**Intro to Model Tuning: Grid and Random Search**

최적의 하이퍼파라미터는 학습데이터에 따라 달라진다. 따라서 각 문제에 맞게 조정해야 한다. 하이퍼파라미터 조정에는 몇 가지 방법이 있다.

1. 매뉴얼: 직관, 경험, 추측을 기바능로 하이퍼 파라미터를 선택하고 하이퍼 파라미터로 모델을 교육한 후 검증 데이터에 점수를 매긴다. 인내심이 바닥나거나 결과에 만족할 때까지 과정을 반복한다.
2. 그리드 서치: 하이퍼파라미터 값의 그리드를 설정하고 각 조합에 대해 모델을 교육하고 검증 데이터에 대한 점수를 매긴다. 이 접근 방식에서는 모든 하이퍼 파라미터 값의 조합을 시도하지만 매우 비효율적일 수 있다.
3. 랜덤 서치: 하이퍼파라미터 ㄱ밧의 그리드를 설정하고 랜덤 조합을 선택하여 모델과 점수를 학습한다. 검색 반복 횟수는 시간, 리소스에 따라 설정된다.
4. 자동 하이퍼파라미터 조정: 그레디언트 강하, 베이지안 최적화 또는 진화 알고리즘과 같은 방법을 사용하여 최상의 하이퍼파라미터에 대한 검색을 수행한다.

이 노트에서는 graident boosting모델에 대한 접근법을 그리드 서치와 랜덤 서치를 사용해서 구현할 것이다. 향후 노트북에서는 베이지안 최적화, 특히 Hyperopt 라이브러리를 사용하여 자동화된 하이퍼 파라미터 튜닝을 구현할 예정이다.

**Gradient Boosting**

Gbm은 최근 최고의 기계 학습 모델 중 하나로 떠오르고 있다. Gbm은 정보가 행과 열에 있는 구조화된 데이터와 수백만 개의 관측치가 있는 중간 크기의 데이터 세트에 매우 효과적이다. 이 모델은 현재 캐글의 대부분의 경쟁에서 최고의 성능을 발휘하는 방법이고 성능이 하이퍼 매개 변수 선택에 크게 의존하기 때문에 우리는 이 모델에 초점을 맞출 것이다. Gbm에 대해 당신이 알아야 할 기본은 거의 항상 많은 개별 학습자를 훈련시켜 작동하는 앙상블 방식이라는 것이다. 그러나 나무가 병렬로 훈련되는 랜덤 포레스트와 달리 gbm에서는 각 트리가 이전 트리의 실수로부터 학습하면서 순차적으로 훈련된다. 수백 또는 수천 명의 약한 학습자가 겷삽되어 경사하강법을 사용하여 교육 중에 학습한 각 개인의 기여도를 바탕으로 하나의 강력한 앙상블 학습자가 된다.

gbm에는 전체 앙상블과 개별 의사결정트리를 제어하는 많은 하이퍼 파라미터가 있다. 하이퍼 파라미터 간에는 복잡한 상호 작용이 있기 때문에 이론만으로 어떤 하이퍼 파라미터 조합이 가장 적합한지 알기는 어렵다. 최적의 하이퍼 파라미터 값을 찾는 유일한 방법은 데이터 세트에서 다양한 조합을 시도하는 것이다.

우리는 lgbm을 사용해서 gbm을 구현할 것이다. 이 방법은 사이킷런에서 사용할 수 있는 것보다 훨씬 빠르다(일부는 더 정확하다고 말한다).

여기서 사용할 데이터는 1만개의 행으로 구성된 데이터의 하위 집합으로 작업할 것이다. 하이퍼파라미터 조정은 계산 비용이 매우 많이 들고 캐글 커널에서 전체 데이터 세트를 사용하는 작업은 몇 번 이상의 검색 반복으로는 가능하지 않다. 그러나 여기서 구현할 동일한 아이디어를 전체 데이터 세트에 적용할 수 있으며 이 노트는 gbm을 대상으로 하지만, 모든 머신러닝 모델에 동일하게 적용할 수 있다.

튜닝 결과를 테스트하기 위해 교육 데이터 중 일부인 6천 행을 별도의 테스트 세트로 저장할 것이다. 하이퍼 파라미터 튜닝을 수행할 때 테스트 데이터의 하이퍼 파라미터를 조정하지 않는 것이 중요하다. 우리는 검증 데이터에 대해 조정된 최종 모델을 평가할 때 테스트 데이터를 한 번만 사용할 수 있다. 이 노트에서 실제로 우리의 방법을 테스트하려면 모든 교육 데이터에 대해 최상의 모델을 교육하고 실제 테스트 데이터를 예측한 다음 제출해야 한다.

*# Data manipulation*

import pandas as pd

import numpy as np

*# Modeling*

import lightgbm as lgb

*# Splitting data*

from sklearn.model\_selection import train\_test\_split

N\_FOLDS = 5

MAX\_EVALS = 5

features = pd.read\_csv('../input/home-credit-default-risk/application\_train.csv')

*# Sample 16000 rows (10000 for training, 6000 for testing)*

features = features.sample(n = 16000, random\_state = 42)

*# Only numeric features*

features = features.select\_dtypes('number')

*# Extract the labels*

labels = np.array(features['TARGET'].astype(np.int32)).reshape((-1, ))

features = features.drop(columns = ['TARGET', 'SK\_ID\_CURR'])

*# Split into training and testing data*

train\_features, test\_features, train\_labels, test\_labels = train\_test\_split(features, labels, test\_size = 6000, random\_state = 50)

제공된 훈련 데이터를 자체적으로 훈련데이터와 테스트데이터로 나눈다.

**교차검증**

각 하이퍼파라미터 값의 조합을 평가하려면 검증 세트에서 점수를 매겨야 한다. 테스트 데이터에서 하이퍼 파라미터를 조정할 수 없다. 우리는 최종 모델을 평가할 때 테스트 데이터를 한 번만 사용할 수 있다. 테스트 데이터는 실제 데이터에 배치될 때 모델 성능의 추정치 역할을 하도록 되어 있으므로 실제 성능에 대한 공정한 추정치가 제공되지 않기 때문에 테스트 데이터에 모델을 최적화하면 안된다. 따라서 검증 세트를 사용하는 것이 올바른 방법이다. 그러나 중요한 교육 데이터를 별도의 교육 및 검증 세트로 분할하는 대신 Kfold 교차 검증을 사용한다. 교육 데이터를 보전하는 것 외에도 단일 검증 세트를 사용하는 것보다 테스트 세트의 일반화 성능을 더 잘 추정할 수 있따. 각 하이퍼 파라미터 세트의 성능은 교차 검증을 통해 곡선 아래의 영역(ROC\_AUC)에 의해 결정된다.

이 예에서는 성능을 평가하기 위해 각 하이퍼 파라미터 값 집합을 사용하여 모형을 5회 교육하고 테스트하는 5배 교차 검증을 사용한다. 하이퍼 파라미터 튜닝에 시간이 많이 걸리는 이유 중 하나는 교차 검증을 사용하기 때문이다. 교육 세트가 충분히 크면 별도의 단일 검증 세트를 사용하는 것으로도 해결할 수 있지만 교차 검증이 과적합을 방지하는 더 안전한 방법이다.

Kfold 교차 검증을 구현하기 위해 lightgbm 교차 검증 기능 (cv)를 사용할 것이다. 조기 정지를 사용할 필요가 없는 다른 머신러닝 모델의 경우 사이킷런의 GridSearchCV나 RandomSearchCV를 사용할 수 있다.

**Early stopping**

gbm에서 가장 중요한 하이퍼 파라미터 중 하나는 추정기 수이다. 우리는 이것을 우리의 검색에서 또 다른 하이퍼 파라미터로 설정할 수 있지만, 더 좋은 방법이 있다. 조기중지는 지정된 반복 횟수 동안 유효성 검사 오류가 줄어들지 않을 때까지 훈련하는 것을 의미한다. Gbm의 경우 더 많은 의사 결정 트리를 교육하는 것을 의미하며, 이 예에서는 100라운드로 조기 중지를 사용하므로 검증 오류가 100라운드로 줄어들지 않을 때까지 교육이 계속된다. 그런 다음 검증 데이터에서 가장 높은 점수르 ㄹ얻은 추정치의 수가 최종 모형에 사용할 추정기의 수로 선택된다.

조기 정지 개념은 일반적으로 gbm과 심층 신경망에 적용되므로 이해하기 좋은 기술이다. 이것은 교육 데이터에 과적합하지 않음으로써 검정 세트의 일반화 성능을 향상시키는 것을 목표로 하는 많은 형태의 정규화 중 하나이다. 추정기를 계속 추가하면 모델의 용량이 증가하기 때문에 교육 오차는 항상 줄어들 것이다. 비록 이것이 긍정적으로 보일 수 있지만, 이것은 모형이 훈련 데이터를 암기하기 시작하고 나서 새로운 시험 데이터에 대해 잘 수행되지 않을 것이라는 것을 의미한다. 모형이 교육 데이터에 너무 많이 의존하기 시작하므로 추정기를 계속 추가할수록 모형의 분산이 증가한다.

초기 중지는 교차 검증 기능에서 lightgbm 라이브러리로 구현하기 쉽다. 조기 정지 횟수만 넘기면 된다. Cv 기능을 사용하려면 먼저 lightgbm 데이터 세트를 만들어야 한다.

