



HPO 2부) Bayesian Optimization + alpha

0. Hyper-Parameter Tuning Techniques

0.1 Black Box Optimization

0.2 Multi-fidelity optimization

1. Grid Search vs Random Search

1.1 Grid Search

1.2 Random Search

1.3 Grid Search와 Random Search의 문제점

2. SMBO

2.1 Surrogate Model

2.2 Acquisition function

2.2.1 Probability of Improvement(PI)

2.2.2 Expected Improvement(EI)

2.3 Bayesian Optimization

2.4 Surrogate Model(1) - Gaussian Process

2.5 Surrogate Model(2) - Tree-structured Parzen Estimator Approach (TPE)

2.6 Optuna

코드 (TPE vs Skopt)

참고자료

0. Hyper-Parameter Tuning Techniques

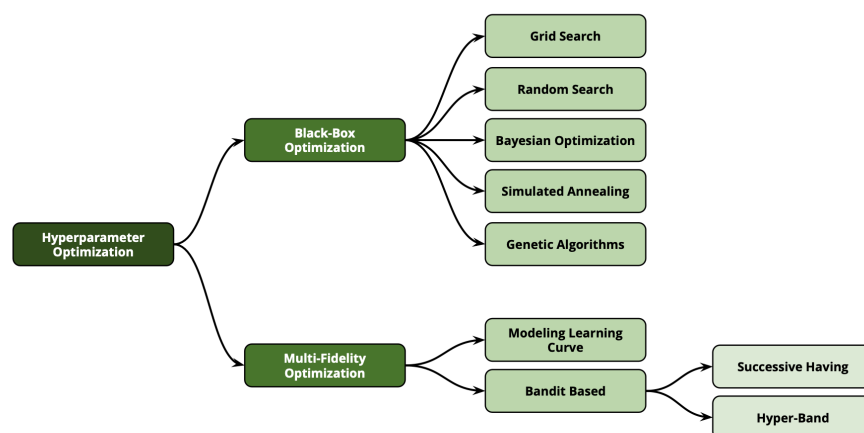


Figure 5: A Taxonomy for the Hyper-parameter Optimization Techniques.

기본적으로, 자동화된 파라미터 튜닝 방법(automated hyper-parameter tuning technique)은 black-box optimization과 Multi-Fidelity optimization으로 분류될 수 있다.

0.1 Black Box Optimization

Black Box optimization이란, 우리가 최적화 하고자 하는 함수의 전체적인 형태는 알 수 없지만, 입력에 대한 출력은 확인 할 수 있는 함수 (evaluation만 가능한 black box)에 대해 우리가 원하는 최적의 input값을 찾는 방법을 의미한다. 기존의 딥러닝 기법들은 경사도 기반 방법 (gradient descent method)들을 사용하여 loss 함수를 최적화하는 것을 통해 학습을 해왔지만, black box optimization은 이와 다르게 함수의 gradient ($\nabla f(x)$)나 hessian 행렬 ($\nabla^2 f(x)$)를 직접 구하지 않고도 최적화를 진행한다. 예를 들어 $P(\lambda; D)$ subject to $\lambda \in \Lambda$ 는 하이퍼 파라미터 튜닝 문제로 블랙박스 최적화 문제이다. 즉, 하이퍼 파라미터 람다가 주어졌을 때 성능은 모델을 학습하고 평가하기 전까지는 알 수 없다.

Black-Box optimization의 종류로는 grid search, random search, bayesian optimization, simulated annealing과 genetic algorithms가 있다.

0.2 Multi-fidelity optimization

Multi-Fidelity Optimization이란, 다수의 cheap low-fidelity evaluation과 소수의 expensive high-fidelity evaluation을 활용해 evaluation cost를 낮추는 방법이다. 이러한 최적화 방법은 큰 데이터셋이 주어져 하나의 하이퍼 파라미터를 최적화 하는데 오랜 시간(몇 일 정도)이 걸릴 경우 사용한다. Multi-Fidelity Optimization을 통해 high-fidelity evaluation과 low-fidelity evaluation과 같이 다양한 수준에서 샘플을 평가할 수 있다. High-fidelity evaluation은 데이터 셋에서 정밀한(precise) 결과를 출력해내며, 반대로 low-fidelity evaluation은 더 cheap한 결과를 출력한다.

Multi-Fidelity Evaluation의 기본적인 개념은 다수의 low-fidelity evaluation을 사용해 최종적인 evaluation cost를 줄이는 방법이다. 비록, low fidelity optimization으로 인해 최적화 성능이 저하될 수 있지만, 최적화되는 속도가 빠르다.

▼ 영어와 이것저것



Fidelity: 충실함 / 정확도 / 충실도

Multi-fidelity optimization is an optimization technique which focuses on decreasing the evaluation cost by combining a large number of cheap low-fidelity evaluations and a small number of expensive high-fidelity evaluation. In practice, such optimization technique is essential when dealing with big datasets as training one hyper-parameter may take days.

More specifically, in multi-fidelity optimization, we can evaluate samples in different levels. For example, we may have two evaluation functions: high-fidelity evaluation and low-fidelity evaluation. The high-fidelity evaluation outputs precise evaluation from the whole dataset. On the other hand, the low-fidelity evaluation is a cheaper evaluation from a subset of the dataset.

The idea behind the multi-fidelity evaluation is to use many low-fidelity evaluation to reduce the total evaluation cost. Although the low fidelity optimization results in cheaper evaluation cost that may suffer from optimization performance, but the speedup achieved is more significant than the approximation error.

1. Grid Search vs Random Search

보통 하이퍼파라미터를 최적화할때 보편적으로 많이 사용하는 방법은 다음 세가지 방법이다. 이 세 방법은 앞으로 알아볼 HPO 기법보다 직관적이기 때문에 많이 사용한다.

- Manual search: 일일이 연구자가 어떤 파라미터가 성능 향상에 좋은 지를 살펴보고 비교한다
- Grid Search: 파라미터의 Grid를 설정하고 범위 내의 모든 파라미터 조합을 사용하여 비교한다
- Random Search: 파라미터의 Grid를 설정하지만, 범위 내의 Random 조합을 사용하여 비교한다.


1.1 Grid Search

Grid Search는 모델 하이퍼 파라미터를 넣을 수 있는 값들을 순차적으로 입력한 뒤에 가장 높은 성능을 보이는 하이퍼 파라미터의 조합을 탐색하는 방법이다. 즉, 사전에 정의된 n^k 쌍의 하이퍼파라미터를 탐색하는 방법이다. 그렇기 때문에 Grid Search는 computationally expensive한 방법이며, 차원이 높아질수록 사용이 불가하며, 차원의 저주 문제가 발생한다.

정리하자면, Grid Search의 장점과 단점은 다음과 같다.

- 장점: 구현이 쉬우며, 각 trial이 독립적이므로 병렬화가 용이하다
- 단점

- Brute-force 탐색이며, Global optimum, 즉 최적의 하이퍼파라미터 조합을 찾기 위해서는 모든 조합을 실행해야 하기 때문에 많은 컴퓨팅 파워가 요구된다. (동작의 비효율성)
- 최적의 해를 찾을 수 있다는 보장 없음

 code :

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV

# 데이터를 로딩하고 학습데이터와 테스트 데이터 분리
iris = load_iris()
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target,
                                                    test_size=0.2, random_state=121)

dtree = DecisionTreeClassifier()

### parameter 들을 dictionary 형태로 설정
parameters = {'max_depth':[1,2,3], 'min_samples_split':[2,3]}

# param_grid의 하이퍼 파라미터들을 3개의 train, test set fold 로 나누어서 테스트 수행 설정.
### refit=True 가 default 임. True이면 가장 좋은 파라미터 설정으로 재 학습 시킴.
grid_dtree = GridSearchCV(dtree, param_grid=parameters, cv=3, refit=True)

