todos los valores de los hiper-parámetros que podamos y eligiendo la mejor combinación. Éste método se llama Grid Search (“búsqueda de cuadrícula”).

Por ejemplo, si tenemos dos hiperparámetros, a y b, que pueden tomar valores a = {1,3} y b = {2,4}



Elegimos el modelo con mejor score (o menor error si corresponde):



Entonces, una vez que sabemos qué métrica queremos optimizar, **Grid Search consiste en:**

1. Elegimos los valores que puede tomar cada hiperparámetro
2. Armamos las combinaciones “todos con todos” → Tenemos nuestra grilla.
3. Recorremos la grilla, entrenamos un modelo para cada combinación y lo evaluamos.
4. Elegimos los hiperparámetros que definen el mejor modelo.

Entonces, una vez que sabemos qué métrica queremos optimizar, **Grid Search consiste en:**

1. Elegimos los valores que puede tomar cada hiperparámetro
2. Armamos las combinaciones “todos con todos” → Tenemos nuestra grilla
3. Recorremos la grilla, entrenamos un modelo para cada combinación y **lo evaluamos.**
4. Elegimos los hiperparámetros que definen el mejor modelo.

¿Cómo evaluamos un modelo?

¿Con Train/Test split? ¿Con Validación Cruzada?

Al estar probando MUCHOS modelos, podría suceder que uno se desempeñe muy bien en el conjunto de Train simplemente por azar.

Por esto ,es muy importante evaluar cada modelo creado por Grid Search con Validación Cruzada en el conjunto de Dev (Train).

¿Cómo evaluamos un modelo?

¿Con Train/Test split? ¿Con Validación Cruzada?

Al estar probando MUCHOS modelos, podría suceder que uno se desempeñe muy bien en el conjunto de Train simplemente por azar.

Por esto ,es muy importante evaluar cada modelo creado por Grid Search con Validación Cruzada en el conjunto de Dev (Train).

¡Grid Search y Validación Cruzada suelen venir juntos!

Entonces, una vez que elegimos los mejores hiperparámetros con Grid Search + Validación Cruzada:

1. Elegimos Entrenamos un modelo con esos hiperparámetros con todo el conjunto de Dev (Train)
2. Evaluamos su performance en el conjunto de Held-Out (Test).

Entonces, una vez que elegimos los mejores hiperparámetros con Grid Search + Validación Cruzada:

1. Elegimos Entrenamos un modelo con esos hiperparámetros con todo el conjunto de Dev (Train)
2. Evaluamos su performance en el conjunto de **Held-Out (Test).**

Como la evaluación de performance de CV puede estar sesgada, es importante evaluar al final en este conjunto para obtener una estimación menos sesgada del desempeño.

Para pensar:
¿por qué puede estar sesgada la evaluación de performance del modelo con CV?

¿Qué desventajas tiene Grid Search?

¿Qué ocurre si tenemos, por ejemplo, cinco hiperparámetros y cinco valores para probar por hiperparámetro? ¿Qué tamaño tiene la grilla?

¿Qué desventajas tiene Grid Search?

¿Qué ocurre si tenemos, por ejemplo, cinco hiperparámetros y cinco valores para probar por hiperparámetro? ¿Qué tamaño tiene la grilla?

¿Y si, además, para cada modelo tenemos que hacer Validación Cruzada?

¿Qué desventajas tiene Grid Search?