*# Create a training and testing dataset*

train\_set = lgb.Dataset(data = train\_features, label = train\_labels)

test\_set = lgb.Dataset(data = test\_features, label = test\_labels)

일련의 하이퍼 파라미터를 교차 검증에 전달해야 하므로 lightgbm의 기본 하이퍼 파라미터를 사용한다. Cv 호출에서 num\_boost\_round는 1만 이상으로 설정되지만 조기 중지를 사용하고 있으므로 이 숫자에 실제로 도달하지 않는다. 우리가 사용하고 있는 지표는 곡선 아래의 수신기 작동 특성 영역(roc\_auc)이다. 아래 코드는 5 kfold 및 100회 조기 정지 라운드로 교차 검증을 수행한다.

*# Get default hyperparameters*

model = lgb.LGBMClassifier()

default\_params = model.get\_params()

*# Remove the number of estimators because we set this to 10000 in the cv call*

del default\_params['n\_estimators']

*# Cross validation with early stopping*

cv\_results = lgb.cv(default\_params, train\_set, num\_boost\_round = 10000, early\_stopping\_rounds = 100,

metrics = 'auc', nfold = N\_FOLDS, seed = 42)

cv\_results는 메트릭 평균 및 메트릭 표준편차에 대한 목록이 있는 사전이다. 마지막 열에는 최고 수행 점수가 포함되어 있다. 사전에 있는 각 목록의 길이는 교육할 추정기의 최적 개수가 된다.

print('The maximum validation ROC AUC was: **{:.5f}** with a standard deviation of **{:.5f}**.'.format(cv\_results['auc-mean'][-1], cv\_results['auc-stdv'][-1]))

print('The optimal number of boosting rounds (estimators) was **{}**.'.format(len(cv\_results['auc-mean'])))

텍스트이(가) 표시된 사진

자동 생성된 설명

우리는 이 결과를 이기기 위한 기준 모델로 사용할 수 있다. 모형이 테스트 데이터에 대해 얼마나 잘 수행하는지 확인하기 위해, 교차 검증과 조기 중지 시 가장 많은 추정기가 발견된 모든 교육 데이터에서 해당 모형을 재교육할 것이다.

from sklearn.metrics import roc\_auc\_score

*# Optimal number of esimators found in cv*

model.n\_estimators = len(cv\_results['auc-mean'])

*# Train and make predicions with model*

model.fit(train\_features, train\_labels)

preds = model.predict\_proba(test\_features)[:, 1]

baseline\_auc = roc\_auc\_score(test\_labels, preds)

print('The baseline model scores **{:.5f}** ROC AUC on the test set.'.format(baseline\_auc))



이것은 하이퍼 파라미터 튜닝 전 기준 점수이다. 기본 모형과 다른 점은 조기 중지를 사용하여 추정기 수를 설정한 것이다.

**하이퍼 파라미터 구현**

이제 기본 프레임워크가 마련되었다. 교차 검증을 사용하여 모델 하이퍼 파라미터의 성능을 결정하고 gbm을 조기에 중지하여 추정기 수를 조정할 필요가 없다. 그리드 및 랜덤 서치의 기본 전략은 간단하다. 각 하이퍼 파라미터 값 조합에 대해 교차 검증 점수를 평가하고 결과를 하이퍼 파라미터와 함께 기록한다. 그런 다음 검색이 끝나면 교차 검증 점수가 가장 높은 하이퍼 파라미터를 선택하고 모든 교육 데이터에 대해 모형을 교육한 다음 테스트 데이터를 예측한다.

**하이퍼파라미터 조정의 네 가지 부분**

하이퍼파라미터 조정은 다음과 같은 네 가지 부분으로 이루어진다고 생각하면 유용하다.

목표 함수: 하이퍼 파라미터를 사용하고 최소화 또는 최대화하려는 점수를 반환하는 함수

도메인: 검색할 하이퍼 파라미터 값의 집합

알고리즘: 목적 함수에서 평가할 다음 하이퍼파라미터의 집합을 선택하는 방법

결과 기록: 각 하이퍼파라미터 집합과 목표 함수의 결과 점수를 포함하는 데이터 구조

그리드에서 랜덤 서치이나 베이지안 최적화로 전환하려면 이 네 가지 부분을 조금만 수정하면 된다.

**목표 함수**

목표 함수는 하이퍼 파라미터를 사용하고 점수를 나타내는 값을 출력한다. 기본 최적화에서는 이 점수를 최소화해야 했지만 여기서는 물론 최대화하고 싶은 roc auc가 된다. 나중에 베이지안 최적화에 도달하면 최소화할 값을 사용해야 하므로 1 – roc auc를 점수로 사용할 수 있다. 목적 함수의 중간에서 발생하는 것은 문제에 따라 다르지만, 이 문제에 대해서는 지정된 모델 하이퍼 파라미터와의 교차 검증을 사용하여 교차 검증 roc auc를 얻을 것이다. 그런 다음 이 점수는 최적의 모형 하이퍼 파라미터 값을 선택하는 데 사용된다.

최대화할 값을 반환하는 것 외에도 목표함수는 하이퍼 파라미터와 검색 반복을 반환한다. 이 결과를 통해 다시 돌아가서 검색 중에 발생한 일을 검사할 수 있다. 아래 코드는 그리드 및 랜덤 서치에 모두 사용할 수 잇는 간단한 목적 함수를 구현한다.

def objective(hyperparameters, iteration):

*"""Objective function for grid and random search. Returns*

*the cross validation score from a set of hyperparameters."""*

*# Number of estimators will be found using early stopping*

if 'n\_estimators' **in** hyperparameters.keys():

del hyperparameters['n\_estimators']

*# Perform n\_folds cross validation*

cv\_results = lgb.cv(hyperparameters, train\_set, num\_boost\_round = 10000, nfold = N\_FOLDS,

early\_stopping\_rounds = 100, metrics = 'auc', seed = 42)

*# results to retun*

score = cv\_results['auc-mean'][-1]

estimators = len(cv\_results['auc-mean'])

hyperparameters['n\_estimators'] = estimators

return [score, hyperparameters, iteration]

score, params, iteration = objective(default\_params, 1)

print('The cross-validation ROC AUC was **{:.5f}**.'.format(score))



**도메인**

도메인 또는 검색 공간은 검색하려는 모든 하이퍼 파라미터의 가능한 값이다. 랜덤 및 그리드 서치의 경우 도메인은 하이퍼 파라미터 그리드이며 일반적으로 각 하이퍼 파라미터에 대한 값 목록과 하이퍼 파라미터 키가 있는 사전의 형태를 취한다.

Gbm의 하이퍼 파라미터

어떤 설정을 조정할 수 있는지 보려면 모델을 만들어 인쇄해보자. 모든 하이퍼 파라미터에 대한 설명은 LightGBM 설명서를 참조할 수 있다.  [LightGBM documentation](http://lightgbm.readthedocs.io/en/latest/Parameters.html)

*# Create a default model*

model = lgb.LGBMModel()

model.get\_params()

텍스트이(가) 표시된 사진

자동 생성된 설명

Silent, objective, random\_state, n\_jobs와 같은 일부 항목은 조정할 필요가 없으며, 조기 중지를 사용하여 가장 중요한 하이퍼 파라미터인 n\_estimator를 결정한다. 예를 들어 min\_child\_samples 및 min\_child\_weight 모두 최소 리프 관찰 요구 사항을 조정하여 개별 의사 결정 트리의 복잡성을 제한하므로 하이퍼 파라미터 중 일부는 조정할 필요가 없다. 그러나 최적화 해야 할 하이퍼 파라미터가 여전히 많으므로 10개를 선택하여 튜닝할 것이다.

하이퍼파라미터 그리드를 선택하는 것은 아마도 하이퍼파라미터 조정에서 가장 어려운 부분일 것이다. 어떤 하이퍼파라미터 값이 잘 작동하고 데이터세트에 따라 최적의 설정이 결정될지 미리 말하는 것은 거의 불가능하다. 또한 하이퍼 파라미터 간의 상호 작용이 복잡하기 때문에 다른 하이퍼 파라미터를 변경하기 시작하면 방금 조정한 하이퍼 파라미터에 영향을 미치기 때문에 한 번에 하나씩 튜닝하는 것만으로는 작동하지 않는다.

이전에 모델을 사용해본 경험이 있다면 하이퍼 파라미터의 최적 값이 일반적으로 어디에 있는지 또는 검색 공간이 얼마나 좋은지 알 수 있다. 그러나 경험이 많지 않다면 단순히 넓은 검색 공간을 정의하고 그 어딘가에 최상의 값이 있기를 바랄 수 있다. 일반적으로 메소드를 처음 사용할 때는 기본 값을 중심으로 넒은 검색 공간을 정의한다. 그런 다음 일부 하이퍼 파라미터가 더 잘 작동하는 경향이 있는 경우 해당 값을 중심으로 검색을 집중할 수 있다.

10개의 하이퍼 파라미터에 대한 전체 그리드가 아래에 정의되어 있다. 사전의 각 값은 목록이어야 하므로 범위, np.linespace 및 np.logspace와 결합된 목록을 사용하여 각 하이퍼 파라미터에 대한 값 범위를 정의한다.