# 붓꽃 Train 데이터로 param_grid의 하이퍼 파라미터들을 순차적으로 학습/평가 .
grid_dtree.fit(X_train, y_train)

# GridSearchCV 결과 추출하여 DataFrame으로 변환
scores_df = pd.DataFrame(grid_dtree.cv_results_)
scores_df[['params', 'mean_test_score', 'rank_test_score', \
            'split0_test_score', 'split1_test_score', 'split2_test_score']]
```

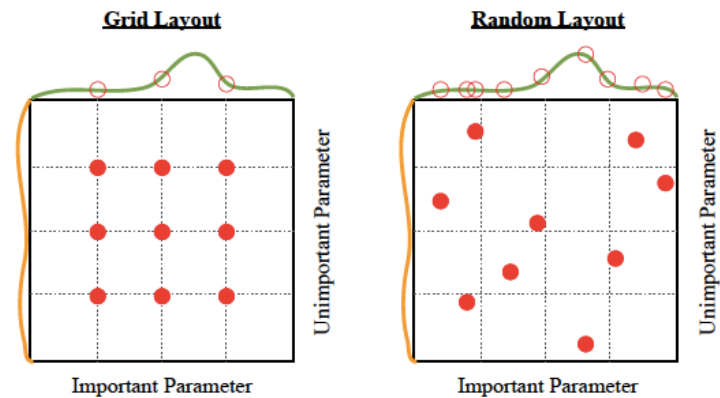
1.2 Random Search


랜덤 서치는 임의의 하이퍼파라미터를 선정하는 과정을 통해 최적의 해를 찾아가는 기법으로, 몇몇 하이퍼 파라미터가 다른 하이퍼 파라미터에 비해 더 중요한 경우 좋은 성능을 보인다. 즉, 사전에 정의된 grid 상에서 무작위로 뽑힌 n 쌍의 hyper-parameter를 탐색하는 방법이다. 여기서 n 은 number of trials이다. Grid search는 지정한 값에 대해서만 학습과 검증을 시도하지만 Random search는 범위 내에 있는 임의의 값에 대하여 예상치 못한 조합을 시도해볼 수 있으며, 최적의 해를 확률적으로 더 빠르게 찾을 수 있다는 장점이 있다.

랜덤 서치의 장단점은 다음과 같다.

- 장점:
 - 구현이 쉬우며, 각 trial이 독립적이며, 그리드 서치에 비해 병렬화가 용이하다.
 - 시간복잡도가 $O(n)$ 으로 그리드 서치보다 빠르다 ($O(n^k)$)
 - 동일한 차원하에서 grid search에 비해 global optimum을 찾을 확률이 더 높다
- 단점:

- Grid search에 비해 결과 해석이 어려움
- 사전지식이 반영되지 않았기 때문에 불필요한 탐색 반복



 code:

```
from sklearn.model_selection import RandomizedSearchCV

random_search = {'criterion': ['entropy', 'gini'],
                  'max_depth': [2],
                  'max_features': ['auto', 'sqrt'],
                  'min_samples_leaf': [4, 6, 8],
                  'min_samples_split': [5, 7, 10],
                  'n_estimators': [20]}

clf = RandomForestClassifier()
model = RandomizedSearchCV(estimator = clf, param_distributions = random_search, n_iter = 10,
                           cv = 4, verbose= 1, random_state= 101, n_jobs = -1)

model.fit(X_Train, Y_Train)

print(model.best_params_)

randompredict = model.best_estimator_.predict(X_Test)
print(classification_report(Y_Test, randompredict))
print("\nRandom Search : ", accuracy_score(Y_Test, randompredict))
```

▼ 🙄(데이터)병렬화란 무엇인가 🙄

데이터 병렬화란 어떠한 데이터 집합을 분해한 뒤, 각 프로세서에 할당하여 동일한 연산을 수행하는 방법으로 벡터화라고도 부른다.



▼ 왜 Grid Search보다 Random Search가 더 적은 탐색 횟수로 좋은 성능을 가져오는지

하나의 중요한 파라미터와 덜 중요한 파라미터가 있다고 가정하자. Grid Search 같은 경우, 중요 파라미터의 3개의 지점을 살펴볼 수 있지만, Random Search는 9개의 distinct한 지점에 대하여 모두 살펴볼 수 있기 때문에, 최적의 파라미터를 발견할 수 있는 가능성이 높아지는 것이다.

<https://www.jmlr.org/papers/volume13/bergstra12a/bergstra12a.pdf>

1.3 Grid Search와 Random Search의 문제점

Grid Search와 Random Search는 기존의 trial의 결과값(loss, acc 등의 평가지표)과 독립적이기 때문에 성능이 좋지 않았던 값들에 대해서도 모두 끝까지 탐색해야하기 때문에 많은 계산, 즉 컴퓨팅 파워를 필요로 한다. **맞다.. 컴퓨터 과로사를 일으키기 좋은 기화다..**

예를 들어, Grid Search 같은 경우 10개의 하이퍼파라미터가 (1,100)의 range 후보를 가지고 있다면, 100^{10} 의 탐색을 진행해야 하며, random search도 범위 내에서 마구잡이로 탐색을 하기 때문에 정확한 탐색이 불가하기 때문에 많은 컴퓨팅 파워가 필요하다.

따라서, 이러한 방법이 아닌 특정한 기준을 가진 모델을 통해, 파라미터를 탐색해나가는 방법이 필요하다. 즉, 맨 처음 선택한 하이퍼파라미터에 대해 어떤 하이퍼파라미터를 조절해야지 성능이 향상될지 알려주는 무언가가 있다면, 마구잡이로 탐색할 필요가 없어진다. 이런 모델을 바로 'Sequential Model-based Global Optimization(SMBO)'라고 부르며, 이는 sequential하게 모델에 기반하여, 전체 파라미터 공간에 대한 탐색을 하는 최적화 방법이다.

- Grid Search와 Random Search 모두, 바로 다음 번 시도할 후보 hyperparameter 값을 선정하는 과정에서, 이전까지의 조사 과정에서 얻어진 hyperparameter 값들의 성능 결과에 대한 **‘사전 지식’**이 전혀 반영되어 있지 않음
- 매 회 새로운 hyperparameter 값에 대한 조사를 수행할 시 **‘사전 지식’**을 충분히 반영하면서, 동시에 전체적인 탐색 과정을 체계적으로 수행할 수 있는 방법론으로, Bayesian Optimization을 들 수 있음

2. SMBO

SMBO는 다음으로 시도해볼만한 파라미터 조합에 대하여 근거를 가지고 최적화를 진행하는 방법이다. 이를 이해하기 위해서는 Surrogate Model과 Acquisition function이 무엇인지 알아야 한다.

2.1 Surrogate Model

Surrogate Model은 현재까지 조사된 입력값-함수결과값 점들($x_1, f(x_1)$)을 바탕으로 미지의 목적 함수의 형태에 대한 확률적인 추정을 수행하는 모델이다. 즉, 목적함수가 어떻게 생겼는지 알수 없으므로 지금까지 나온 결과들을 통해 목적함수를 어느정도 대체할 수 있을 정도로 확률적인 표현을 하는 모델이다.

```

SMBO( $f, M_0, T, S$ )
1    $\mathcal{H} \leftarrow \emptyset$ ,
2   For  $t \leftarrow 1$  to  $T$ ,
3      $x^* \leftarrow \operatorname{argmin}_x S(x, M_{t-1})$ ,
4     Evaluate  $f(x^*)$ ,  $\triangleright$  Expensive step
5      $\mathcal{H} \leftarrow \mathcal{H} \cup (x^*, f(x^*))$ ,
6     Fit a new model  $M_t$  to  $\mathcal{H}$ .
7   return  $\mathcal{H}$ 