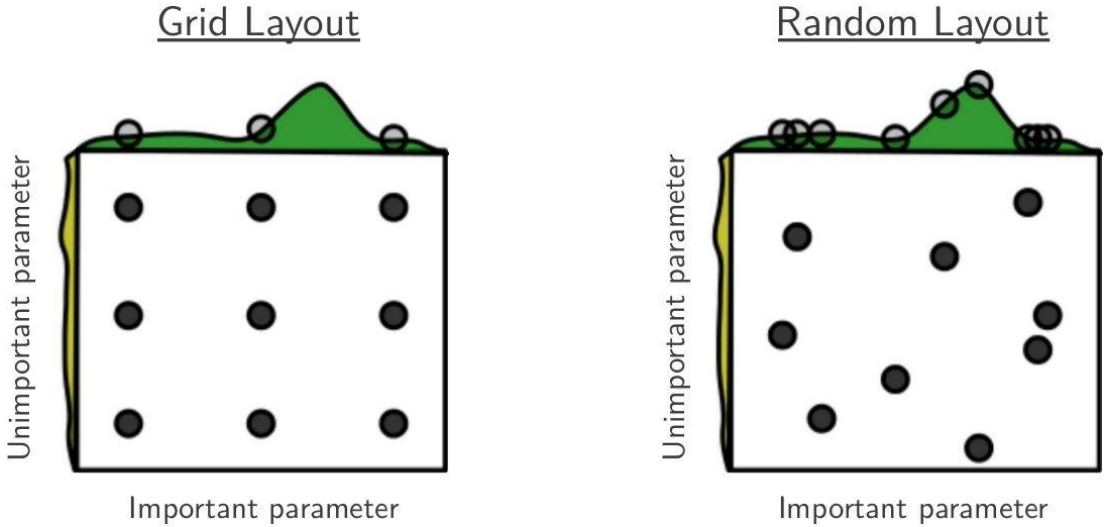
Grid Search + CV puede ser computacionalmente muy demandante.

Y tal vez al "probar" con pocos valores de los hiperparámetros nos estamos perdiendo algunas cosas importantes.

Además, suele haber hiperparámetros mucho más importantes que otros.

¿Y entonces?

Random Search explora opciones y combinaciones al azar, de manera menos "ordenada". En muchas circunstancias, ¡esto es más eficiente, tanto desde el punto de vista de performance del modelo como de desempeño computacional!



Fuente

Random Search:

- En la búsqueda aleatoria, creamos una cuadrícula de hiperparámetros y entrenamos / probamos nuestro modelo en solo una combinación aleatoria de estos hiperparámetros.

- Al realizar tareas de aprendizaje automático, generalmente dividimos nuestro conjunto de datos en conjuntos de entrenamiento y de prueba. Esto se hace para probar nuestro modelo después de haberlo entrenado (de esta manera podemos verificar su desempeño cuando trabajamos con datos invisibles). Cuando usamos la validación cruzada, dividimos nuestro conjunto de entrenamiento en otras N particiones para asegurarnos de que nuestro modelo no sobreajuste nuestros datos.