*# Hyperparameter grid*

param\_grid = {

'boosting\_type': ['gbdt', 'goss', 'dart'],

'num\_leaves': list(range(20, 150)),

'learning\_rate': list(np.logspace(np.log10(0.005), np.log10(0.5), base = 10, num = 1000)),

'subsample\_for\_bin': list(range(20000, 300000, 20000)),

'min\_child\_samples': list(range(20, 500, 5)),

'reg\_alpha': list(np.linspace(0, 1)),

'reg\_lambda': list(np.linspace(0, 1)),

'colsample\_bytree': list(np.linspace(0.6, 1, 10)),

'subsample': list(np.linspace(0.5, 1, 100)),

'is\_unbalance': [True, False]

}

유의해야 할 한 가지 측면은 boosting\_type이 goss이면 하위 샘플을 사용할 수 없다는 것이다. 따라서 boost\_type=goss일 경우 하위 샘플을 1.0으로 설정하는 논리 라인이 알고리즘에 필요하다. 아래의 예에서 하이퍼 파라미터 세트를 임의로 선택하고 부스팅 유형이 goss인 경우 하위 샘플을 1.0으로 설정한다.

import random

random.seed(50)

*# Randomly sample a boosting type*

boosting\_type = random.sample(param\_grid['boosting\_type'], 1)[0]

*# Set subsample depending on boosting type*

subsample = 1.0 if boosting\_type == 'goss' else random.sample(param\_grid['subsample'], 1)[0]

print('Boosting type: ', boosting\_type)

print('Subsample ratio: ', subsample)

boosting\_type 및 is\_unbalance 도메인은 범주형 변수이므로 매우 간단하다. 정수(num\_leaves, min\_child\_samples)여야 하는 하이퍼 파라미터의 경우 단계별로 시작부터 중지까지의 범위를 반환한다. Range는 항상 정수를 반환한다. 즉 분수가 될 수 있는 일정한 간격의 값을 사용하려면 np.linespace를 사용해야 한다.

마지막으로 np.logspace는 로그 스케일에 일정한 간격으로 값을 반환한다. ‘선형 공간에서는 시퀀스가 basestart에서 시작하여 basestop으로 끝난다’ 라는 문서에 따르면 이는 학습 속도와 같이 몇 자릿수 이상 차이가 나는 값에 유용하다.

import matplotlib.pyplot as plt

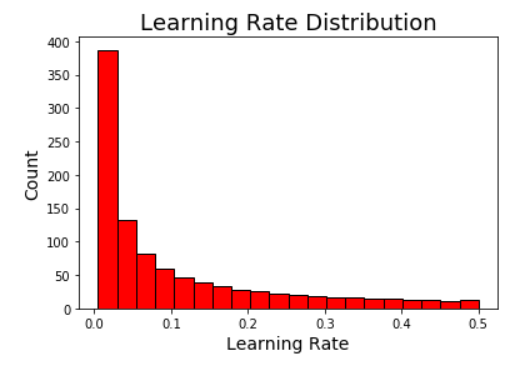
import seaborn as sns

%matplotlib inline

*# Learning rate histogram*

plt.hist(param\_grid['learning\_rate'], bins = 20, color = 'r', edgecolor = 'k');

plt.xlabel('Learning Rate', size = 14); plt.ylabel('Count', size = 14); plt.title('Learning Rate Distribution', size = 18);



**학습 속도 도메인**

학습 속도 도메인은 0.005부터 0.5까지이다. 로그 균등 분포를 사용하면 0.05에서 0.5까지의 값만큼 0.005에서 0.05까지의 도메인을 만들 수 있다. 선형 공간에서는 이 값이 더 큰 거리를 나타내므로 0.05에서 0.5까지의 값이 훨씬 더 많겠지만 로그 공간에서는 이 두 구간이 각각 10의 배수이기 때문에 폭이 같다. 로그 척도의 경우 이러한 구간은 크기가 동일하지만 선형 척도의 경우 후자의 크기가 전자의 10배이다. 다시 말해, 로그의 균등 분포는 우리가 몇 개의 크기에 따라 변화하는 도메인에서 더 고르게 표본을 추출할 수 있게 해준다.

이게 좀 헷갈리면 위의 그래프로 더 명확하게 알 수 있을 것 같다. 또한 각 간격의 값 수를 세어 간격이 정확한지 확인하기 위해 온전성 검사를 할 수 있다.

a = 0

b = 0

*# Check number of values in each category*

for x **in** param\_grid['learning\_rate']:

*# Check values*

if x >= 0.005 **and** x < 0.05:

a += 1

elif x >= 0.05 **and** x < 0.5:

b += 1

print('There are **{}** values between 0.005 and 0.05'.format(a))

print('There are **{}** values between 0.05 and 0.5'.format(b))

텍스트, 화면, 오렌지, 설정이(가) 표시된 사진

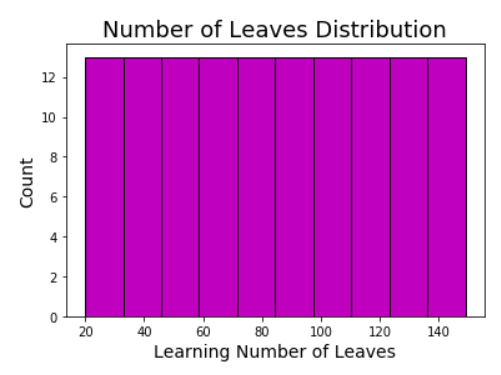
자동 생성된 설명

단순 도메인의 예로서 num\_leaves는 균일한 분포이다. 즉, 선형 척도로 값이 일정한 간격으로 배치된다.

*# number of leaves domain*

plt.hist(param\_grid['num\_leaves'], color = 'm', edgecolor = 'k')

plt.xlabel('Learning Number of Leaves', size = 14); plt.ylabel('Count', size = 14); plt.title('Number of Leaves Distribution', size = 18);



**다음 값을 선택하는 알고리즘**

우리는 일반적으로 그리드를 그렇게 생각하지 않지만, 그리드와 랜덤 서치는 모두 알고리즘이다. 그리드 서치의 경우 도메인을 입력하고 알고리즘이 순서에 따라 각 하이퍼 파라미터에 대한 다음 값을 선택한다. 그리드 서치의 유일한 요구 사항은 그리드의 모든 조합을 한 번만 시도해야 한다는 것이다. 랜덤 서치를 위해 도메인을 입력하고 알고리즘이 시도할 하이퍼 파라미터 값의 임의 조합을 제공한다. 랜덤 서치에는 다음 값이 무작위로 선택된다는 것 외에는 요구 사항이 없다.

하이퍼 파라미터 튜닝의 마지막 부분을 다루는 즉시 이러한 알고리즘을 구현할 것이다.

**결과**

결과 내역은 목적 함수에 대한 하이퍼 파라미터 조합과 결과 점수를 포함하는 데이터 구조이다. 베이지안 최적화에 도달하면, 모형은 실제로 과거의 결과를 사용하여 평가할 다음 하이퍼파라미터를 결정한다. 랜덤 및 그리드 서치는 과거 기록을 사용하지 않는 비정보적인 방법이지만, 어떤 하이퍼 파라미터가 가장 잘 작동했는지 확인할 수 있는 기록이 여전히 필요하다.

*# Dataframes for random and grid search*

random\_results = pd.DataFrame(columns = ['score', 'params', 'iteration'],

index = list(range(MAX\_EVALS)))

grid\_results = pd.DataFrame(columns = ['score', 'params', 'iteration'],

index = list(range(MAX\_EVALS)))

**그리드 서치 구현**

그리드 서치는 극단적인 추측과 확인으로 가장 잘 설명된다. 문제가 있다. 최상의 교차 검증 점수를 산출하는 하이퍼 파라미터와 하이퍼 파라미터 그리드에서 시도할 값 집합을 찾는다. 답을 찾기 위한 그리드 서치는 도메인에서 모든 값의 조합을 시도하고 최적의 조합이 그리드에 있기를 바라는 것이다(실제로는 무한히 많은 시간이 소요되는 하이퍼 파라미터 그리드가 없으면 최적의 설정을 찾았는지 알 수 없다).

그리드 서치에는 한 가지 제한 문제가 있다. 즉, 그리드의 모든 하이퍼 파라미터 조합에 대해 교차 검증을 수행해야 하기 때문에 계산 비용이 매우 많이 든다. 우리가 개발한 간단한 작은 그리드에 총 몇 개의 하이퍼 파라미터 설정이 있는지 알아보겠다.

com = 1

for x **in** param\_grid.values():

com \*= len(x)

print('There are **{}** combinations'.format(com))



캐글이 커널을 양자 컴퓨터로 업그레이드하기 전까지, 우리는 위 조합의 일부를 실행할 수 없을 것이다. 평가 당 100초가 소요된다고 가정해보겠다.

print('This would take **{:.0f}** years to finish.'.format((100 \* com) / (60 \* 60 \* 24 \* 365)))



좀 더 나은 접근이 필요할 것 같다. 대안을 논의하기 전에 이 그리드를 실제로 어떻게 사용하고 모든 하이퍼 파라미터를 평가하는지에 대해 살펴보겠다.