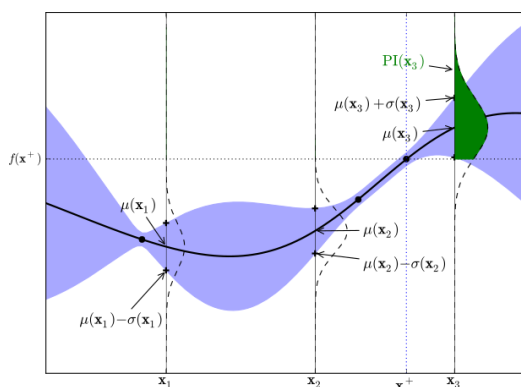
```

Figure 1: The pseudo-code of generic Sequential Model-Based Optimization.

Pseudo code를 확인해보면, objective function의 관측값이 surrogate model의 관측값이 되며, surrogate model은 objective function의 관측되지 않은 값들이 어떻게 이루어져있는지 설명해주고 있음을 볼 수 있다.

2.2 Acquisition function

Acquisition function은 목적 함수에 대한 현재까지의 확률적 추정 결과를 바탕으로, ‘최적 입력값을 찾는 데 있어 가장 유용할 만한’ 다음 입력값의 후보를 추천하는 함수이다. 즉, Acquisition function은 surrogate model이 목적함수에 대하여 실제 데이터를 기반으로 다음번 조사할 x값을 확률적으로 계산하여 추천해주는 함수이다.



가우시안 프로세스로 모델링한 미지의 목적함수를 좀 더 잘 추정하거나 또는 그 최대값을 구하기 위해서, 그 다음 iteration에서 데이터를 샘플링하는데 필요한 포인트를 선정해야하는데, 데이터셋을 최소화하기 위해서는 어떤 샘플링 전략이 필요할 것이다. 두가지 극단적인 전략을 생각해 볼 수 있는데, 하나는 가우시안 프로세스의 불확실성을 최소화할 수 있는 포인트를 선정하는 것이고, 다른 하나는 함수의 최대값과 가까울 것으로 예측되는 포인트를 선정하는 것이다.

- 탐색(exploration) 전략: 함수의 공분산이 크다는 것을 곧 불확실성이 크다는 의미이므로, 공분산이 최대인 지점을 다음 샘플링 포인트로 선정

- 활용(exploitation) 전략: 현재까지 취득한 데이터를 기반으로 함수의 최대값으로 예측되는 지점인 평균함수를 최대화하는 점을 다음 샘플링포인트로 지정, 즉, 기존에 취득한 데이터에 기대어 최대값을 예측(탐욕적 방법)

기본적으로, 가장 적절한 다음 후보는 지금까지 좋았던 결과 근처에 있거나 지금까지 탐색하지 않았던 곳에 있을 것이다. 이 두 가지 지역을 탐색하기 위해 필요한 것이 exploitation과 expectation 방법인데, 이 둘은 trade-off 관계를 가진다. 따라서, 획득함수는 이 두가지를 잘 조율해서 모두 일정한 수준을 포함하도록 한다.

획득함수의 종류는 다양한데, 가장 자주 쓰이는 improvement 기반의 획득 함수에 대해서 알아보자.

2.2.1 Probability of Improvement(PI)

Probability of Improvement(PI)는 어느 후보 입력값 x 에 대하여, 현재까지 조사된 데이터의 함수값 중 최대 함수값을 찾는 방법으로, 찾은 최대값보다 더 큰 함수값을 도출할 확률을 PI라고 한다. 즉, 이는 블랙박스 함수 f 에 대하여 지금까지 최적의 하이퍼 파라미터인 λ^* 보다 더 좋은 성능을 내는 하이퍼파라미터에 대한 확률을 통해 다음 후보를 찾는 방법으로도 볼 수 있다.

$$\lambda^+ = \arg \max_{\lambda} PI(\lambda) = \arg \max_{\lambda} P(f(\lambda) \leq f(\lambda^*))$$

개념은 grid search와 상당히 유사해보이지만, PI는 확률값만 계산하고 실험(fitting)을 하지 않는다는 점에서 다르다. 즉, 탐색보다는 extraction에 초점을 맞추고 있다. 이때, 이미 얻은 최적의 하이퍼파라미터 근처의 하이퍼파라미터만 반환하는 경우가 있는데, 이때는 exploration과 exploitation의 비율을 조절하면 된다.

$$\lambda^+ = \arg \max_{\lambda} P(f(\lambda) \leq f(\lambda^*) - \epsilon)$$

이때, $\epsilon \geq 0$ 의 비율을 설정해야 하는데, ϵ 값이 너무 크거나 작으면 문제가 발생한다. ϵ 이 0에 가까울수록 작다면 기존의 PI와 차이가 없어 이미 얻은 최적의 하이퍼파라미터 근처의 하이퍼파라미터만 반환하는 문제가 발생한다. 반대로 ϵ 이 너무 크다면, 어떤 하이퍼파라미터가 나오더라도 PI 값이 크게 나올 수 있다. 따라서 적당히 조절해야 한다.

▼ PI를 Gaussian으로 나타낸 경우 + 코드

$$\begin{aligned} MPI(x|D) &= \arg \max_x P(y \geq (1+m)y_{max}|x, D), \quad y \sim N(\mu, \sigma^2) \\ &= \arg \max_x P\left(\frac{y - \mu}{\sigma} \geq \frac{(1+m)y_{max} - \mu}{\sigma}\right) \\ &= \arg \max_x \left\{1 - \Phi\left(\frac{(1+m)y_{max} - \mu}{\sigma}\right)\right\} \\ &= \arg \max_x \Phi\left(\frac{\mu - (1+m)y_{max}}{\sigma}\right) \end{aligned}$$

- y_{max} 와 std를 기준으로 mean의 Z (Standardized normal)을 뺀다
- x_i 는 y_{max} 를 어느정도 이동시키는 역할을 한다

- \max 값과 $f(x^+)$ 간의 차이값이 클수록 Z는 음수에서 양수로 값은 계속 커질 것이다. 즉, y_{\max} 보다 값이 커질 예정이다
- 종합적으로 고려하여, acquisition function은 PI가 가장 높은 구간을 선택할 것이다.

```
from scipy.stats import norm
# ppf : 누적분포함수의 역함수(inverse cumulative distribution function)
>>> a = norm.ppf(0.05)
>>> b = norm.ppf(0.95)
>>> print(a, b)

-1.6448536269514729
1.6448536269514722

# pdf(x) : 확률분포의 x에 대한 확률
# cdf(x) : 확률분포의 x에 대한 누적확률
>>> print(norm.pdf(a))
>>> print(norm.pdf(b))
>>> print(norm.cdf(a))
>>> print(norm.cdf(b))

0.10313564037537128
0.10313564037537139
0.049999999999999975
0.95

>>> c = [a,b]
>>> print(norm.cdf(c))
[0.05, 0.95]

@staticmethod
def _poi(x, gp, y_max, xi):
    with warnings.catch_warnings():
        warnings.simplefilter("ignore")
        mean, std = gp.predict(x, return_std=True)

    z = (mean - y_max - xi)/std
    return norm.cdf(z)
```

2.2.2 Expected Improvement(EI)

PI는 현재 최적의 하이퍼파라미터를 기준으로 결과가 개선될 확률만 보이기 때문에 실제로 어느 정도 개선이 되었는지 알 수 없다. 이런 문제를 해결하는 것이 EI이며, 이는 개선 확률이 아닌 얼마나 개선할 수 있는지를 나타내는 지표이다. 즉, $f(\lambda^*)$ 관점에서 더 개선할 수 있는 정도의 기댓값을 최대화하는 다음 후보를 찾는 방법으로 계산한 PI 값에서 지금 \max 인 값과 \max 로 예상되는 값의 차이만큼 가중하여 다음에 시행할 EI를 최종적으로 계산한다.

$$\lambda^+ = \arg \max_{\lambda} EI(\lambda) = \arg \max_{\lambda} E[\max(0, f(\lambda^*) - f(\lambda)) | D]$$

식에서 보이다시피 하이퍼 파라미터의 함수값이 기존보다 안좋아 질 경우엔 0을 기댓값으로 출력한다. 이제 $y = f(\lambda)$ 로 설정한 다음 EI를 적분 식으로 나타내면 다음과 같다.

$$EI(\lambda) = \int_{-\infty}^{\infty} \max(0, f(\lambda^*) - y)p(y|\lambda)dy$$

실제로 EI가 PI보다 성능이 좋은 경우가 많기 때문에, 베이지안 최적화에서는 EI를 주로 많이 쓴다. PI보다 성능이 좋은 이유는 Local Minima의 문제를 어느정도 해결하기 때문이다. PI 같은 경우, 몇 번의 시도 끝에 얻은 하이퍼파라미터 근처에서 최적화 결과를 얻어 local minima에 빠지는 문제가 발생하나, EI는 PI보다 탐색비중이 더 높기 때문에 local minima에 빠질 확률이 더 낮다.

2.3 Bayesian Optimization

블랙박스 모델에 대한 global 최적화를 위한 방법론으로 어떤 입력값(x)를 받는 미지의 목적 함수 (f(x))를 상정하여 해당 함수값 f(x)를 최대로 만드는 최적의 해를 찾는 것이 목적이다. 즉, 목적함수와 하이퍼파라미터의 쌍을 대상으로 surrogate model를 만들고, 순차적으로 하이퍼 파라미터를 업데이트해가면서 평가를 통해 최적의 하이퍼파라미터 조합을 탐색한다.

베이지안 최적화에는 다음과 같은 두 가지 필수 요소가 존재한다.

1. Surrogate Model: 현재까지 조사된 입력값-함수결과값 점들(x1,f(x1))을 바탕으로 미지의 목적 함수의 형태에 대한 확률적인 추정을 수행하는 모델
2. Acquisition Function: 목적 함수에 대한 현재까지의 확률적 추정 결과를 바탕으로 최적 입력값을 찾는 데 있어 가장 유용할 만한 다음 입력값 후보를 추천해주는 함수

이 두가지 요소를 알아보기 전에, 먼저 베이지안 최적화의 의사코드를(Pseudo-code) 알아보자.

Algorithm 1: Bayesian Optimization

