# Otimização de Hiperparâmetros

Que bom que vocês chegaram até aqui :) Espero que a jornada de evolução do curso esteja sendo positiva!

No tópico de hoje, falaremos sobre otimização de hiperparâmetros! Mas, antes, que tal darmos uma relembrada?

. . .

## O que são hiperparâmetros?

Para a aula de hoje, vamos pegar uma regressão ElasticNet como exemplo:

```
from sklearn.linear_model import ElasticNet
```

E vamos, também, utilizar um dataset de exemplo:

```
from sklearn.datasets import fetch_california_housing
import numpy as np


dataset = fetch_california_housing()
dataset_features = dataset["data"]
target = dataset["target"]
feature_names = dataset["feature_names"]
target_name = dataset["target_names"]

dataset = pd.DataFrame(
    np.hstack((dataset_features, target.reshape(-1, 1))),
    columns=np.hstack((feature_names, target_name)),
)
dataset.head()
```

| | MedInc | HouseAge | AveRooms | AveBedrms | Population | AveOccup | Latitude | Longitude | MedHouseVal |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 8.3252 | 41.0 | 6.984127 | 1.023810 | 322.0 | 2.555556 | 37.88 | -122.23 | 4.526 |
| 1 | 8.3014 | 21.0 | 6.238137 | 0.971880 | 2401.0 | 2.109842 | 37.86 | -122.22 | 3.585 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 7.2574 | 52.0 | 8.288136 | 1.073446 | 496.0 | 2.802260 | 37.85 | -122.24 | 3.521 |
| 3 | 5.6431 | 52.0 | 5.817352 | 1.073059 | 558.0 | 2.547945 | 37.85 | -122.25 | 3.413 |
| 4 | 3.8462 | 52.0 | 6.281853 | 1.081081 | 565.0 | 2.181467 | 37.85 | -122.25 | 3.422 |

Primeiras 5 linhas do california housing dataset

Agora, vamos separar nossos dados em treino e teste e fitar o modelo

```
dataset = fetch_california_housing()
dataset_features = dataset["data"]
target = dataset["target"]
feature_names = dataset["feature_names"]
target_name = dataset["target_names"]

dataset = pd.DataFrame(
    np.hstack((dataset_features, target.reshape(-1, 1))),
    columns=np.hstack((feature_names, target_name)),
)
dataset.head()
```

Agora, vamos separar nossos dados em treino e teste e fitar o modelo

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error

sc_features = StandardScaler()
sc_target = StandardScaler()
X_train, X_test, y_train, y_test = train_test_split(
    dataset[feature_names], dataset[target_name], random_state=42
)
X_train = sc_features.fit_transform(X_train)
X_test = sc_features.transform(X_test)
y_train = sc_target.fit_transform(y_train.values.reshape(-1, 1))
y_test = sc_target.transform(y_test.values.reshape(-1, 1))

def execute_train_and_compute_metric(
    params={},
    X_train=X_train,
    X_test=X_test,
    y_train=y_train,
    y_test=y_test,
    metric_func=mean_squared_error,
):
    model = ElasticNet(random_state=42, **params)
    model.fit(X_train, y_train)
    metric_result = metric_func(
        sc_target.inverse_transform(y_test),
        sc_target.inverse_transform(model.predict(X_test)),
```

```
    )
    return metric_result

metric_result = execute_train_and_compute_metric()
metric_result
```

Obtemos o valor de 1.1165348335795664 !

Perceba que *execute_train_and_compute_metric* é uma função e que será utilizada em vários momentos aqui

Agora, como vimos em aulas passadas, podemos acessar os parâmetros **aprendidos** pelo modelo:

```
model_trainded = ElasticNet(random_state=42)
model_trainded.fit(X_train, y_train)
coefs = pd.DataFrame(
    sc_features.inverse_transform(model_trainded.coef_),
    columns=["Coefficients"],
    index=feature_names,
)
intercept = pd.DataFrame(
    model_trainded.intercept_, columns=["Coefficients"], index=
["Intercept"]
)

pd.concat([coefs, intercept])
```

]:

| | Coefficients |
|---|---|
| **MedInc** | 4.117799e+00 |
| **HouseAge** | 2.859599e+01 |
| **AveRooms** | 5.435598e+00 |
| **AveBedrms** | 1.096881e+00 |
| **Population** | 1.427497e+03 |
| **AveOccup** | 3.106660e+00 |
| **Latitude** | 3.564672e+01 |

| | |
|---|---|
| **Longitude** | -1.195837e+02 |
| **Intercept** | 5.835789e-17 |

Como podemos ver, esses coeficientes foram **aprendidos** pelo modelo. Chamamos isso de **parâmetros** do modelo. Contudo se olharmos as variáveis da classe *ElasticNet*, temos o seguinte:

> *- alpha : float, default=1.0*
>
> *Constant that multiplies the penalty terms. Defaults to 1.0.*
> *See the notes for the exact mathematical meaning of this*
> *parameter. ``alpha = 0`` is equivalent to an ordinary least square,*
> *solved by the :class:`LinearRegression` object. For numerical*
> *reasons, using ``alpha = 0`` with the ``Lasso`` object is not advised.*
> *Given this, you should use the :class:`LinearRegression` object.*
>
> *- l1_ratio : float, default=0.5*
>
> *The ElasticNet mixing parameter, with ``0 <= l1_ratio <= 1``. For*
> *``l1_ratio = 0`` the penalty is an L2 penalty. ``For l1_ratio = 1`` it*
> *is an L1 penalty. For ``0 < l1_ratio < 1``, the penalty is a*
> *combination of L1 and L2.*

Como podemos ver, `alpha` e `l1_ratio` são **parâmetros da classe** e recebem os valores `1.0` e `0.5` como *default*. Logo, o que acontece se alterarmos esse valores?

```
metric_result = execute_train_and_compute_metric(
    params={"alpha": 10.0, "l1_ratio": 0.3}
)
metric_result
```

Obteremos o valor de 1.3232694135654075. Em outras palavras, podemos ver que ao alterar os valores, o nosso resultado muda! Acontece que `alpha` e `l1_ratio` são

*parâmetros* que **não** são aprendidos pelo modelo, mas que influenciam o resultado do treinamento. Damos a eles o nome de **hiperparâmetros**.

Logo, a pergunta a pergunta a ser feita seria a seguinte:

> Ora, então como que eu encontro os **melhores** hiperparâmetros para o problema que eu estou enfrentando?

Felizmente, existem métodos que nos ajudam nisso :)

## GridSearch

A primeira técnica é meio direta:

```
- Como em muitos casos o número de parâmetros tem um domínio
contínuo, podemos determinar *um subconjunto* para testarmos

- Iteramos sobre todo esse subconjunto através de uma validação
cruzada (para evitar overfitting) e escolhemos o melhor com base na
métrica resultante
```

Esse método é conhecido como GridSearch

```
%%time
from sklearn.model_selection import GridSearchCV
tuned_parameters = [{"alpha": [0.3, 1.0, 5.0, 10.0], "l1_ratio":
[0.0, 0.3, 0.5, 1.0]}]

score = "r2"
print(f"# Tuning hyper-parameters for {score}" )
print()

clf = GridSearchCV(ElasticNet(random_state=42), tuned_parameters,
scoring=score)
clf.fit(X_train, y_train)

print("Best parameters set found on development set:")
print(clf.best_params_)
print()
print("Grid scores on development set:")
print()
means = clf.cv_results_["mean_test_score"]
stds = clf.cv_results_["std_test_score"]
for mean, std, params in zip(means, stds,
clf.cv_results_["params"]):
    print("%0.3f (+/-%0.03f) for %r" % (mean, std * 2, params))
print()
```

```
# Tuning hyper-parameters for r2

Best parameters set found on development set:
{'alpha': 0.3, 'l1_ratio': 0.0}

Grid scores on development set:

0.515 (+/-0.008) for {'alpha': 0.3, 'l1_ratio': 0.0}
0.457 (+/-0.007) for {'alpha': 0.3, 'l1_ratio': 0.3}
0.428 (+/-0.006) for {'alpha': 0.3, 'l1_ratio': 0.5}
0.384 (+/-0.005) for {'alpha': 0.3, 'l1_ratio': 1.0}
0.394 (+/-0.006) for {'alpha': 1.0, 'l1_ratio': 0.0}
0.262 (+/-0.003) for {'alpha': 1.0, 'l1_ratio': 0.3}
0.157 (+/-0.005) for {'alpha': 1.0, 'l1_ratio': 0.5}
-0.001 (+/-0.001) for {'alpha': 1.0, 'l1_ratio': 1.0}
0.160 (+/-0.003) for {'alpha': 5.0, 'l1_ratio': 0.0}
-0.001 (+/-0.001) for {'alpha': 5.0, 'l1_ratio': 0.3}
-0.001 (+/-0.001) for {'alpha': 5.0, 'l1_ratio': 0.5}
-0.001 (+/-0.001) for {'alpha': 5.0, 'l1_ratio': 1.0}
0.091 (+/-0.001) for {'alpha': 10.0, 'l1_ratio': 0.0}
-0.001 (+/-0.001) for {'alpha': 10.0, 'l1_ratio': 0.3}
-0.001 (+/-0.001) for {'alpha': 10.0, 'l1_ratio': 0.5}
-0.001 (+/-0.001) for {'alpha': 10.0, 'l1_ratio': 1.0}

CPU times: user 6.88 s, sys: 52.4 ms, total: 6.94 s
Wall time: 1.76 s
```

---

No caso, vemos que a melhor combinação aqui foram `alpha` =0.3 e `l1_ratio` =0.0

```
metric_result = execute_train_and_compute_metric(
   params={"alpha": 0.3, "l1_ratio": 0.0}
)
metric_result
```

Obtemos o resultado de 0.6429909650973414 :)

Lembrando que nossa métrica de referência foi `MSE` , conseguimos um modelo bem superior! Contudo, já podemos perceber alguns problemas:

- Formar uma lista de valores pode ser meio complicada, principalmente quando os valores podem variar de $-\infty$ a $\infty$
- A iteração pode ser custosa e levar algum tempo. No caso da ElasticNet só temos dois hiperparâmetros, mas pense no caso da RandomForest que tem os seguintes hiperparâmetros:

```
n_estimators=,
criterion=',
max_depth=,
min_samples_split=,
min_samples_leaf=,
min_weight_fraction_leaf=,
max_features=,
max_leaf_nodes=,
min_impurity_decrease=,
min_impurity_split=,
bootstrap=
```

Será que conseguimos fazer melhor?

## RandomSearch

Uma outra forma de fazer a busca de hiperparâmetros seria definir uma distribuição e deixar a busca fazer amostras com base nessas distribuições.

```
%%time
from sklearn.model_selection import RandomizedSearchCV
tuned_parameters = [
    {"alpha": np.logspace(-2, 2, num=100), "l1_ratio":
np.logspace(-3, 0, num=100)}
]

score = "r2"
print(f"# Tuning hyper-parameters for {score}")
print()

clf = RandomizedSearchCV(
    ElasticNet(random_state=42), tuned_parameters, scoring=score,
random_state=42
)
clf.fit(X_train, y_train)