아래 코드는 그리드 서치를 위한 알고리즘을 보여준다. 먼저 라인 키, 값= zip\_param\_grid.grid.grad를 사용하여 하이퍼 파라미터 그리드에서 값의 압축을 푼다. 핵심 라인은 itertolls.product(\*values)의 v에 대한 것이며, 여기서 하이퍼 파라미터 그리드에서 가능한 모든 값의 조합을 한 번에 하나씩 반복한다. 각 값 조합에 대해 사전 하이퍼 파라미터=dict(zip(keys, v))를 만든 다음 앞에서 정의한 목표 함수에 전달한다. 목표 함수는 데이터 프레임에 기록하는 하이퍼 파라미터에서 교차 검증 점수를 반환한다. 이 과정은 각 및 모든 하이퍼 파라미터 조합에 대해 반보고딘다. Itertools.product를 사용하면 메모리에 보관하기에는 너무 큰 모든 가능한 조합 목록을 할당하는 대신 생성기를 만들 수 있다.

import itertools

def grid\_search(param\_grid, max\_evals = MAX\_EVALS):

*"""Grid search algorithm (with limit on max evals)"""*

*# Dataframe to store results*

results = pd.DataFrame(columns = ['score', 'params', 'iteration'],

index = list(range(MAX\_EVALS)))

*# https://codereview.stackexchange.com/questions/171173/list-all-possible-permutations-from-a-python-dictionary-of-lists*

keys, values = zip(\*param\_grid.items())

i = 0

*# Iterate through every possible combination of hyperparameters*

for v **in** itertools.product(\*values):

*# Create a hyperparameter dictionary*

hyperparameters = dict(zip(keys, v))

*# Set the subsample ratio accounting for boosting type*

hyperparameters['subsample'] = 1.0 if hyperparameters['boosting\_type'] == 'goss' else hyperparameters['subsample']

*# Evalute the hyperparameters*

eval\_results = objective(hyperparameters, i)

results.loc[i, :] = eval\_results

i += 1

*# Normally would not limit iterations*

if i > MAX\_EVALS:

break

*# Sort with best score on top*

results.sort\_values('score', ascending = False, inplace = True)

results.reset\_index(inplace = True)

return results

일반적으로 그리드 서치에서는 평가 횟수를 제한하지 않는다. 평가 횟수는 하이퍼 파라미터 그리드의 총 조합에 의해 설정된다. 따라서 위 코드에서 MAX\_EVALS는 실제로 사용하지 않는다. 여기서는 간단한 테스트를 위해 MAX\_EVALS를 5로 제한한다.

grid\_results = grid\_search(param\_grid)

print('The best validation score was **{:.5f}**'.format(grid\_results.loc[0, 'score']))

print('**\n**The best hyperparameters were:')

import pprint

pprint.pprint(grid\_results.loc[0, 'params'])

텍스트이(가) 표시된 사진

자동 생성된 설명

이제 최고의 하이퍼 파라미터를 보유하고 있으므로 실제 테스트 데이터가 아닌 우리가 만든 테스트 데이터에서 하이퍼 파라미터를 평가할 수 있다.

*# Get the best parameters*

grid\_search\_params = grid\_results.loc[0, 'params']

*# Create, train, test model*

model = lgb.LGBMClassifier(\*\*grid\_search\_params, random\_state=42)

model.fit(train\_features, train\_labels)

preds = model.predict\_proba(test\_features)[:, 1]

print('The best model from grid search scores **{:.5f}** ROC AUC on the test set.'.format(roc\_auc\_score(test\_labels, preds)))

텍스트이(가) 표시된 사진

자동 생성된 설명

교차 검증에서보다 테스트 세트에서 모형이 더 좋은 점수를 받는 것이 흥미롭다. 모형이 검증 데이터에 맞춰져 있기 때문에 일반적으로 그 반대인 경우가 많다. 이 경우 테스트 데이터의 크기가 작기 때문에 성능이 향상될 수 있으며, 실제 경쟁 데이터로 변환되지는 않지만 운이 매우 좋다.

그리드 서치의 작동 방식을 파악하기 위해 평가된 하이퍼 파라미터의 진행 과정을 살펴볼 수 있다.

pd.options.display.max\_colwidth = 1000

grid\_results['params'].values

텍스트, 실외, 명판, 스크린샷이(가) 표시된 사진

자동 생성된 설명

서브 샘플과 is\_unbalance를 살펴보자. 이러한 매개변수만 변경되기 때문이다. 실제로 이러한 값 변경의 효과는 너무 작아서 검증 점수는 말 그대로 런 전체에 걸쳐 변경되지 않았다(이렇게 작은 변경은 모형에 아무런 영향을 미치지 않음을 나타냄).

**응용 프로그램**

전체 데이터 세트에서 이 기능을 실행하려면 이러한 함수를 사용하여 스크립트에 넣으면 된다. 그러나 하이퍼 파라미터 그리드가 매우 작지 않은 경우 그리드 서치를 사용하지 않는 것이 좋다. 이는 철저한 방법이기 대문이다. 나중에 위에서 사용한 것과 동일한 데이터 하위 집합에 대해 1000번의 그리드 반복 및 랜덤 서치 결과를 살펴보겠다. 전체 데이터에 대해 어떤 형식의 그리드 서치도 실행하려고 시도하지 않았으며 이 방법은 시도하지 않을 것이다.

**랜덤 서치**

무작위 검색은 그리드 검색에 비해 놀라울 정도로 효율적이다. 그리드 서치는 하이퍼 파라미터의 최적값을 결국 찾겠지만, 무작위 검색은 일반적으로 훨씬 적은 반복으로 충분히 값을 찾는다. [This great paper explains why this is so](http://www.jmlr.org/papers/volume13/bergstra12a/bergstra12a.pdf): 이 훌륭한 노문은 그리드 서치가 모든 조합을 평가해야 하기 때문에 하이퍼 파라미터 검색 공간의 정제되지 않은 영역을 평가하는데 너무 많은 시간을 소비하는 이유를 설명한다. 반대로 랜덤 서치는 검색 공간을 더 잘 탐색하므로 일반적으로 훨씬 적은 반복으로 적절한 하이퍼 파라미터 조합을 찾을 수 있다.

이 기사에서 설명한 바와 같이, 무작위 검색은 그 효과 때문에 최초로 시도되는 하이퍼 파라미터 최적화 방법이 될 것이다. 비록 정보가 없는 방법이지만 임의 검색은 기본값보다 더 나은 값을 찾을 수 있으며 실행이 간단하다.

랜덤 서치는 알고리즘으로도 생각할 수 있다. 그리드에서 다음 하이퍼 파라미터 세트를 임의로 선택한다. 다음과 같이 각 하이퍼 파라미터에 대해 하나의 무작위 값을 선택하여 하이퍼 파라미터 사전을 만들 수 있다.

random.seed(50)

*# Randomly sample from dictionary*

random\_params = {k: random.sample(v, 1)[0] for k, v **in** param\_grid.items()}

*# Deal with subsample ratio*

random\_params['subsample'] = 1.0 if random\_params['boosting\_type'] == 'goss' else random\_params['subsample']

random\_params

텍스트이(가) 표시된 사진

자동 생성된 설명

다음으로 랜덤 서치 함수를 정의한다. 이것은 다음 하이퍼파라미터 값을 선택하는 데 사용되는 방법을 제외하고 그리드 서치와 동일한 일반 구조를 사용한다. 또한 임의 검색은 항상 검색 반복 횟수에 제한을 두고 실행된다.

def random\_search(param\_grid, max\_evals = MAX\_EVALS):

*"""Random search for hyperparameter optimization"""*

*# Dataframe for results*

results = pd.DataFrame(columns = ['score', 'params', 'iteration'],

index = list(range(MAX\_EVALS)))

*# Keep searching until reach max evaluations*

for i **in** range(MAX\_EVALS):

*# Choose random hyperparameters*

hyperparameters = {k: random.sample(v, 1)[0] for k, v **in** param\_grid.items()}

hyperparameters['subsample'] = 1.0 if hyperparameters['boosting\_type'] == 'goss' else hyperparameters['subsample']

*# Evaluate randomly selected hyperparameters*

eval\_results = objective(hyperparameters, i)

results.loc[i, :] = eval\_results

*# Sort with best score on top*

results.sort\_values('score', ascending = False, inplace = True)

results.reset\_index(inplace = True)

return results

random\_results = random\_search(param\_grid)

print('The best validation score was **{:.5f}**'.format(random\_results.loc[0, 'score']))

print('**\n**The best hyperparameters were:')

import pprint

pprint.pprint(random\_results.loc[0, 'params'])

텍스트이(가) 표시된 사진

자동 생성된 설명

또한 테스트 데이터에 대한 최적의 랜덤 서치 모델을 평가할 수 있다.

*# Get the best parameters*

random\_search\_params = random\_results.loc[0, 'params']

*# Create, train, test model*

model = lgb.LGBMClassifier(\*\*random\_search\_params, random\_state = 42)

model.fit(train\_features, train\_labels)

preds = model.predict\_proba(test\_features)[:, 1]

print('The best model from random search scores **{:.5f}** ROC AUC on the test set.'.format(roc\_auc\_score(test\_labels, preds)))

텍스트이(가) 표시된 사진

자동 생성된 설명

마지막으로, 하이퍼 파라미터의 랜덤 서치 시퀀스를 볼 수 있다.

random\_results['params']

이번에는 거의 무작위로 선택한 것처럼 곳곳에 있는 하이퍼 파라미터 값을 볼 수 있다. 무작위 검색은 검색 도메인을 탐색하는 그리드 검색보다 훨씬 더 좋다. 하이퍼 파라미터를 평가할 시간이 제한된 경우 이러한 이유로 그리드 서치보다 랜덤 서치가 더 나은 옵션이다.

**랜덤 및 그리드 검색 스태킹**

하이퍼 파라미터 조정을 보다 현명하게 구현하기 위한 한 가지 옵션은 랜덤 서치와 그리드 서치를 결합하는 것이다.