```

1 BayesianOptimization (f, S0, T, A);
   Input : Blackbox function f,
           Initiated surrogate function S0,
           Max number of iterations T,
           Acquisition function A

   Output: H
2 H ← ∅
3 for t ← 0 to T by 1 do
4   λ* ← argmaxλ ∈ Λ A(λ, St-1)
5   Evaluate f(λ*) // Expensive step
6   H ← H ∪ (λ*, f(λ*))
7   Fit a new model St to H
8 end
9 return H

```

과정을 살펴보자면, 블랙박스 함수 f(손실함수), surrogate function, 최대 반복 횟수, acquisition function을 input 값으로 넣으면, output으로 H를 반환하는 형태이다. 더 자세히 알아보자면, 우선 하이퍼 파라미터와 그 하이퍼 파라미터로 얻은 모델에 대한 f값을 저장할 집합 H를 초기화한 다음, acquisition function을 통해 하이퍼 파라미터의 후보를 찾는다. 그 다음 그 후보값을 이용해 f를 계산하여 H에 저장한다음, 최종적으로 surrogate function에 적합시킨다.

▼ 파이썬으로 베이지안 최적화 구현해보기

```

from hyperopt import hp, fmin, tpe, STATUS_OK, Trials ## HyperOpt라는 프레임 워크를 통해 사

space = {'criterion': hp.choice('criterion', ['entropy', 'gini']), ## hp.ch

```

```

oice: 주어진 리스트 내에서 sampling
    'max_depth': hp.quniform('max_depth', 10, 12, 10),          ## hp.qu
niform: 지정된 범위 내에서 일정 간격을 두어 sampling
    'max_features': hp.choice('max_features', ['auto', 'sqrt', 'log2', None]),
    'min_samples_leaf': hp.uniform('min_samples_leaf', 0, 0.5),  ## hp.un
iform: 지정된 범위 내에서 random sampling
    'min_samples_split' : hp.uniform('min_samples_split', 0, 1),
    'n_estimators' : hp.choice('n_estimators', [10, 50])
}

def objective(space): ## 목적함수
    ## RandomForest 설정:
    hopt = 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 -> 교차검증
    accuracy = cross_val_score(hopt, X_Train, Y_Train, cv = 4).mean()
    return {'loss': -accuracy, 'status': STATUS_OK }

trials = Trials()

"""
## fmin: Minimize a function using the downhill simplex algorithm.
scipy의 fmin 함수: 함수의 최솟값을 구하기 위한 최적화 알고리즘 중 하나
이 함수는 주어진 초기 추정값(initial guess)을 시작으로
주어진 함수를 최소화하는 파라미터 값을 찾음
"""
best = fmin(fn= objective,
           space= space,
           algo= tpe.suggest,
           max_evals = 20,
           trials= trials
           )

# 최적의 하이퍼파라미터 조합
best

### 최적 파라미터 적용하기
crit = {0: 'entropy', 1: 'gini'}
feat = {0: 'auto', 1: 'sqrt', 2: 'log2', 3: None}
est = {0: 10, 1: 50, 2: 75, 3: 100, 4: 125}

trainedforest = RandomForestClassifier(criterion = crit[best['criterion']],
                                     max_depth = best['max_depth'],
                                     max_features = feat[best['max_features']],
                                     min_samples_leaf = best['min_samples_leaf'],
                                     min_samples_split = best['min_samples_split'],
                                     n_estimators = est[best['n_estimators']]
                                     ).fit(X_Train, Y_Train)