print("Best parameters set found on development set:")
print(clf.best_params_)
print()
print("Grid scores on development set:")
```

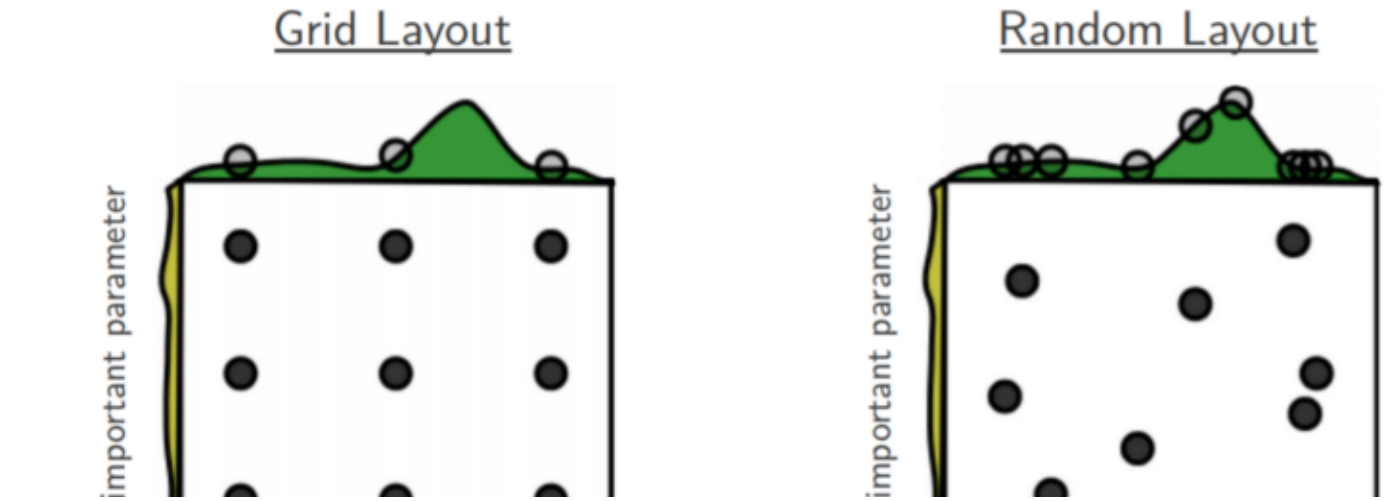
```
print()
for mean, std, params in zip(means, stds,
clf.cv_results_["params"]):
    print("%0.3f (+/-%0.03f) for %r" % (mean, std * 2, params))
print()
```

```
# Tuning hyper-parameters for r2

Best parameters set found on development set:
{'l1_ratio': 0.1, 'alpha': 0.014508287784959394}

Grid scores on development set:

0.515 (+/-0.008) for {'l1_ratio': 0.13219411484660287, 'alpha': 8.111308307896872}
0.457 (+/-0.007) for {'l1_ratio': 0.06579332246575682, 'alpha': 0.02104904144512021}
0.428 (+/-0.006) for {'l1_ratio': 0.5336699231206312, 'alpha': 1.384886371393873}
0.384 (+/-0.005) for {'l1_ratio': 0.572236765935022, 'alpha': 1.149756995397737}
0.394 (+/-0.006) for {'l1_ratio': 0.010722672220103232, 'alpha': 2.0092330025650478}
0.262 (+/-0.003) for {'l1_ratio': 0.093260334688322, 'alpha': 3.1992671377973845}
0.157 (+/-0.005) for {'l1_ratio': 0.1, 'alpha': 0.014508287784959394}
-0.001 (+/-0.001) for {'l1_ratio': 0.006135907273413176, 'alpha': 0.5994842503189411}
0.160 (+/-0.003) for {'l1_ratio': 0.23101297000831603, 'alpha': 1.6681005372000592}
-0.001 (+/-0.001) for {'l1_ratio': 0.004641588833612782, 'alpha': 22.570197196339215}