큰 하이퍼 파라미터 그리드와 함께 랜덤 서치를 사용한다. 랜덤 서치를 사용하여 최상의 성능을 발휘하는 하이퍼파라미터 값 주위에 초점 하이퍼파라미터 그리드를 구축할 수 있다. 축소된 하이퍼파라미터 그리드에서 그리드 검색을 실행한다. 최대 계산/시간 예산이 초과될 때까지 더 집중된 그리드에 대해 그리드 검색을 반복한다. 이후 노트북에서는 과거 평가 결과를 사용하여 목적 함수에서 시도할 다음 하이퍼 파라미터 값을 선택하는 방법을 살펴보겠다. 이러한 방법(베이지안 최적화 포함)은 기본적으로 위에서 설명한 전략에서 수행하는 작업이다. 이전 결과에서 검색을 시도한 다음 값을 조정한다. 이러한 정보에 입각한 방법의 전체적인 목표는 과거의 평가 결과를 바탕으로 시도할 다음 값을 추론하여 목적 함수의 평가를 제한하는 것이다. 따라서 이러한 알고리즘은 보다 유망한 하이퍼 파라미터 값을 평가하여 시간을 절약할 수 있다. 이것은 정말 멋진 주제이며 베이지안 최적화는 매우 흥미로운 일이다.

**다음 단계**

이제 이러한 랜덤 및 그리드 서치 기능을 사용하여 전체 데이터셋 또는 선택한 데이터셋에 사용할 수 있다. 이러한 검색 방법은 비용이 많이 들기 때문에 하이퍼 파라미터 조정에 시간이 걸릴 것으로 예상된다.

지금은 결과를 비교하기 위해 데이터셋에서 1000회 동안 랜덤 및 그리드 검색을 구현하는 것으로 전환하겠다. 결과는 이 커널에 있는 데이터의 일부로 제공된다.

**진행률을 모니터링하기 위해 파일에 쓰기**

이러한 검색을 장시간 실행할 경우 검색이 진행되는 동안 성능을 추적하는 것이 당연하다. 명령 프롬프트에 정보를 인쇄할 수 있지만 1000번 반복하면 더 복잡해지고 명령 프롬프트를 닫으면 결과가 사라진다. 더 나은 해결책은 각 반복 시 csv 파일에 줄을 쓰는 것이다. 그런 다음 파일을 확인하여 검색이 실행되는 동안 진행 상황을 추적하고 검색이 완료되면 전체 결과를 저장할 수 있다.

**파일 확인에 대한 매우 중요한 참고 사항**

Csv 파일을 확인하고 싶을 때, 검색이 진행 중인 동안에는 엑셀에서 열지 않는다. 그러면 Python에서 권한 오류가 발생하고 검색이 종료된다. 대신 bash에서 tail out\_file.csv를 입력하여 파일의 끝을 볼 수 있다. 여기서 out\_file.csv는 쓰려는 파일의 이름이다. 또한 검색이 수행되는 동안 결과를 안전하게 열 수 있는 메모장 또는 서브라임텍스트와 같은 일부 텍스트 편집기도 있다. 그러나 파이썬으로 기록 중인 파일을 열 때 엑셀을 사용하면 안된다.

아래는 우리가 검색 전에 실행해야 할 코드이다. 이렇게 하면 csv 파일이 생성되고 연결이 열리며 헤더가 기록되고 연결이 닫힌다. 이렇게 하면 out\_file에 현재 있는 모든 정보를 덮어쓰므로 새 검색을 시작할 때마다 새 파일 이름으로 변경하자.

import csv

*# Create file and open connection*

out\_file = 'random\_search\_trials.csv'

of\_connection = open(out\_file, 'w')

writer = csv.writer(of\_connection)

*# Write column names*

headers = ['score', 'hyperparameters', 'iteration']

writer.writerow(headers)

of\_connection.close()

이제 이 파일에 쓸 때마다 랜덤 서치와 그리드 서치를 약간 수정해야 한다. 이번에는 연결을 열고 추가에 a 옵션을 사용하여 원하는 정보가 포함된 줄을 작성한다. 그런 다음 함수가 다시 호출될 때까지 연결을 닫는다.

def random\_search(param\_grid, out\_file, max\_evals = MAX\_EVALS):

*"""Random search for hyperparameter optimization.*

*Writes result of search to csv file every search iteration."""*

*# Dataframe for results*

results = pd.DataFrame(columns = ['score', 'params', 'iteration'],

index = list(range(MAX\_EVALS)))

for i **in** range(MAX\_EVALS):

*# Choose random hyperparameters*

random\_params = {k: random.sample(v, 1)[0] for k, v **in** param\_grid.items()}

random\_params['subsample'] = 1.0 if random\_params['boosting\_type'] == 'goss' else random\_params['subsample']

*# Evaluate randomly selected hyperparameters*

eval\_results = objective(random\_params, i)

results.loc[i, :] = eval\_results

*# open connection (append option) and write results*

of\_connection = open(out\_file, 'a')

writer = csv.writer(of\_connection)

writer.writerow(eval\_results)

*# make sure to close connection*

of\_connection.close()

*# Sort with best score on top*

results.sort\_values('score', ascending = False, inplace = True)

results.reset\_index(inplace = True)

return results

def grid\_search(param\_grid, out\_file, max\_evals = MAX\_EVALS):

*"""Grid search algorithm (with limit on max evals)*

*Writes result of search to csv file every search iteration."""*

*# Dataframe to store results*

results = pd.DataFrame(columns = ['score', 'params', 'iteration'],

index = list(range(MAX\_EVALS)))

*# https://codereview.stackexchange.com/questions/171173/list-all-possible-permutations-from-a-python-dictionary-of-lists*

keys, values = zip(\*param\_grid.items())

i = 0

*# Iterate through every possible combination of hyperparameters*

for v **in** itertools.product(\*values):

*# Select the hyperparameters*

parameters = dict(zip(keys, v))

*# Set the subsample ratio accounting for boosting type*

parameters['subsample'] = 1.0 if parameters['boosting\_type'] == 'goss' else parameters['subsample']

*# Evalute the hyperparameters*

eval\_results = objective(parameters, i)

results.loc[i, :] = eval\_results

i += 1

*# open connection (append option) and write results*

of\_connection = open(out\_file, 'a')

writer = csv.writer(of\_connection)

writer.writerow(eval\_results)

*# make sure to close connection*

of\_connection.close()

*# Normally would not limit iterations*

if i > MAX\_EVALS:

break

*# Sort with best score on top*

results.sort\_values('score', ascending = False, inplace = True)

results.reset\_index(inplace = True)

return results

이러한 함수를 1000회 반복 실행하려면 아래 셀의 주석 처리를 취소하면 된다.

*# MAX\_EVALS = 1000*

*# # Create file and open connection*

*# out\_file = 'grid\_search\_trials\_1000.csv'*

*# of\_connection = open(out\_file, 'w')*

*# writer = csv.writer(of\_connection)*

*# # Write column names*

*# headers = ['score', 'hyperparameters', 'iteration']*

*# writer.writerow(headers)*

*# of\_connection.close()*

*# grid\_results = grid\_search(param\_grid, out\_file)*

*# # Create file and open connection*

*# out\_file = 'random\_search\_trials\_1000.csv'*

*# of\_connection = open(out\_file, 'w')*

*# writer = csv.writer(of\_connection)*

*# # Write column names*

*# headers = ['score', 'hyperparameters', 'iteration']*

*# writer.writerow(headers)*

*# of\_connection.close()*

*# random\_results = random\_search(param\_grid, out\_file)*

**제한된 데이터에 대한 결과**

축소된 데이터셋에서 위의 기능에 대한 1000번의 검색 반복을 검토할 수 있다. 나중에 전체 데이터셋에 있는 작은 버전의 데이터에 가장 적합한 하이퍼 파라미터를 사용하여 데이터 크기를 30배 늘리면 최상의 하이퍼 파라미터가 변환되는지 확인할 수 있다. 커널에서 1000번의 검색 반복이 실행되지 않았지만 12시간 제한 내에 완료될 수 있다.