predictionforest = trainedforest.predict(X_Test)
print(classification_report(Y_Test, predictionforest))

```

2.4 Surrogate Model(1) - Gaussian Process

베이지안 최적화에 있어 가우시안 프로세스는 전통적으로 많이 사용한 모델이며, 가우시안 프로세스 (GP)란 프로세스 집합 내에 있는 랜덤 변수들의 임의의 조합이 모두 결합 가우시안 분포를 갖는 랜덤 프로세스로 정의된다.

- 랜덤변수(Random Variables) : 확률 실험의 결과에 실수값을 대응시키는 함수로 정의
- 랜덤프로세스(Random Process): 어떤 파라미터로 인덱스 된 무한개의 랜덤변수의 집합으로 정의
→ 확률 실험 결과와 인덱스 파라미터 등 두개의 변수로 구성된 함수
 - 예) 인덱스 x_1, x_2, \dots, x_m 에 해당하는 랜덤 변수가 $f_i = f(x_i)$ 일 때, 이로 부터 가능한 모든 부분 집합 $\{f_1\}, \{f_1, f_2\}, \dots, \{f_1, f_2, \dots, f_m\}$ 이 모두 결합 가우시안 분포를 갖는 프로세스
 - m은 임의로 선정한 인덱스의 갯수 → 가우시안 랜덤 벡터는 무한 차원으로 설명 가능
 - 가우시안 프로세스 → 가우시안 랜덤벡터를 무한 차원으로 확장한 것

GP의 가장 기본적인 아이디어는 $y = f(x)$ 로 함수 f 에 x 입력값을 넣었을 때 나타나는 y 가 하나의 값이 아닌 σ 를 표준편차로 갖는 분포의 평균값이다.

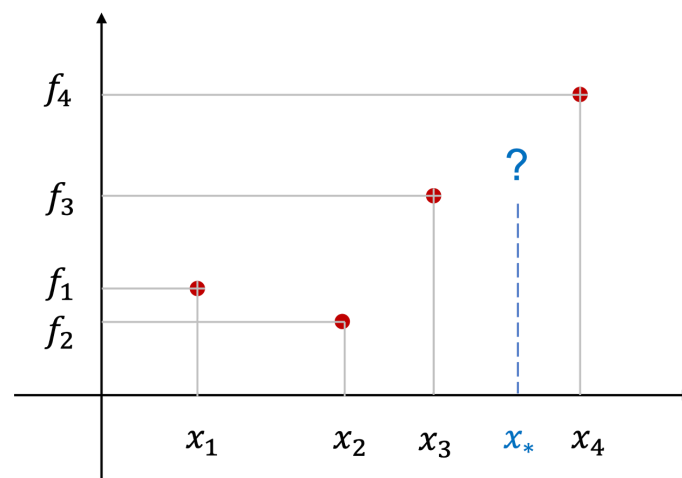
1. 가우시안 프로세스 회귀

가우시안 프로세스도 평균함수 $\mu(x)$ 와 공분산 $k(x, x')$ 로 특징지을 수 있음

$$f(x) \sim \mathcal{GP}(\mu(x), k(x, x'))$$

$$k(x, x') = E[(f(x) - \mu(x))(f(x') - \mu(x'))]$$

다음과 같이 m개의 데이터셋 $\mathcal{D} = \{(x_1, f_1), (x_2, f_2), \dots, (x_m, f_m)\}$ 이 주어졌을 때, 데이터셋에 없는 x_* 에 대응하는 종속변수 f_* 을 추정해본다고 하자.



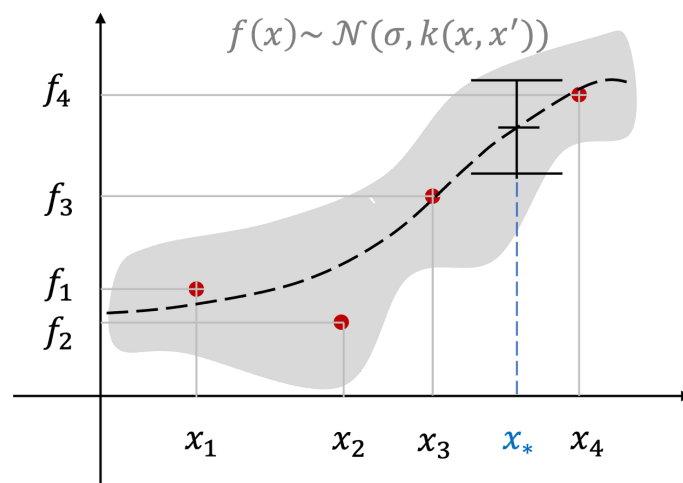
- GP 회귀 문제에서는 우리가 적합하고자하는 목표함수가 특정 형태를 갖는다고 가정하는 대신에 가우시안 프로세스 $f(x)$ 로 간주 → 구현 가능한 모든 함수에 대해 확률을 부여

- 미지의 함수를 가우시안 프로세스 $f(x)$ 로 표현 $\rightarrow f_{1:m} = [f_1, f_2, \dots, f_m]^T$ 를 어떤 가우시안 프로세스의 특정 인덱스 $x_{1:m}$ 에서 샘플링 된 값으로 간주

- $f_{1:m} \sim \mathcal{N}(\mu(x_{1:m}), K)$

- $x_{1:M} = x_1, x_2, \dots, x_m$

- $$K = \begin{bmatrix} k(x_1, x_1) & k(x_1, x_2) & \cdots & k(x_1, x_m) \\ k(x_2, x_1) & k(x_2, x_2) & \cdots & k(x_2, x_m) \\ \vdots & \vdots & \ddots & \vdots \\ k(x_m, x_1) & k(x_m, x_2) & \cdots & k(x_m, x_m) \end{bmatrix}$$



이렇게 GP regression은 예측값의 신뢰구간 혹은 비신뢰구간을 알 수 있다.

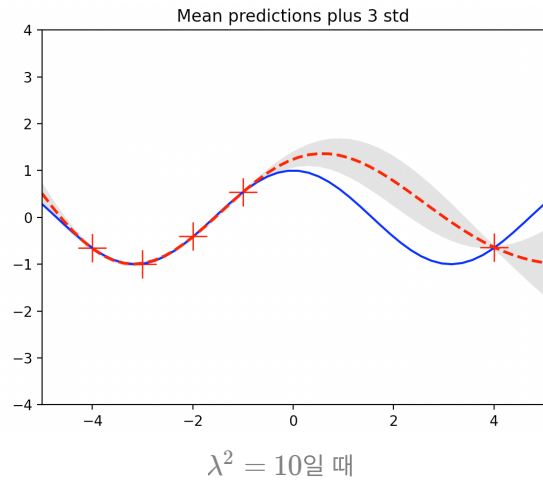
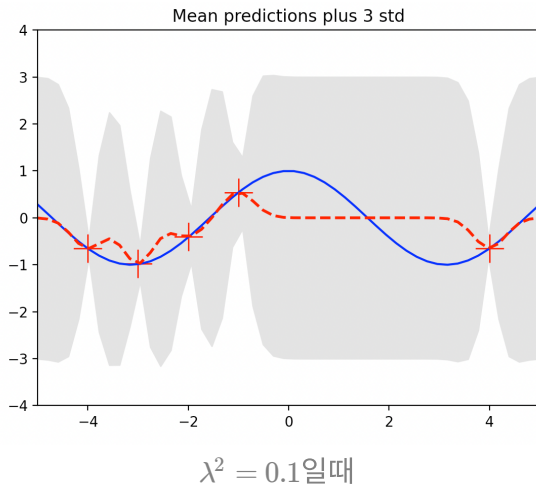
위 그림은 GP regression으로 예측된 mean function와 mean이 가질 수 있는 분산을 area로 표현해서 신뢰구간을 표현한 것이다.

- 평균함수 $\mu(x)$: 관측값의 전반적인 추세를 나타냄 \rightarrow 미지의 목표함수에 대한 사전정보가 없는 경우 0으로 설정
- 커널함수 $k(x, x')$: 두 관측값 사이의 기하학적인 유사도를 나타냄
 - 예) RBF(radial basis function) : $k(x, x') = \sigma^2 \exp(-\frac{\|x - x'\|^2}{2\lambda^2})$
 - σ_f^2 : 커널이 표현할 수 있는 최대 공분산의 크기 $\leftarrow x \approx x'$ 일 때 (서로 밀접할 때)
 - λ^2 : 관련성의 정도 조절
 - $\sigma_f^2, \lambda^2 \rightarrow$ 커널 파라미터



커널 파라미터에 따른 비교

- λ^2 값이 작을 수록 데이터셋의 관련성 작게 반영 → 분산값이 매우 커짐
- λ^2 값이 클수록 데이터셋의 관련성 과대하게 반영 → 분산값이 매우 작아지지만 편향 커짐



GP 회귀 문제는 m 개의 데이터셋 $\mathcal{D} = \{(x_1, f_1), (x_2, f_2), \dots, (x_m, f_m)\}$ 이 주어졌을 때 p 개의 입력 $\{x_{m+1}, x_{m+2}, \dots, x_{m+p}\}$ 에 대응하는 $f_* = [f_{m+1}, f_{m+2}, \dots, f_{m+p}]^T$ 를 추정하는 문제이다.

즉, 다음과 같은 가우시안 분포가 주어질 때,

$$\begin{bmatrix} f_{1:m} \\ f_* \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \mu(x_{1:m}) \\ \mu(x_{m+1:m+p}) \end{bmatrix}, \begin{bmatrix} K & K_* \\ K_*^T & K_{**} \end{bmatrix}\right)$$

$$K_* = \begin{bmatrix} k(\mathbf{x}_1, \mathbf{x}_{m+1}) & \cdots & k(\mathbf{x}_1, \mathbf{x}_{m+p}) \\ \vdots & \ddots & \vdots \\ k(\mathbf{x}_m, \mathbf{x}_{m+1}) & \cdots & k(\mathbf{x}_m, \mathbf{x}_{m+p}) \end{bmatrix}$$

$$K_{**} = \begin{bmatrix} k(\mathbf{x}_{m+1}, \mathbf{x}_{m+1}) & \cdots & k(\mathbf{x}_{m+1}, \mathbf{x}_{m+p}) \\ \vdots & \ddots & \vdots \\ k(\mathbf{x}_{m+p}, \mathbf{x}_{m+1}) & \cdots & k(\mathbf{x}_{m+p}, \mathbf{x}_{m+p}) \end{bmatrix}$$

f_* 의 조건부 확률 밀도 함수 $p(f_* | x_{1:m}, x_{m+1:m+p}, f_{1:m})$ 를 구하는 문제이다.

가우시안 프로세스 $f(x)$ 의 관측값에는 노이즈가 포함되어 있다고 가정하는 것이 일반적이다.

노이즈를 평균이 0이고 분산이 σ_n^2 인 가우시안으로 모델링한다고 가정하면 GP측정 모델은 다음과 같다.

$$y = f(\mathbf{x}) + \epsilon$$

$$\epsilon \sim \mathcal{N}(0, \sigma_n^2),$$

$$f(\mathbf{x}) \sim \mathcal{GP}(\mu(\mathbf{x}), k(\mathbf{x}, \mathbf{x}'))$$

노이즈가 가우시안 프로세스와 독립이라고 가정하면, 가우시안 프로세스 y 의 평균과 공분산 (covariance)은 다음과 같이 된다.

$$\mathbb{E}[y] = \mathbb{E}[f(\mathbf{x}) + \epsilon] = \mu(\mathbf{x})$$

$$\mathbb{E}[(y - \mu(\mathbf{x}))(y' - \mu(\mathbf{x}'))]$$

$$= \mathbb{E}[(f(\mathbf{x}) + \epsilon - \mu(\mathbf{x}))(f(\mathbf{x}') + \epsilon' - \mu(\mathbf{x}'))]$$

$$= \mathbb{E}[(f(\mathbf{x}) - \mu(\mathbf{x}))(f(\mathbf{x}') - \mu(\mathbf{x}'))]$$

$$+ \mathbb{E}[(f(\mathbf{x}) - \mu(\mathbf{x}))\epsilon']$$

$$+ \mathbb{E}[\epsilon(f(\mathbf{x}') - \mu(\mathbf{x}'))] + \mathbb{E}[\epsilon\epsilon']$$

$$= k(\mathbf{x}, \mathbf{x}') + \sigma_n^2 \delta_{ii'}$$

즉, $y \sim \mathcal{GP}(\mu(\mathbf{x}), k(\mathbf{x}, \mathbf{x}') + \sigma_n^2 \delta_{ii'})$ 로 간단히 정리할 수 있다.

그럼 앞서 GP를 정의한 것처럼 조건부 밀도함수 $p(\mathbf{y}|\mathbf{y}_{1:m})$ 를 구해보자.

관련 랜덤벡터의 결합 확률분포는 다음과 같다.

$$\begin{bmatrix} \mathbf{y}_{1:m} \\ \mathbf{y}_* \end{bmatrix} \sim \mathcal{N} \left(\begin{bmatrix} \mu \\ \mu_* \end{bmatrix}, \begin{bmatrix} K + \sigma_n^2 I & K_* \\ K_*^T & K_{**} \end{bmatrix} \right)$$

$$\mu = \mu(\mathbf{x}_{1:m}), \mu_* = \mu(\mathbf{x}_{m+1:m+p})$$