Fuente

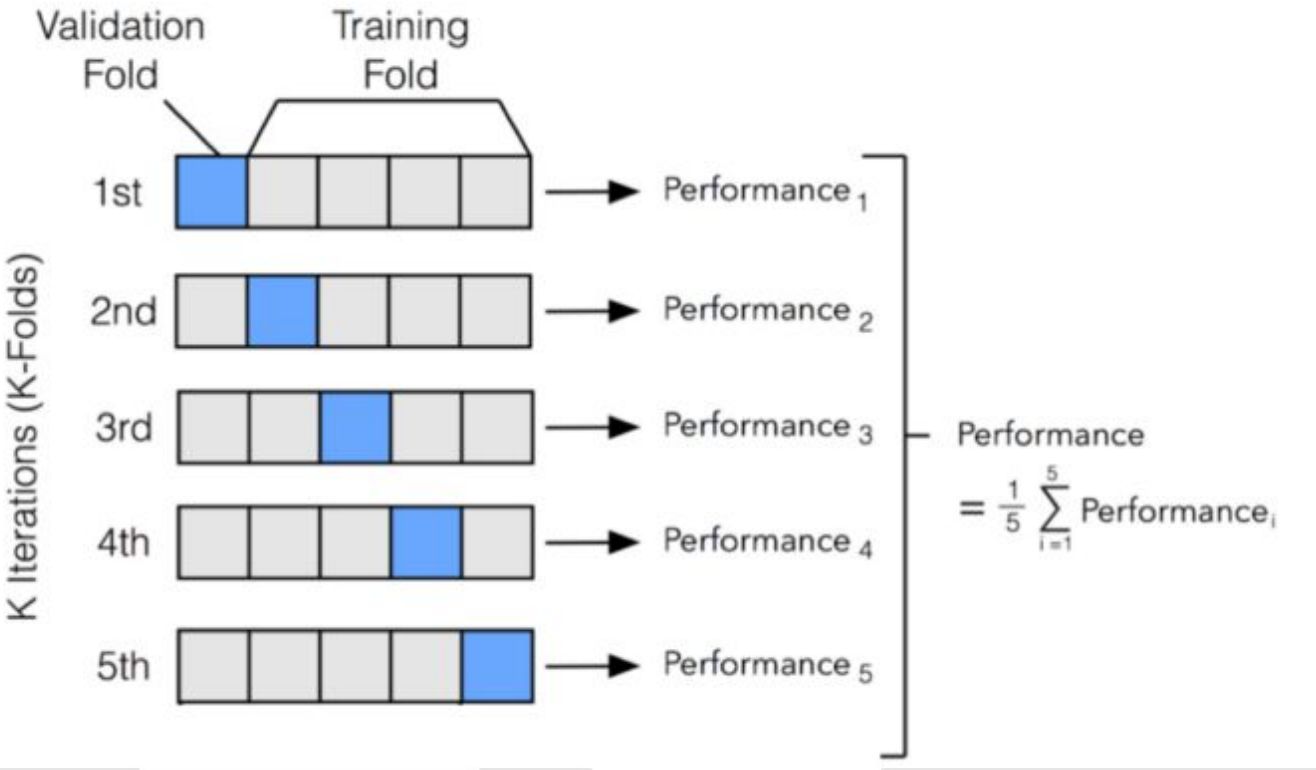
Random Search:

- Uno de los métodos de validación cruzada más utilizados es la validación de K-Fold. En K-Fold, dividimos nuestro conjunto de entrenamiento en N particiones y luego entrenamos iterativamente nuestro modelo usando N-1 particiones y lo probamos con la partición sobrante (en cada iteración cambiamos la partición sobrante).

- Una vez que hemos entrenado N veces el modelo, promediamos los resultados de entrenamiento obtenidos en cada iteración para obtener nuestros resultados de rendimiento de entrenamiento general.

Fuente

Random Search:



Fuente

Random Search:

- El uso de la validación cruzada al implementar la optimización de hiperparámetros puede ser realmente importante. De esta manera, podríamos evitar el uso de algunos hiperparámetros que funcionan muy bien con los datos de entrenamiento pero no tan bien con los datos de prueba.

- Ahora podemos comenzar a implementar la búsqueda aleatoria desafiando primero una cuadrícula de hiperparámetros que se muestrearán aleatoriamente al llamar a ***RandomizedSearchCV ()***. Para este ejemplo, decidí dividir nuestro conjunto de entrenamiento en 4 pliegues (CV = 4) y seleccionar 80 como el número de combinaciones para muestrear (n_iter = 80).

- Usando el atributo scikit-learn best_estimator_attribute, podemos recuperar el conjunto de hiperparámetros que funcionaron mejor durante el entrenamiento para probar nuestro modelo.

Fuente

Random Search:

```
import numpy as np
from sklearn.model_selection import RandomizedSearchCV
from sklearn.model_selection import cross_val_score

random_search = {'criterion': ['entropy', 'gini'],
                 'max_depth': list(np.linspace(10, 1200, 10, dtype = int)) + [None],
                 'max_features': ['auto', 'sqrt','log2', None],
                 'min_samples_leaf': [4, 6, 8, 12],
                 'min_samples_split': [5, 7, 10, 14],
                 'n_estimators': list(np.linspace(151, 1200, 10, dtype = int))}

clf = RandomForestClassifier()
model = RandomizedSearchCV(estimator = clf, param_distributions = random_search, n_iter = 80,
                                cv = 4, verbose= 5, random_state= 101, n_jobs = -1)
model.fit(X_Train,Y_Train)
```

Fuente

Automated Hyperparameter Tuning:

Cuando se utiliza el ajuste de hiperparámetros automatizado, los hiperparámetros del modelo a utilizar se identifican mediante técnicas como: optimización bayesiana, descenso de gradiente y algoritmos evolutivos.