어떤 방법이 가장 좋은 결과를 냈는지 알 수 있다.

random\_results = pd.read\_csv('../input/home-credit-model-tuning/random\_search\_trials\_1000.csv')

grid\_results = pd.read\_csv('../input/home-credit-model-tuning/grid\_search\_trials\_1000.csv')

결과를 csv에 저장할 때 어떤 이유로 사전이 문자열로 저장된다. 따라서 ast.literal\_eval 함수를 사용하여 결과를 읽은 후 다시 사전으로 변환해야 한다.

import ast

*# Convert strings to dictionaries*

grid\_results['hyperparameters'] = grid\_results['hyperparameters'].map(ast.literal\_eval)

random\_results['hyperparameters'] = random\_results['hyperparameters'].map(ast.literal\_eval)

이제 하이퍼파라미터 검색 결과를 구문 분석하는 함수를 만들어보자. 이렇게 하면 각 열이 하이퍼 파라미터이고 각 행에 검색 결과가 하나씩 있는 데이터 프레임이 반환된다.

def evaluate(results, name):

*"""Evaluate model on test data using hyperparameters in results*

*Return dataframe of hyperparameters"""*

*# Sort with best values on top*

results = results.sort\_values('score', ascending = False).reset\_index(drop = True)

*# Print out cross validation high score*

print('The highest cross validation score from **{}** was **{:.5f}** found on iteration **{}**.'.format(name, results.loc[0, 'score'], results.loc[0, 'iteration']))

*# Use best hyperparameters to create a model*

hyperparameters = results.loc[0, 'hyperparameters']

model = lgb.LGBMClassifier(\*\*hyperparameters)

*# Train and make predictions*

model.fit(train\_features, train\_labels)

preds = model.predict\_proba(test\_features)[:, 1]

print('ROC AUC from **{}** on test data = **{:.5f}**.'.format(name, roc\_auc\_score(test\_labels, preds)))

*# Create dataframe of hyperparameters*

hyp\_df = pd.DataFrame(columns = list(results.loc[0, 'hyperparameters'].keys()))

*# Iterate through each set of hyperparameters that were evaluated*

for i, hyp **in** enumerate(results['hyperparameters']):

hyp\_df = hyp\_df.append(pd.DataFrame(hyp, index = [0]),

ignore\_index = True)

*# Put the iteration and score in the hyperparameter dataframe*

hyp\_df['iteration'] = results['iteration']

hyp\_df['score'] = results['score']

return hyp\_df

grid\_hyp = evaluate(grid\_results, name = 'grid search')

텍스트이(가) 표시된 사진

자동 생성된 설명

random\_hyp = evaluate(random\_results, name = 'random search')

텍스트이(가) 표시된 사진

자동 생성된 설명

**시각화**

시각화는 만드는 것이 즐겁고 우리에게 기술을 직관적으로 볼 수 있게 해준다. 여기서는 맷플롭립, 시본, 알테어를 사용하여 몇 가지 간단한 그림을 그릴 것이다. 그러나 노트북을 렌더링할 때 알테어 시각화가 표시되지 않는다. 알테어 수치를 보려면 노트북을 직접 실행해야 한다.

먼저 검증 점수 대 반복을 그래프로 표시할 수 있다. 여기서는 알테어 시각화 라이브러리를 사용하여 그림을 그릴 것이다. 먼저 데이터를 긴 형식의 데이터 프레임에 넣어야 한다.

import altair as alt

alt.renderers.enable('notebook')

*# Combine results into one dataframe*

random\_hyp['search'] = 'random'

grid\_hyp['search'] = 'grid'

hyp = random\_hyp.append(grid\_hyp)

hyp.head()

max\_random = random\_hyp['score'].max()

max\_grid = grid\_hyp['score'].max()

c = alt.Chart(hyp, width = 400, height = 400).mark\_circle(size = 150).encode(alt.Y('score', scale = alt.Scale(domain = [0.65, 0.76])),

x = 'iteration', color = 'search')

c.title = 'Score vs Iteration'

c

아래에서는 알테어 시각화가 렌더링된 노트북에 나타나지 않기 때문에 시보은 사용하여 동일한 그래프를 만든다.

best\_grid\_hyp = grid\_hyp.iloc[grid\_hyp['score'].idxmax()].copy()

best\_random\_hyp = random\_hyp.iloc[random\_hyp['score'].idxmax()].copy()

hyp.sort\_values('search', inplace = True)

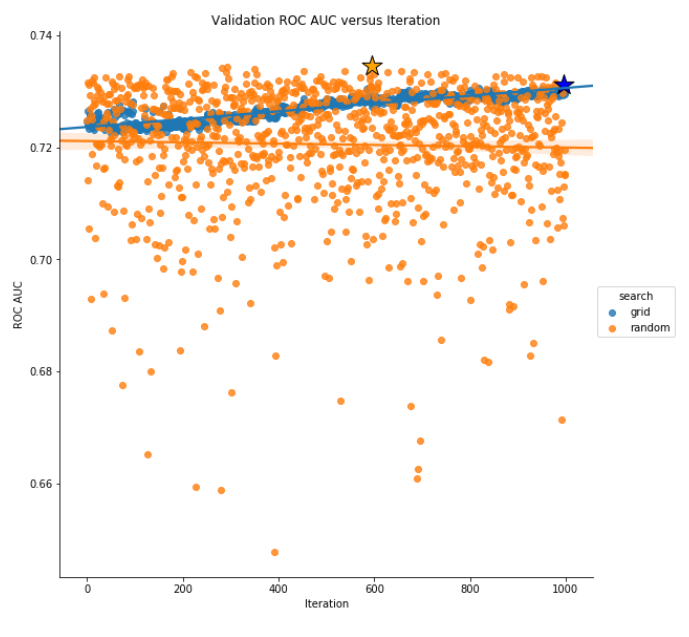
*# Plot of scores over the course of searching*

sns.lmplot('iteration', 'score', hue = 'search', data = hyp, size = 8);

plt.scatter(best\_grid\_hyp['iteration'], best\_grid\_hyp['score'], marker = '\*', s = 400, c = 'blue', edgecolor = 'k')

plt.scatter(best\_random\_hyp['iteration'], best\_random\_hyp['score'], marker = '\*', s = 400, c = 'orange', edgecolor = 'k')

plt.xlabel('Iteration'); plt.ylabel('ROC AUC'); plt.title("Validation ROC AUC versus Iteration");



print('Average validation score of grid search = **{:.5f}**.'.format(np.mean(grid\_hyp['score'])))

print('Average validation score of random search = **{:.5f}**.'.format(np.mean(random\_hyp['score'])))

텍스트, 오렌지, 어두운, 닫기이(가) 표시된 사진

자동 생성된 설명

그리드 교차 검증 점수는 시간이 지남에 따라 증가한다. 이는 그리드 서치에서 변경되는 하이퍼 파라미터가 점차 점수를 높이고 있음을 나타낸다. 반면 무작위 교차 검증 점수는 예상대로 천차만별이다. 이 그리드 검색은 검색 공간의 상대적으로 성능이 낮은 영역에 고착된 것으로 보이며, 그리드의 모든 값을 시도하도록 제한되기 때문에 더 나은 성능을 발휘할 수 없다. 랜덤 서치 방법은 검색된 하이퍼파라미터 값을 살펴볼 때 볼 수 있는 검색 공간을 탐색하는 데 매우 유용하다.

**검색 값의 분포**

랜덤 서치에 대한 검색 값의 분포를 표시할 수 있다. 우리는 이것들이 무작위로 될 것이라고 예상하지만, 우리의 코드를 양적으로나 시각적으로 확인하는 것은 항상 좋은 생각이다.

*# Create bar chart*

bars = alt.Chart(random\_hyp, width = 400).mark\_bar().encode(x = 'boosting\_type', y = alt.Y('count()', scale = alt.Scale(domain = [0, 400])))

bars.title = 'Boosting Type for Random Search'

*# Add text for labels*

text = bars.mark\_text(align = 'center', baseline = 'bottom', size = 20).encode(text = 'count()')

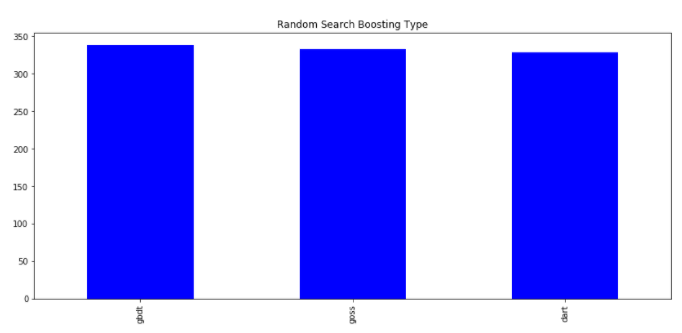
*# Display*

bars + text

부스팅 유형은 랜덤 서치를 위해 고르게 분포되어 있어야 한다. 렌더링된 노트북에 시각화가 나타나도록 하려면 이 차트를 시본으로 다시 작성해야 한다.

*# Bar plots of boosting type*

random\_hyp['boosting\_type'].value\_counts().plot.bar(figsize = (14, 6), color = 'blue', title = 'Random Search Boosting Type');



다음으로 숫자 하이퍼 파라미터의 경우, 커널 밀도 추정치(KDE) 그림에 샘플링 분포와 랜덤 서치 결과를 모두 표시하겠다. 랜덤 서치는 랜덤 값을 그리는 것이므로 랜덤 서치 분포가 샘플링 그리드와 정렬될 것으로 예상한다. 예를 들어, 아래는 표본 분포와 랜덤 서치 겨로가 모두에서 학습 속도 분포를 표시한다. 수직 파선은 임의 검색에서 찾은 최적의 값을 나타낸다.

random\_hyp['score'] = random\_hyp['score'].astype(float)

best\_random\_hyp = random\_hyp.loc[0, :].copy()

plt.figure(figsize = (20, 8))

plt.rcParams['font.size'] = 18

*# Density plots of the learning rate distributions*

sns.kdeplot(param\_grid['learning\_rate'], label = 'Sampling Distribution', linewidth = 4)

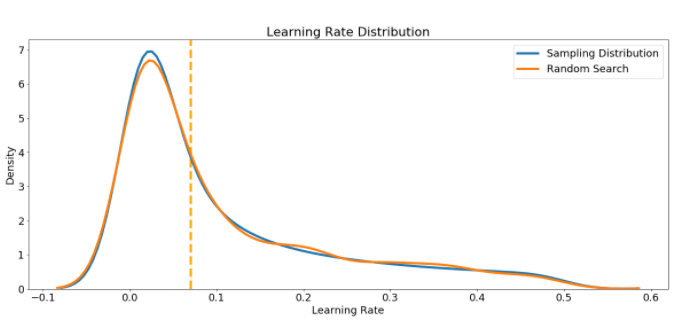
sns.kdeplot(random\_hyp['learning\_rate'], label = 'Random Search', linewidth = 4)

plt.vlines([best\_random\_hyp['learning\_rate']],

ymin = 0.0, ymax = 50.0, linestyles = '--', linewidth = 4, colors = ['orange'])

plt.legend()

plt.xlabel('Learning Rate'); plt.ylabel('Density'); plt.title('Learning Rate Distribution');



다음 코드는 모든 숫자 하이퍼 파라미터에 대해 이 그림을 반복한다.