이를 통해 구한 조건부 확률 밀도함수의 평균과 공분산은 다음과 같이 주어진다.

$$p(\mathbf{y}_*|\mathbf{y}_{1:m}) = \mathcal{N}(\mu_{pos}, \Sigma)$$

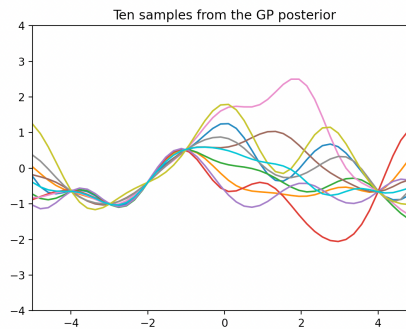
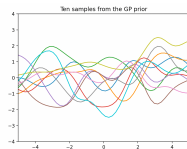
$$\Sigma = K_{**} - K_*^T [K + \sigma_n^2 I]^{-1} K_*$$

$$\mu_{pos} = \mu_* + K_*^T [K + \sigma_n^2 I]^{-1} (\mathbf{y}_{1:m} - \mu)$$

(위의 수식은 역행렬 계산이 포함되어 출레스키 분해를 이용해 연산을 수행한다.)

2. GP Prior & GP Posterior

- 사전 프로세스(GP prior) : 앞서 미지의 목표함수에 대해 사전 정보가 없을 경우 평균함수를 0으로 설정한다고 했다. 이처럼 처음에 가정했던 가우시안 프로세스 확률 정보를 이용한 것을 사전 프로세스라고 한다.
- 사후 프로세스(GP posterior) : 데이터 셋을 이용하여 GP prior를 업데이트하여 얻은 프로세스



- 앞서 구한 조건부 확률 밀도함수가 Posterior Process인데 이의 공분산 함수는 Prior의 공분산 함수에서 양의 함수를 뺀 값과 같다.
- 즉, Prior의 그것보다 언제나 작은 값을 가진다는 의미인데, 이는 데이터가 우리에게 정보를 제공하여 Posterior의 분산이 감소하였다고 생각하면 논리적이다.

가우시안 프로세스의 문제점은 큰 계산량이다. ~~내 컴퓨터 죽어가는 소리 안들리냐...!~~ 가우시안 프로세스는 데이터의 수 n 에 대해서 $O(n^3)$ 의 계산 복잡도를 가진다. 이런 긴 시간을 요구하는 이유는 행렬 K 의 inverse와 determinant를 구해야 하기 때문이다. 따라서 이런 계산 복잡도를 해결한 모델이 등장하게 된다.

▼ 🌟Brownian Motion and Gaussian Process🌟

2.5 Surrogate Model(2) - Tree-structured Parzen Estimator Approach (TPE)

TPE와 가우시안 프로세스는 어떤 것을 모델링하는가에 차이를 둔다. 먼저, 가우시안 프로세스는 어떤 하이퍼 파라미터에서 어떤 목적함수 값이 나오는지를 모델링한다. 즉, posterior인 조건부 확률 $P(Y|\lambda)$ 를 모델링하는 것이 목표로 prior인 $P(Y)$ 와 likelihood인 $P(\lambda|Y)$ 를 사용한다. 하지만, TPE는 prior를 non-parametric density로 대체해서 $P(\lambda|Y)$ 를 모델링한다. 즉, 주어진 목적 함수의 값에 대해서 하이퍼파라미터의 확률 분포를 모델링하는 것이다.

TPE 역시 하이퍼파라미터 탐색의 매 iteration마다 어떤 조합으로 시도를 할 지에 대한 추천을 해주는데, GP와 크게 두 부분에서 차이가 나타난다.

1. TI 식의 차이

2. 사용하는 분포의 차이: GP = Gaussian / TPE = Parzen Window Density (KDE)

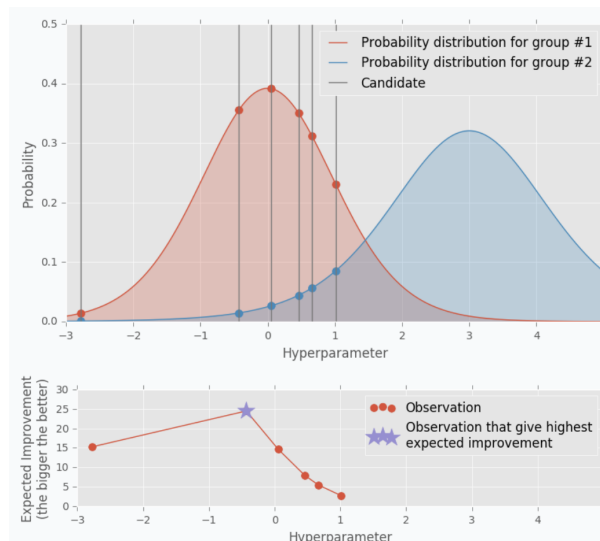
TPE의 알고리즘은 다음과 같다. 먼저, 최초 몇 번의 시행을 통해 목적 함수 값들을 얻어 알고리즘을 초기화한다. 그 다음, 얻은 결과를 acquisition function 값으로 들어가는데, 이때 어떤 기준이 되는 임계값 y^* 이 있다고 하면 $P(\lambda|Y)$ 는 다음과 같이 나눌 수 있다.

$$p(\lambda|y) = \begin{cases} l(\lambda) & \text{if } y < y^* \\ g(\lambda) & \text{if } y \geq y^* \end{cases}$$

$l(\lambda)$ 는 acquisition function의 값이 임계값보다 낮은, 즉 성능이 좋은 하이퍼 파라미터에 대한 분포가 된다. 이때, 분포를 생성할 때 Parzen Estimator를 사용하게 된다. Parzen Estimator는 Kernel density estimator로 생각하면 된다. 즉, 매 데이터들의 gaussian의 평균으로 가정한 분포를 지속적으로 겹쳐서 하나의 mixture model을 구성하는 것이다. 마지막으로, 임계값을 설정해야하는데, 이는 어느 분위수(quantile) γ 를 통해 설정한다.

$$P(y < y^*) = \gamma$$

이 임계값을 설정할때는 $P(y)$ 에 대한 모델 가정이 필요없다. 하지만, 일반적으로 γ 는 0.15나 0.2로 설정한다. 즉, surrogate model에 대해서 특정값보다 크거나 작은 그룹으로 나누는 것이다. 그 다음은, 나눈 그룹들에 대해서 분포를 생성하는데, 이를 TI를 통해 보여주면 아래와 같다.



그래프를 보면, $l(\lambda)$ 에 있을 확률이 $g(\lambda)$ 에 있을 확률보다 높다는 것을 확인할 수 있으며, $l(\lambda)$ 의 값들을 추천 파라미터로 나타내고 있다.

여기서의 TI는 $EI(\lambda) = \frac{l(\lambda)}{g(\lambda)}$ 이다.

▼ EI 최적화

$$EI(\lambda) = \int_{-\infty}^{\infty} \max(0, f(\lambda^*) - y) p(y|\lambda) dy$$

$$EI(\lambda) = \int_{-\infty}^{\infty} \max(0, y^* - y) p(y|\lambda) dy = \int_{-\infty}^{y^*} (y^* - y) p(y|\lambda) dy$$

베이즈 정리와 $p(\lambda)$ 의 marginalization을 통해 다음과 같이 쓸 수 있다.

$$EI(\lambda) = \int_{-\infty}^{y^*} (y^* - y) \frac{p(\lambda|y)p(y)}{p(\lambda)} dy$$

$$p(\lambda) = \int_{\mathbb{R}} p(\lambda|y)p(y) dy$$

$$= \int_{-\infty}^{y^*} p(\lambda|y)p(y) dy + \int_{y^*}^{\infty} p(\lambda|y)p(y) dy$$

$$= \gamma l(\lambda) + (1 - \gamma)g(\lambda)$$

$$\int_{-\infty}^{y^*} (y^* - y) p(\lambda|y)p(y) dy = l(\lambda) \int_{-\infty}^{y^*} (y^* - y) p(y) dy$$

$$= l(\lambda) \int_{-\infty}^{y^*} y^* p(y) - yp(y) dy$$

$$= l(\lambda) \gamma y^* - l(\lambda) \int_{-\infty}^{y^*} yp(y) dy$$

$$= l(\lambda) \gamma y^* - l(\lambda) A$$

지금까지 결과로 EI 식을 정리하면 다음과 같다.

$$EI(\lambda) = \int_{-\infty}^{y^*} (y^* - y) \frac{p(\lambda|y)p(y)}{p(\lambda)} dy$$

$$= \frac{l(\lambda) \gamma y^* - l(\lambda) A}{\gamma l(\lambda) + (1 - \gamma)g(\lambda)}$$

$$= \frac{\gamma y^* - A}{\gamma + (1 - \gamma) \frac{g(\lambda)}{l(\lambda)}}$$

$$= C \cdot \left(\gamma + \frac{g(\lambda)}{l(\lambda)} (1 - \gamma) \right)^{-1} \quad \text{where } C = \gamma y^* - A$$

$$\propto \left(\gamma + \frac{g(\lambda)}{l(\lambda)} (1 - \gamma) \right)^{-1}$$

<https://otzslayer.github.io/assets/images/2022-12-03-bayesian-optimization/TPE.gif>

2.6 Optuna

Optuna는 파이썬 기반의 오픈 소스 라이브러리로, 하이퍼 파라미터 최적화를 수행하는데 사용된다. Optuna는 베이지 최적화를 기반으로 하며, 현재까지 시도한 하이퍼 파라미터 값들과 모델의 성능을 이용해 확률 모델을 구축하고, 이를 토대로 최적의 하이퍼 파라미터 값을 추천하기 때에 기존의 그리드 서치 방법보다 효과적이다.

Optuna는 쉽고 간단한 API를 제공하며, 분산 컴퓨팅을 지원하여 병렬로 하이퍼 파라미터 검색을 수행할 수 있다. 또한, 기존의 여러 하이퍼 파라미터 최적화 라이브러리와 호환되며, TensorFlow, PyTorch, Scikit-learn 등 다양한 머신 러닝 프레임워크와 함께 사용할 수 있다.

Optuna의 기능은 다음과 같다.

- 베이지 최적화 알고리즘 기반의 하이퍼 파라미터 최적화
- 다양한 모델과 라이브러리와 호환성
- 분산 컴퓨팅 지원
- 간단하고 직관적인 API 제공

정리하자면, Optuna를 통해 머신 러닝 모델의 성능을 최대한 높일 수 있는 최적의 하이퍼 파라미터 값을 빠르고 쉽게 찾을 수 있다.

코드 (TPE vs Skopt)

define-by-run: 사용자로 하여금 동적으로 탐색공간을 구축하도록 하는 작동방식

- 메트릭을 포함한 목적함수를 최대화/최소화하는 과정
- study: 개별 학습(탐색 과정) / trial : 메트릭을 이용한 학습 과정 평가
- **optimize API** : **objective** 함수를 입력으로 받아, trial 객체를 받음