- **Optimización Bayesiana**
    - La optimización bayesiana se puede realizar en Python utilizando la biblioteca **Hyperopt**. La optimización bayesiana usa la probabilidad para encontrar el mínimo de una función. El objetivo final es encontrar el valor de entrada a una función que nos pueda dar el valor de salida más bajo posible.
      Se ha demostrado que la optimización bayesiana es más eficiente que la búsqueda aleatoria, en cuadrícula o manual. La optimización bayesiana puede, por lo tanto, conducir a un mejor rendimiento en la fase de prueba y reducir el tiempo de optimización.

Automated Hyperparameter Tuning:

- En Hyperopt, la Optimización Bayesiana se puede implementar dando 3 tres parámetros principales a la función fmin ().
  - Función objetivo = define la función de pérdida para minimizar.
  - Espacio de dominio = define el rango de valores de entrada para probar (en Optimización Bayesiana este espacio crea una distribución de probabilidad para cada uno de los hiperparámetros usados).
  - Algoritmo de optimización = define el algoritmo de búsqueda que se utilizará para seleccionar los mejores valores de entrada para utilizar en cada nueva iteración.
  - Además, también se puede definir en fmin () el número máximo de evaluaciones a realizar.

- La optimización bayesiana puede reducir el número de iteraciones de búsqueda eligiendo los valores de entrada teniendo en cuenta los resultados pasados. De esta forma, podemos concentrar nuestra búsqueda desde el principio en valores que estén más cerca de nuestro resultado deseado.

- Ahora podemos ejecutar nuestro Optimizador Bayesiano usando la función fmin (). Primero se crea un objeto Trials () para poder visualizar más tarde lo que estaba sucediendo mientras se ejecutaba la función fmin () (por ejemplo, cómo cambiaba la función de pérdida y cómo cambiaban los hiperparámetros usados).

```
from hyperopt import hp, fmin, tpe, STATUS_OK, Trials

space = {'criterion': hp.choice('criterion', ['entropy', 'gini']),
        'max_depth': hp.quniform('max_depth', 10, 1200, 10),
        'max_features': hp.choice('max_features', ['auto', 'sqrt','log2', None]),
        'min_samples_leaf': hp.uniform ('min_samples_leaf', 0, 0.5),
        'min_samples_split' : hp.uniform ('min_samples_split', 0, 1),
        'n_estimators' : hp.choice('n_estimators', [10, 50, 300, 750, 1200])
    }

def objective(space):
    model = RandomForestClassifier(criterion = space['criterion'],
                                   max_depth = space['max_depth'],
                                max_features = space['max_features'],
                                min_samples_leaf = space['min_samples_leaf'],
                                min_samples_split = space['min_samples_split'],
                                n_estimators = space['n_estimators'],
                                )

    accuracy = cross_val_score(model, X_Train, Y_Train, cv = 4).mean()

    return {'loss': -accuracy, 'status': STATUS_OK }

trials = Trials()
best = fmin(fn= objective,
            space= space,
            algo= tpe.suggest,
            max_evals = 80,
            trials= trials)
best
```

- Ahora podemos recuperar el conjunto de mejores parámetros identificados y probar nuestro modelo utilizando el mejor diccionario creado durante el entrenamiento. Algunos de los parámetros se han almacenado en el diccionario numéricamente usando índices, por lo tanto, primero necesitamos convertirlos nuevamente como cadenas antes de ingresarlos en nuestro Random Forest (por ejemplo).

```
5   trainedforest = RandomForestClassifier(criterion = crit[best['criterion']],
6                                          max_depth = best['max_depth'],
7                                          max_features = feat[best['max_features']],
8                                          min_samples_leaf = best['min_samples_leaf'],
9                                          min_samples_split = best['min_samples_split'],
10                                         n_estimators = est[best['n_estimators']]
11                                        ).fit(X_Train,Y_Train)
12  predictionforest = trainedforest.predict(X_Test)
13  print(confusion_matrix(Y_Test,predictionforest))
14  print(classification_report(Y_Test,predictionforest))
15  acc5 = accuracy_score(Y_Test,predictionforest)
```