*# Iterate through each hyperparameter*

for i, hyper **in** enumerate(random\_hyp.columns):

if hyper **not** **in** ['boosting\_type', 'iteration', 'subsample', 'score', 'learning\_rate', 'is\_unbalance', 'metric', 'verbose', 'iteration', 'n\_estimators', 'search']:

plt.figure(figsize = (14, 6))

*# Plot the random search distribution and the sampling distribution*

if hyper != 'loss':

sns.kdeplot(param\_grid[hyper], label = 'Sampling Distribution', linewidth = 4)

sns.kdeplot(random\_hyp[hyper], label = 'Random Search', linewidth = 4)

plt.vlines([best\_random\_hyp[hyper]],

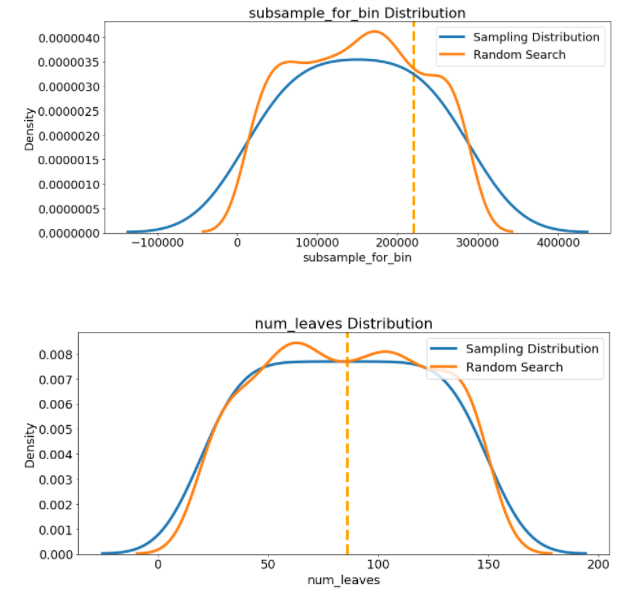
ymin = 0.0, ymax = 10.0, linestyles = '--', linewidth = 4, colors = ['orange'])

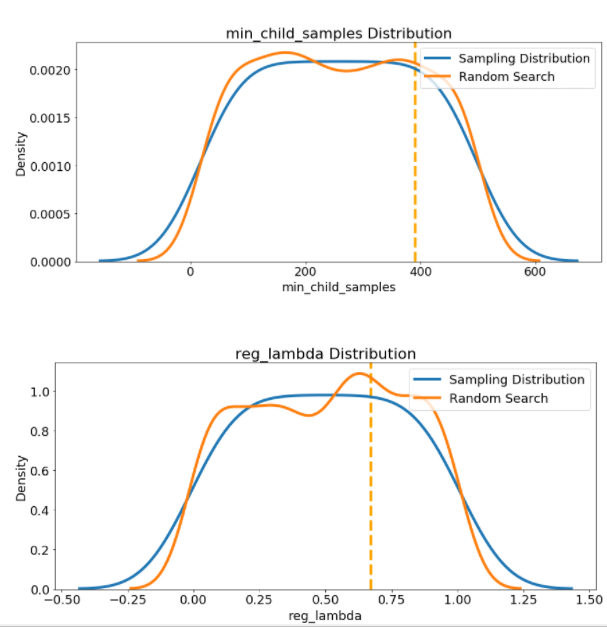
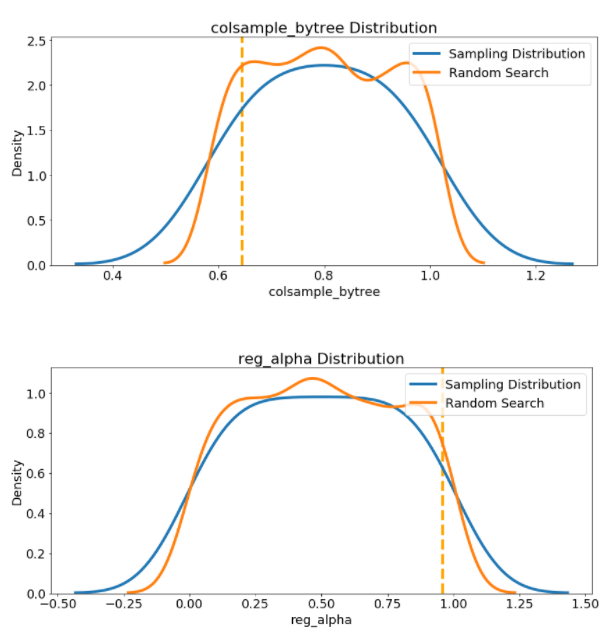
plt.legend(loc = 1)

plt.title('**{}** Distribution'.format(hyper))

plt.xlabel('**{}**'.format(hyper)); plt.ylabel('Density');

plt.show();





**검색 값의 순서**

마지막으로, 우리는 랜덤 서치르 ㄹ위한 반복에 대한 검색 값의 순서를 플롯할 수 있다. 분명히 어떤 순서도 없겠지만, 이것은 우리가 랜덤 서치에서 어떤 일이 일어나는지 시각화할 수 있게 해준다. 별은 발견된 하이퍼파라미터의 최상의 값을 나타낸다.

fig, axs = plt.subplots(1, 4, figsize = (24, 6))

i = 0

*# Plot of four hyperparameters*

for i, hyper **in** enumerate(['colsample\_bytree', 'learning\_rate', 'min\_child\_samples', 'num\_leaves']):

random\_hyp[hyper] = random\_hyp[hyper].astype(float)

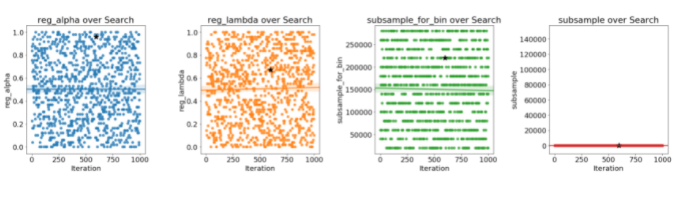
*# Scatterplot*

sns.regplot('iteration', hyper, data = random\_hyp, ax = axs[i])

axs[i].scatter(best\_random\_hyp['iteration'], best\_random\_hyp[hyper], marker = '\*', s = 200, c = 'k')

axs[i].set(xlabel = 'Iteration', ylabel = '**{}**'.format(hyper), title = '**{}** over Search'.format(hyper));

plt.tight\_layout()



점수 대 하이퍼파라미터 비교

마지막으로 그림으로 점수 대 각 하이퍼 파라미터 값을 표시할 수 있다. 하이퍼 파라미터는 한 번에 하나씩 변경되지 않으므로 값과 점수 사이에 관계가 있다고 해서 특정 하이퍼 파라미터가 점수에 영향을 미치는 것은 아니다. 그러나 보다 유망한 하이퍼 파라미터의 값을 식별할 수 있다. 대부분 이 줄거리는 나 자신을 위한 것이다. 어떤 유행이 있는지 보기 위함이다.

fig, axs = plt.subplots(1, 4, figsize = (24, 6))

i = 0

*# Plot of four hyperparameters*

for i, hyper **in** enumerate(['colsample\_bytree', 'learning\_rate', 'min\_child\_samples', 'num\_leaves']):

random\_hyp[hyper] = random\_hyp[hyper].astype(float)

*# Scatterplot*

sns.regplot(hyper, 'score', data = random\_hyp, ax = axs[i])

axs[i].scatter(best\_random\_hyp[hyper], best\_random\_hyp['score'], marker = '\*', s = 200, c = 'k')

axs[i].set(xlabel = '**{}**'.format(hyper), ylabel = 'Score', title = 'Score vs **{}**'.format(hyper));

plt.tight\_layout()

fig, axs = plt.subplots(1, 4, figsize = (24, 6))

i = 0

*# Scatterplot of next four hyperparameters*

for i, hyper **in** enumerate(['reg\_alpha', 'reg\_lambda', 'subsample\_for\_bin', 'subsample']):

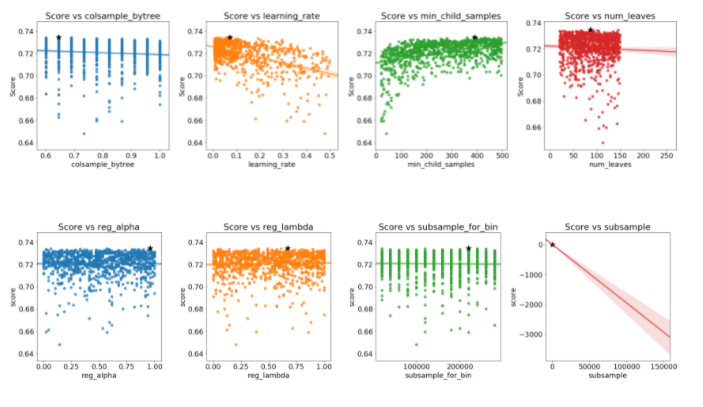
random\_hyp[hyper] = random\_hyp[hyper].astype(float)

sns.regplot(hyper, 'score', data = random\_hyp, ax = axs[i])

axs[i].scatter(best\_random\_hyp[hyper], best\_random\_hyp['score'], marker = '\*', s = 200, c = 'k')

axs[i].set(xlabel = '**{}**'.format(hyper), ylabel = 'score', title = 'Score vs **{}**'.format(hyper));

plt.tight\_layout()