```
def objective(trial):
    params = {
        '파라미터 1' : ( trial 의 범위 )
        '파라미터 2' : ( trial 의 범위 )
        '파라미터 3' : ( trial 의 범위 )}

    model=(** params)
    (데이터 학습 코드 )

    return 평가지표

import optuna
from optuna import Trial
from optuna.samplers import TPESampler

study = optuna.create_study(
    direction='maximize',
    sampler=TPESampler())

study.optimize(objective,n_trials=10)
```

- **suggest API** : 이전의 평가된 trial들의 내역들을 기반으로 하는 통계적 샘플링을 통해 동적으로 매 trial마다의 하이퍼 파라미터를 생성

→ 넓은 범위의 이질적인 파라미터 탐색 공간들을 제시해줌

cf) *define-and-run* : 탐색 환경 구축 단계와 평가 단계가 각각 분리되어있음(위의 방식은 분리 x)

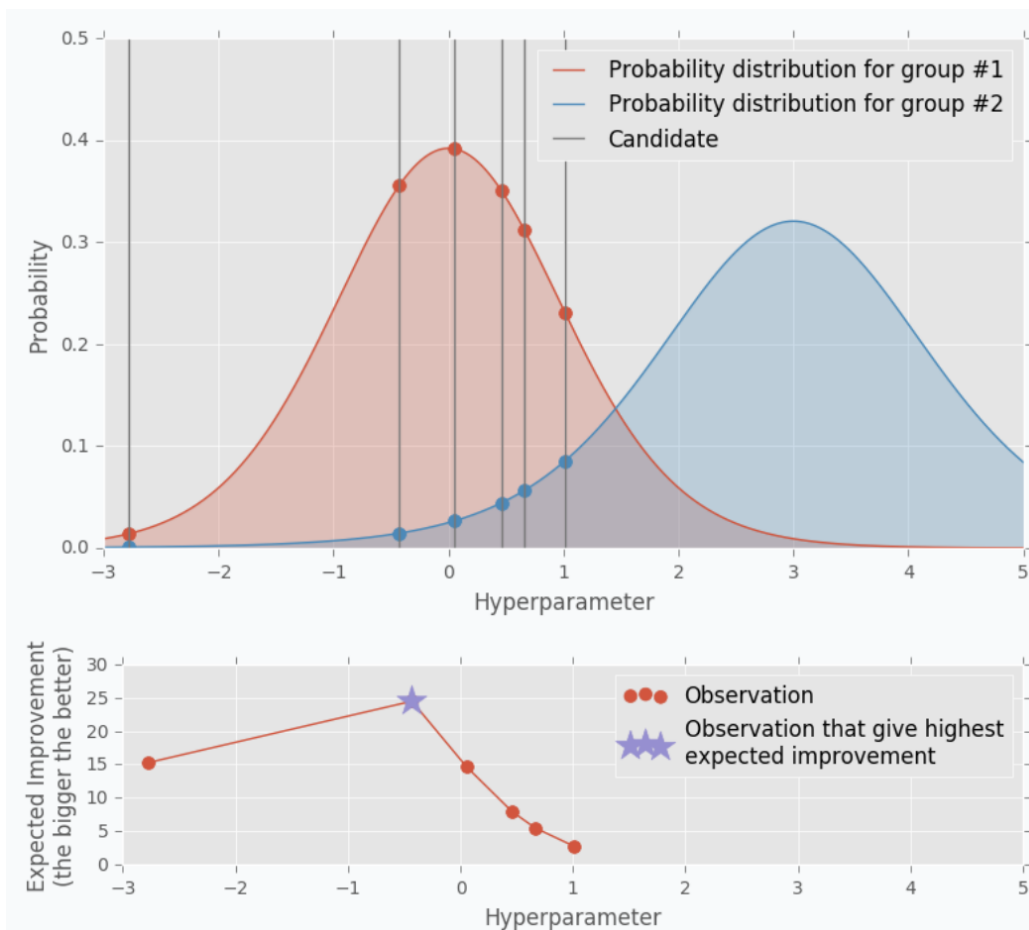
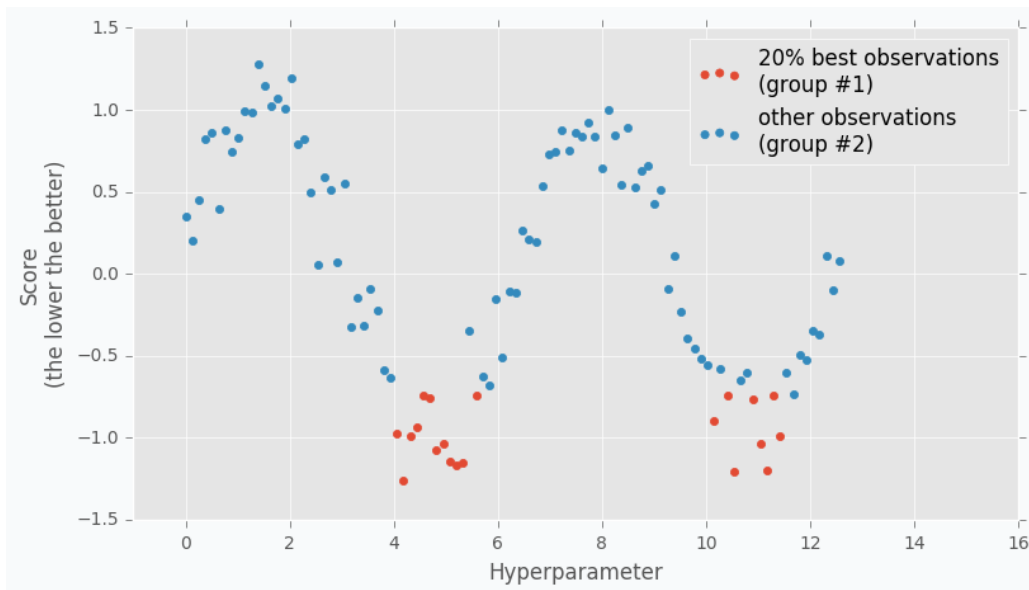
- 효율적인 하이퍼파라미터 탐색을 구현하기 위해 다음 기능을 제공함 :
 - **pruning** : 유의미하지 않은 trial의 사전 제거(자동화된 early stopping)
 - **sampling** : 사용자 정의 샘플러 + 내장 최적화 알고리즘 제공
 - **relational sampling** : 파라미터간 상관관계를 이용해 샘플링 ex) CMA-ES
 - **independent sampling** : 각각의 파라미터를 독립적으로 샘플링 . ex) TPE

	RandomSampler	GridSampler	TPEsampler	CmaEsSampler
Float parameters	✓	✓	✓	✓
Integer parameters	✓	✓	✓	✓
Categorical parameters	✓	✓	✓	▲
Pruning	✓	✓	✓	▲
Multivariate optimization	▲	▲	✓	✓
Conditional search space	✓	▲	✓	▲
Multi-objective optimization	✓	✓	✓	✗
Batch optimization	✓	✓	✓	✓
Distributed optimization	✓	✓	✓	✓
Constrained optimization	✗	✗	✓	✗
Time complexity (per trial) (*)	$O(d)$	$O(dn)$	$O(dn \log n)$	$O(d^3)$
Recommended budgets (#trials) (**)	as many as one likes	number of combinations	100 - 1000	1000 - 10000

TPE Sampler가 파라미터의 형식에 구애받지 않고
유연하게 성능을 발휘함을 확인 가능

TPEsampler

- TPE (Tree-structured Parzen Estimator) 알고리즘 기반 샘플러
- 사용하는 분포의 차이: GP = Gaussian / TPE = Parzen Window Density (KDE)
- *independent sampling* 기반 샘플러
- **$l(x)$** : GMM set of parameter values associated with the best objective values
- **$g(x)$** : another GMM remaining parameter values
- **x** : chooses the parameter value **x** that maximizes the ratio **$l(x)/g(x)=Expected Improvement$**

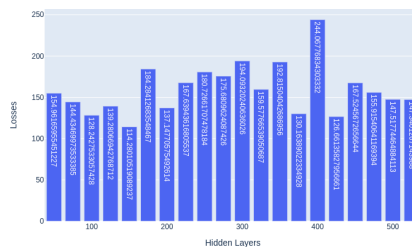


- surrogate model에 대하여 특정값보다 작은 그룹과 큰 그룹으로 나눔
- TPESampler는 적은 시도 횟수에서도 높은 성능을 보이는데, 이는 적은 데이터로도 새로운 샘플링 지점을 제안하기 때문
- 데이터가 적은 경우나 고차원 공간에서는 TPESampler가 적합

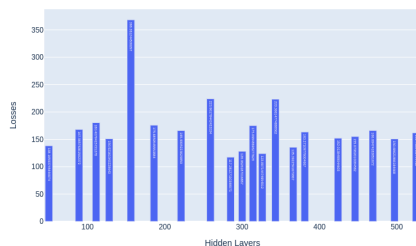
SkoptSampler

- Optuna는 분산 병렬 최적화를 지원하기 위해 다양한 샘플러(Sampler)를 제공합니다. 이 중 ****optuna.integration.SkoptSampler***는 Scikit-Optimize(또는 skopt) 라이브러리를 사용하여 베이지안 최적화를 구현한 샘플러
- SkoptSampler는 기본적으로 Gaussian Process Regression(GP)을 사용하여 모델링
- SkoptSampler는 TPESampler와 달리 데이터가 많을 경우에 성능이 높아지는데, 이는 모델 학습에 필요한 샘플 데이터가 많아질수록 GPR가 더욱 높은 성능을 보이기 때문
- 데이터가 많은 경우나 저차원 공간에서는 SkoptSampler가 적합

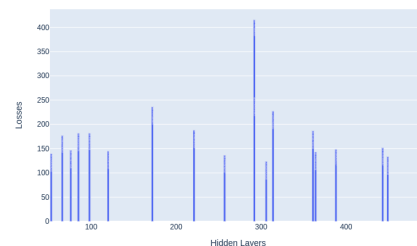
Histogram of Grid Search (Min Loss: 114, Hidden Layer: 475)



Histogram of Gaussian Process (Min Loss: 117, Hidden Layer: 285)



Histogram of Tree Structured Parzen Window Estimation (Min Loss: 123, Hidden Layer: 306.0)



최선의 결과가 나온 곳 주위에서 더 좋은 결과를 기대하고 탐색을 진행함을 알 수 있음

▼ 예시 코드) optuna를 활용한 SVM 하이퍼 파라미터 튜닝해보기