Automated Hyperparameter Tuning:

- **Algoritmos Genéticos**
  - Los algoritmos genéticos intentan aplicar mecanismos de selección natural a contextos de aprendizaje automático. Están inspirados en el proceso darwiniano de selección natural y, por lo tanto, también se les suele llamar algoritmos evolutivos.
  - Imaginemos que creamos una población de N modelos de aprendizaje automático con algunos hiperparámetros predefinidos. Luego podemos calcular la precisión de cada modelo y decidir mantener solo la mitad de los modelos (los que funcionan mejor).
  - Ahora podemos generar algunas crías que tengan hiperparámetros similares a los de los mejores modelos para obtener nuevamente una población de N modelos. En este punto, podemos volver a calcular la precisión de cada modelo y repetir el ciclo para un número definido de generaciones. De esta manera, solo los mejores modelos sobrevivirán al final del proceso.

Automated Hyperparameter Tuning:

- **Algoritmos Genéticos**
  - Para implementar algoritmos genéticos en Python, podemos usar la biblioteca TPOT Auto Machine Learning. TPOT se basa en la biblioteca scikit-learn y se puede utilizar para tareas de regresión o clasificación.
  - *tpot_classifier.score(X_Test, Y_Test)*

```
from tpot import TPOTClassifier

parameters = {'criterion': ['entropy', 'gini'],
              'max_depth': list(np.linspace(10, 1200, 10, dtype = int)) + [None],
              'max_features': ['auto', 'sqrt','log2', None],
              'min_samples_leaf': [4, 12],
              'min_samples_split': [5, 10],
              'n_estimators': list(np.linspace(151, 1200, 10, dtype = int))}

tpot_classifier = TPOTClassifier(generations= 5, population_size= 24, offspring_size= 12,
                                 verbosity= 2, early_stop= 12,
                                 config_dict=
                                 {'sklearn.ensemble.RandomForestClassifier': parameters},
                                 cv = 4, scoring = 'accuracy')
tpot_classifier.fit(X_Train,Y_Train)
```

Automated Hyperparameter Tuning:

- **Artificial Neural Networks (ANNs) Tuning**

  - Con el contenedor KerasClassifier, es posible aplicar la búsqueda en cuadrícula y la búsqueda aleatoria para modelos de aprendizaje profundo de la misma manera que se hizo al usar modelos de aprendizaje automático de scikit-learn.

  - En el siguiente ejemplo, intentaremos optimizar algunos de nuestros parámetros ANN como: cuántas neuronas usar en cada capa y qué función de activación y optimizador usar. Más ejemplos de optimización de hiperparámetros de aprendizaje profundo.

- Ejemplo:

```
from keras.models import Sequential

from keras.layers import Dense, Dropout

from keras.wrappers.scikit_learn import KerasClassifier
```

Automated Hyperparameter Tuning:

- **Artificial Neural Networks (ANNs) Tuning**
  - Ejemplo:

```
def DL_Model(activation= 'linear', neurons= 5, optimizer='Adam'):
   model = Sequential()
   model.add(Dense(neurons, input_dim= 4, activation= activation))
   model.add(Dense(neurons, activation= activation))
   model.add(Dropout(0.3))
   model.add(Dense(1, activation='sigmoid'))
   model.compile(loss='binary_crossentropy', optimizer= optimizer,
metrics=['accuracy'])
   return model
# Definying grid parameters
activation = ['softmax', 'relu', 'tanh', 'sigmoid', 'linear']
neurons = [5, 10, 15, 25, 35, 50]
optimizer = ['SGD', 'Adam', 'Adamax']
param_grid = dict(activation = activation, neurons = neurons, optimizer =
optimizer)
```

Automated Hyperparameter Tuning:

- **Artificial Neural Networks (ANNs) Tuning**
  - Ejemplo:

```
def DL_Model(activation= 'linear', neurons= 5, optimizer='Adam'):
   model = Sequential()
   model.add(Dense(neurons, input_dim= 4, activation= activation))
   model.add(Dense(neurons, activation= activation))
   model.add(Dropout(0.3))
   model.add(Dense(1, activation='sigmoid'))
   model.compile(loss='binary_crossentropy', optimizer= optimizer,
metrics=['accuracy'])
   return model
# Definying grid parameters
activation = ['softmax', 'relu', 'tanh', 'sigmoid', 'linear']
neurons = [5, 10, 15, 25, 35, 50]
optimizer = ['SGD', 'Adam', 'Adamax']
param_grid = dict(activation = activation, neurons = neurons, optimizer =
optimizer)
```

Automated Hyperparameter Tuning:

- **Artificial Neural Networks (ANNs) Tuning**
  - Ejemplo:

```
clf = KerasClassifier(build_fn= DL_Model, epochs= 80, batch_size=40, verbose= 0)


model = GridSearchCV(estimator= clf, param_grid=param_grid, n_jobs=-1)
model.fit(X_Train,Y_Train)


print("Max Accuracy Registred: {} using {}".format(round(model.best_score_,3),
                                                   model.best_params_))
```

Resumen · Para optimizar hiperparámetros necesitamos:

- Una **métrica** (exactitud, precisión, RMSE, ROC AUC, etc.)
- Un **modelo** (regresor o clasificador)
- Un **espacio** de (hiper)parámetros. Depende del tipo de modelo que estemos usando.
- Un **método** para buscar o muestrear los candidatos
  a. **Grid Search**: Plantea opciones y explora todas las combinaciones
  b. **Random Search**: explora opciones y combinaciones al azar.
  c. **Optimización Bayesiana**
  d. **Algoritmos Genéticos**

Guia para ordenar ideas.

1) Separar los datos de Held-Out de los de procesamiento.
2) Armar el Grid-Search, definiendo modelos y rangos de sus hiperparámetros para:
    a) Preprocesamiento de features
    b) Algoritmo de ML
3) Ejecutar el Grid-Search, evaluando con K-fold cross-validation la performance de cada modelo construido con cada una de todas las combinaciones posibles de las opciones planteadas en (2).
4) Elegir el modelo de mejor performance, y entrenarlo con todos los datos de entrenamiento.
5) Estimar la performance del modelo evaluando sobre el Held-out.
6) El modelo sale a producción.

En
Scikit-Learn

sklearn.model_selection.GridSearchCV

class sklearn.model_selection.**GridSearchCV**(*estimator*, *param_grid*, *scoring=None*, *n_jobs=None*, *iid='deprecated'*, *refit=True*, *cv=None*, *verbose=0*, *pre_dispatch='2*n_jobs'*, *error_score=nan*, *return_train_score=False*) [source]

sklearn.model_selection.RandomizedSearchCV

class sklearn.model_selection.**RandomizedSearchCV**(*estimator*, *param_distributions*, *n_iter=10*, *scoring=None*, *n_jobs=None*, *iid='deprecated'*, *refit=True*, *cv=None*, *verbose=0*, *pre_dispatch='2*n_jobs'*, *random_state=None*, *error_score=nan*, *return_train_score=False*) [source]

Buenas prácticas de un data scientist

https://www.youtube.com/watch?v=AXDByU3D1hA&feature=emb_logo

Recursos

Optimización de Hiperparámetros

- Como siempre, la guía de Scikit-Learn es una muy buena referencia.

- **Capítulo 5, "Machine Learning: Hyperparameters and Model Validation", de Python Data Science Handbook.** La última sección tiene un ejemplo con Grid Search.

- **Muy recomendable:** Artículo con código y ejemplos de optimización de Hiperparámetros, mostrando algunas técnicas más de las que vimos en la clase. Además, hace un breve repaso de algunos conceptos que vimos en las clases. ¡Notar qué dataset usan de ejemplo!