CPU times: user 897 ms, sys: 22.9 ms, total: 920 ms
Wall time: 241 ms
```

```
metric_result = execute_train_and_compute_metric(
   params={"alpha": 0.1, "l1_ratio": 0.014508287784959394}
)
metric_result
```

Conseguimos um resultado de 0.5742851327707377 ! Logo, um valor melhor do que o que o `GridSearch` ! Interessante, né? Mas qual o motivo disso?

Acontece que na prática, o `RandomSearch` consegue obter resultados do que o `GridSearch` por motivos que ficam mais claros pela figura abaixo




Fonte: https://blog.usejournal.com/a-comparison-of-grid-search-and-randomized-search-using-scikit-learn-29823179bc85

Contudo, podemos perceber que nos dois casos, nós, cientistas de dados que introduzimos os parâmetros e os valores que julgamos mais relevantes. Será que conseguimos fazer algo mais informativo?

. . .

## Busca Bayesiana

Quando falamos de busca **Bayesiana** de parâmetros, é como se tratassemos essa busca como um problema de regressão e utilizamos inferência bayesiana para fazer "chutes" sobre qual seria o conjunto de parâmetros ideais.

Entender Inferência Bayesiana é um tópico avançado e não cabe aqui. Quem tiver interesse, recomendo esse paper, aqui ou aqui.

Para exemplificar, vamos supor que queremos estimar o valor do `l1_ratio`. A medida que vamos iterando no nosso processo de busca, a Busca Bayesiana faz chutes direcionados, tentando encontrar um equilíbrio entre *Explorar* (**Exploration**), ou seja, explorar valores novos e nunca vistos antes e *Usufruir* (**Exploitation**), que seria utilizar os valores que resultem no **melhor modelo possível**

Na prática, acontece algo como exemplificado pela imagem abaixo. Além disso, para quem quiser entender mais, usaremos essa lib, que possui uma ótima documentação !

```
- Uma função para maximizar. Nesse caso, queremos treinar o modelo e
avaliar o resultado. No caso, iremos usar o r quadrado.

- Nossos parâmetros de acordo com um range
```

Aplicando em código:

```
from sklearn.metrics import r2_score
from bayes_opt import BayesianOptimization

def elastic_eval(alpha, l1_ratio):
    params = {"alpha": alpha, "l1_ratio": l1_ratio}
    metric_result = execute_train_and_compute_metric(
        params=params, metric_func=r2_score
    )
    return metric_result


# Bounded region of parameter space
pbounds = {
    "alpha": (0.01, 2),
    "l1_ratio": (0.001, 1),
}

optimizer = BayesianOptimization(f=elastic_eval, pbounds=pbounds,
random_state=42)
```

Uma vez definido o otimizador que iremos maximizar, temos dois parâmetros de interesse para passar:

- *n_iter*: Quantas iterações de otimização bayesiana queremos fazer. Quanto mais melhor, apesar de durar mais.
- *init_points*: Quantos passos de exploração aleatória queremos fazer. Aqui, quanto mais, maior a chance de encontrarmos os valores ótimos.

```
optimizer.maximize(
    init_points=25,
    n_iter=5,
)
```

| iter | target | alpha | l1_ratio |
|---|---|---|---|
| 1 | -3.878e-0 | 0.7553 | 0.9508 |
| 2 | -3.878e-0 | 1.467 | 0.5991 |
| 3 | 0.4742 | 0.3205 | 0.1568 |
| 4 | 0.4771 | 0.1256 | 0.8663 |
| 5 | -3.878e-0 | 1.206 | 0.7084 |
| 6 | 0.5269 | 0.05096 | 0.9699 |
| 7 | 0.1765 | 1.667 | 0.2131 |

| | | | | |
|---|---|---|---|---|
| 8 | 0.4521 | 0.3718 | 0.1842 |
| 9 | 0.3072 | 0.6154 | 0.5252 |
| 10 | 0.296 | 0.8696 | 0.2919 |
| 11 | 0.2807 | 1.228 | 0.1404 |
| 12 | 0.3522 | 0.5914 | 0.367 |
| 13 | -3.878e-0 | 0.9176 | 0.7854 |
| 14 | 0.388 | 0.4074 | 0.5147 |
| 15 | 0.3364 | 1.189 | 0.0474 |
| 16 | 0.2697 | 1.219 | 0.1714 |
| 17 | 0.4654 | 0.1395 | 0.9489 |
| 18 | -3.878e-0 | 1.932 | 0.8086 |
| 19 | 0.4142 | 0.6162 | 0.09857 |
| 20 | 0.06283 | 1.372 | 0.4407 |
| 21 | 0.447 | 0.2529 | 0.4957 |
| 22 | 0.4985 | 0.07843 | 0.9094 |
| 23 | 0.3125 | 0.525 | 0.6629 |
| 24 | 0.3024 | 0.6303 | 0.5205 |
| 25 | 0.2848 | 1.098 | 0.1857 |
| 26 | 0.5928 | 0.01 | 0.001 |
| 27 | 0.5937 | 0.01 | 0.2288 |
| 28 | 0.5944 | 0.01 | 0.483 |
| 29 | 0.5941 | 0.01 | 0.3647 |
| 30 | 0.5932 | 0.01 | 0.0985 |

resultados do optimzer.maximize