한번에 하나의 하이퍼 파라미터를 변경하고 점수에 미치는 영향을 관찰하는 실험을 수행할 수 있지만 이러한 관계에 너무 많은 중점을 두는 것은 피하고자 한다. 따라서 추세는 우리가 보여주는 단일 하이퍼 파라미터 때문만은 아니다. 이를 더 높은 차원으로 표시할 수 있다면 검색 공간에 더 유망한 영역이 있는지 확인하는 것이 흥미로울 수 있지만 여기서는 단일 하이퍼 파라미터 대 점수로 제한된다. 교차 검증 점수에 대한 한 하이퍼 파라미터의 영향을 관찰하려면 다른 모든 하이퍼 파라미터를 일정하게 유지하면서 해당 하이퍼파라미터만 변경할 수 있다. 그러나 하이퍼파라미터는 저절로 작용하지 않으며 모형 설정 간의 복잡한 교호작용이 있다.

**전체 데이터에 대한 테스트 결과**

축소된 교육 데이터에 대해 1000번의 랜덤 서치 반복에서 찾은 최고의 하이퍼 파라미터를 선택하여 전체 교육 데이터 세트에 적용할 수 있다. 아래 코드는 최적의 랜덤 서치 하이퍼 파라미터를 사용하여 모델을 구축하고 전체 훈련 데이터를 교육한 후 테스트 데이터를 테스트한다. 테스트 데이터는 실제 경쟁 데이터이므로, 이 데이터를 제출하여 점수가 전체 데이터 세트로 얼마나 잘 변환되는지 확인할 수 있다.

*# Read in full dataset*

train = pd.read\_csv('../input/home-credit-simple-featuers/simple\_features\_train.csv')

test = pd.read\_csv('../input/home-credit-simple-featuers/simple\_features\_test.csv')

*# Extract the test ids and train labels*

test\_ids = test['SK\_ID\_CURR']

train\_labels = np.array(train['TARGET'].astype(np.int32)).reshape((-1, ))

train = train.drop(columns = ['SK\_ID\_CURR', 'TARGET'])

test = test.drop(columns = ['SK\_ID\_CURR'])

print('Training shape: ', train.shape)

print('Testing shape: ', test.shape)

텍스트이(가) 표시된 사진

자동 생성된 설명

먼저 랜덤 서치의 최적 모델 하이퍼 파라미터 값을 사용하여 교차 검증 점수를 테스트한다. 이를 통해 테스트 세트의 일반화 오류를 알 수 있다. 더 작은 데이터 세트에서 발견된 추정기 수를 삭제하고 조기 중지를 사용하여 교육할 최적의 의사 결정 트리 수를 찾을 것이다.

train\_set = lgb.Dataset(train, label = train\_labels)

hyperparameters = dict(\*\*random\_results.loc[0, 'hyperparameters'])

del hyperparameters['n\_estimators']

*# Cross validation with n\_folds and early stopping*

cv\_results = lgb.cv(hyperparameters, train\_set,

num\_boost\_round = 10000, early\_stopping\_rounds = 100,

metrics = 'auc', nfold = N\_FOLDS)

print('The cross validation score on the full dataset = **{:.5f}** with std: **{:.5f}**.'.format(

cv\_results['auc-mean'][-1], cv\_results['auc-stdv'][-1]))

print('Number of estimators = **{}**.'.format(len(cv\_results['auc-mean'])))

텍스트이(가) 표시된 사진

자동 생성된 설명

공개 리더보드 점수는 테스트 데이터의 10%에 대해서만 계산되므로 교차 검증 점수는 실제로 모델이 전체 테스트 세트에서 어떻게 수행될지 더 잘 알 수 있다. 일반적으로 교차 검증 점수는 테스트 데이터보다 높을 것으로 예상하지만 테스트 데이터의 크기가 작기 때문에 이 문제에 대해서는 역순으로 나타날 수 있다.

다음으로 대회에 제출할 수 있는 시험 자료를 예측하겠다.

*# Train the model with the optimal number of estimators from early stopping*

model = lgb.LGBMClassifier(n\_estimators = len(cv\_results['auc-mean']), \*\*hyperparameters)

model.fit(train, train\_labels)

*# Predictions on the test data*

preds = model.predict\_proba(test)[:, 1]

submission = pd.DataFrame({'SK\_ID\_CURR': test\_ids, 'TARGET': preds})

submission.to\_csv('submission\_simple\_features\_random.csv', index = False)

시험대회 제출 시 점수는 0.782점이다. 이러한 기능을 얻은 커널의 원래 점수는 0.792이므로 더 작은 데이터 세트에 대한 랜덤 서치로 전체 데이터 세트로 변환되지 않은 결과를 얻을 수 있다.

따라서 작은 데이터셋에서 랜덤 검색을 수행한 결과가 전체 데이터셋으로 변환되지 않는다는 결론을 내릴 수 있습니다. 현재 전체 데이터 세트에 대해 500번의 반복으로 랜덤 검색을 실행하고 있으며 검색이 완료되면 이러한 결과를 공개적으로 제공할 예정입니다.

**모델 조정 후 단계**

여기서는 작성한 함수를 전체 데이터 세트에 적용할 수 있다. 교육 데이터의 임의 부분 집합만 사용했기 때문에 결과가 다를 수 있다. 그러나 이 작업은 훨씬 더 오래 걸린다(100개 대신 30만개 이상의 관측치). 현재 위에서 언급한 커널의 전체 데이터 세트에 대해 임의 검색을 실행하고 있으며, 결과가 어떻게 나오는지 확인할 수 있다(일부 관찰 결과를 샘플링하는 것은 본질적으로 부정적인 것이 아니며 훨씬 짧은 시간 내에 합리적인 답변을 얻는 데 도움이 될 수 있다). 하지만 전체 데이터 세트를 나타내지 않는 데이터의 작은 부분을 사용하는 경우에는 조정 기능이 전체 데이터 세트로 변활될 것으로 예상해서는 안된다.

**결론**

모델 조정은 특정 문제에 가장 적합한 머신러닝 모델 하이퍼 파라미터를 찾는 프로세스이다. 무작위 및 그리드 서치는 그리드 도메인에서 하이퍼파라미터 값을 선택하여 검색하는 하이퍼 파라미터 값을 선택하여 검색하는 하이퍼 파라미터 조정을 위한 두 가지 통일된 방법이다. 하이퍼 파라미터 조정의 네 가지 부분은 다음과 같다.

목표 기능: 하이퍼 파라미터를 사용하고 최대화 또는 최소하하려는 교차 검증 점수를 반환한다.

하이퍼 파라미터의 영역: 검색하려는 값이다.

알고리즘: 목적 함수에서 평가할 다음 하이퍼 파라미터 값을 선택하는 방법이다.

결과: 하이퍼 파라미터 및 교차 검증 점수 내역이다.

이 네 가지 부분은 그리드 및 랜더 ㅁ서치뿐만 아니라 자동화된 하잎 ㅓ파라미터 튜닝의 한 형태인 베이지안 최적화에도 적용된다. 이 노트북에서는 줄어든 데이터셋에 랜덤 치 그리드 서치를 구현하고 결과를 검사한 후 최적의 하이퍼 파라미터를 전체 데이터셋으로 변환하려고 했다. 참고로 교차 검증을 사용하여 교육 데이터에 맞게 ㅎ ㅏ이퍼 파라미터를 조정하므로 발견한 하이퍼 파라미터 값이 교육 데이터에만 최적이 될 수 있다.

소규모 데이터셋에서 가장 우수한 하이퍼 파라미터가 전체 데이터셋에서 제대로 작동하지 않았지만, 이 두 가지 튜닝 방법에 대한 아이디어를 확인할 수 있었다. 또한 여기서 개발된 기능을 부스팅 기법 뿐만 아니라 모든 데이터 세트 또는 기계 학습 모델에 적용할 수 있다.

랜덤 서치는 실제로는 꽤 잘 작동하지만 여전히 다음 하이퍼 파라미터 값을 선택하는 데 과거 평가 결과를 사용하지 않기 때문에 추론 방법은 아니다. 보다 나은 접근 방식은 과거 결과를 사용하여 목표 함수에서 다음으로 시도할 수 있는 최상의 값에 대해 추론하는 것이다. 특히 목표 함수를 평가하는 것은 시간이 많이 소요되기 때문이다. 향후 연구에서는 베이지안 최적화를 이용한 자동화된 하이퍼 파라미터 튜닝 구현에 대해 알아보겠다.

모델의 성능은 하이퍼 파라미터 값의 선택에 따라 크게 좌우될 수 있기 때문에 하이퍼 파라미터 조정은 기계 학습 파이프라인의 중요한 부분이다. 랜덤 및 그리드 서치는 모델 조정을 시작할 수 있는 두 가지 적절한 방법이며, 데이터 과학 기술 세트에서 가져야 할 중요한 도구이다.