```
import optuna
from sklearn.datasets import load_iris
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split

# 데이터 로드
X, y = load_iris(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# 목적 함수 정의: SVM 모델을 생성하고 학습한 뒤, 검증 데이터에 대한 정확도를 반환
def objective(trial):
    # SVM 모델 생성
    clf = SVC(C=trial.suggest_loguniform('C', 1e-3, 1e3),
              gamma=trial.suggest_loguniform('gamma', 1e-3, 1e3))
```

```

# 모델 학습
clf.fit(X_train, y_train)

# 검증 데이터에 대한 정확도 반환
return clf.score(X_test, y_test)

# 최적화 실행
## optuna.create_study: 최적화를 위한 study 객체를 생성
study = optuna.create_study(direction='maximize') # maximize: 정확도가 높을수록 좋음
## study.optimize: 최적화를 실행
study.optimize(objective, n_trials=100) ## n_trials: 조합의 수 지


# 최적 하이퍼파라미터 출력
## study.best_trial: 최적의 하이퍼파라미터와 해당 하이퍼파라미터로 모델을 학습한 결과를 확인
print('Best trial: score {}, params {}'.format(study.best_trial.value, study.best_trial.params))

```

참고자료

HyperParameter Optimization Algorithm (feat. GP / TPE)

Algorithms for Hyper-Parameter Optimization

 <https://hoonst.github.io/2020/11/15/Algorithms-for-Hyperparameter-Optimization/>


```

BO( $f, M_0, T, S$ )
 $\mathcal{H} \leftarrow \emptyset$ 
For  $t \leftarrow 1$  to  $T$ 
   $x^* \leftarrow \operatorname{argmin}_x S(x, M_{t-1})$ 
  Evaluate  $f(x^*)$ ,  $\triangleright$  Expensive step
   $\mathcal{H} \leftarrow \mathcal{H} \cup (x^*, f(x^*))$ 
  Fit a new model  $M_t$  to  $\mathcal{H}$ 
return  $\mathcal{H}$ 

```


Figure 1: The pseudo-code of generic Sequential Model-Based Optimization

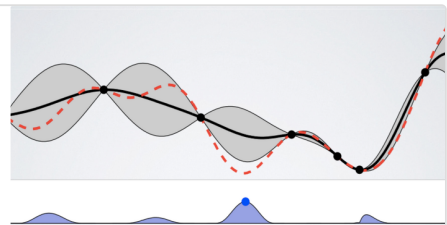
Bayesian Optimization - Jay's Blog

 <https://otzslayer.github.io/ml/2022/12/03/bayesian-optimization.html>

[ML] 베이지안 최적화 (Bayesian Optimization)

Hyperparameter Optimization이란, 학습을 수행하기 위해 사전에 설정해야 하는 값인 hyperparameter(하이퍼파라미터)의 최적값을 탐색하는 문제를 지칭합니다. 보통 Hyperparameter를 찾기 위해 사용되는 방법으로는 Manual Search,

 <https://woono.tistory.com/102>



<https://s3-us-west-2.amazonaws.com/secure.notion-static.com/040b050b-95c4-4beb-a9e1-ba9516745117/NIPS-2011-algorithms-for-hyper-parameter-optimization-Paper.pdf>

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/ae1db1af-372a-410d-915e-908c6890dbb8/A_Tutorial_on_Bayesian_Optimization.pdf

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/9b7c3263-57f0-4edd-a17e-fa0646a66f8a/Automated_machine_learning.pdf

Gaussian Process