```
optimizer.max
```

```
{'target': 0.5943533266706624,
 'params': {'alpha': 0.01, 'l1_ratio': 0.4829975346695592}}
```

Como nosso `target` era o R2, podemos calcular para o `MSE` para podermos comparar com o `RandomSearch` :)

```
metric_result = execute_train_and_compute_metric(
    optimizer.max["params"]
)
metric_result
```

Obtendo um resultado de 0.5367590223471562. Logo, **melhor que o** `RandomSearch` , que era de 0.5742851327707377

## Busca Informada

Além de Processos Gaussianos, existem outros algoritmos que fazem uso de técnicas bayesianas ou fazem usos de informações para encontrar o hiperparâmetro ótimo. Para isso, utilizaremos a lib `hyperopt` . Novamente, não entrarei nos detalhes algoritmicos sobre como ela funciona. Quem quiser, pode ver mais detalhes aqui; aqui; e para os algoritmos aqui para o TPE, aqui e aqui para Simmulated Annealing e aqui para uma comparação.

Diferente do que vimos anteriormente, aqui queremos uma função para **minimizar**. Então, podemos utilizar o MSE

Além disso, também precisamos escolher como nossa distribuição será montada, semelhante a como vimos nos casos anteriores. Contudo, aqui, precisamos usar a API definida pela própria biblioteca. Aqui você encontra um ótimo detalhe de como cada uma das distribuições são definidas.

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

# inspired code at references
space = {
    "alpha": hp.normal("alpha", 0, 0.01),
    "l1_ratio": hp.loguniform("l1_ratio", -7, 0),
}
# trials will contain logging information
trials = Trials()
n_iter = 100

best = fmin(
    execute_train_and_compute_metric,
    space,
    algo=tpe.suggest,
    max_evals=n_iter,
    rstate=np.random.RandomState(42),  # fixing random state for the
reproducibility
)

metric_result = execute_train_and_compute_metric(
    best
)
metric_result
```

Aqui, obtemos um resultado de 0.535658473709299 e, caso, quiséssemos testar outro algoritmo, é só alterar o seguinte trecho:

```
best = fmin(
    execute_train_and_compute_metric,
    space,
    algo=anneal.suggest,
    max_evals=n_iter,
    rstate=np.random.RandomState(42),  # fixing random state for the
reproducibility
)

metric_result = execute_train_and_compute_metric(
    best
)
metric_result
```

Obtendo o resultado de 0.5356638444618915.

Podemos ver, então, que tanto os resultados obtidos pelo `hyperopt` quanto pelo `bayes_opt` foram **extremamente similares**. Isso se deve porquê os três casos que analisamos utlizam alguma informação sobre a rodada anterior para fazer um próximo chute! Legal, né? :)

## Conclusão

Que bom que você chegou até aqui! :)

O processo de escolha de hiperparâmetros é algo extremamente poderoso e pode nos ajudar *muito* quando queremos otimizar alguma métrica de interesse!

## Referências e para saber mais:


- https://scikit-learn.org/stable/auto_examples/model_selection/plot_grid_search_digits.html#sphx-glr-auto-examples-model-selection-plot-grid-search-digits-py
- https://scikit-learn.org/stable/modules/grid_search.html#tuning-the-hyper-parameters-of-an-estimator
- https://blog.usejournal.com/a-comparison-of-grid-search-and-randomized-search-using-scikit-learn-29823179bc85 - https://www.kaggle.com/clair14/tutorial-bayesian-optimization

- https://nbviewer.jupyter.org/github/Yorko/mlcourse.ai/blob/master/jupyter_english/tutorials/hyperparameters_tunning_ilya_larchenko.ipynb
- https://www.kaggle.com/fanvacoolt/tutorial-on-hyperopt