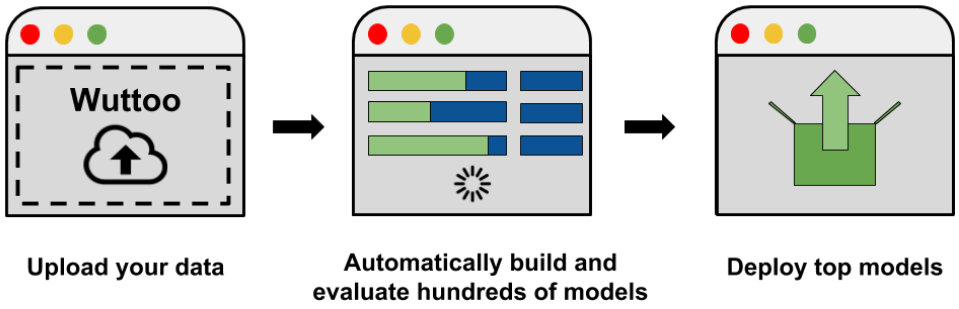
**적당한 '정확도'가 보장되는 모델을 '자동으로' 만들 수는 없을까? | by Yongki Lee | DLIFT | Medium**

**노트북:** 첫 번째 노트북
**만든 날짜:** 2021-06-08 오후 11:11
**URL:** https://medium.com/dlift/%EC%A0%81%EB%8B%B9%ED%95%9C-%EC%A0%95%ED%9...

# 적당한 '정확도'가 보장되는 모델을 '자동으로' 만들 수는 없을까?

Yongki Lee
May 22, 2018 · 22 min read

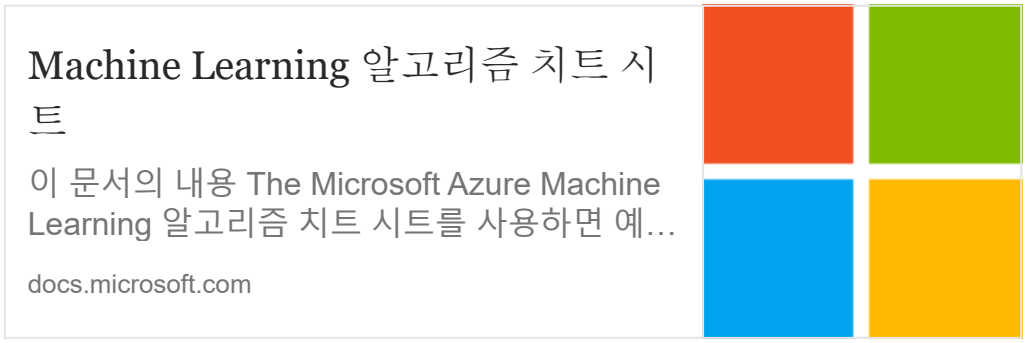



http://wuttoo.com/

모델을 만들고 평가하는 일을 가만히 생각해보다가, 아래와 같은 사고의 흐름에 빠지게 되었다.

- 알고리즘을 선택하고, 그 알고리즘에 입력해줘야 하는 파라미터들에 대한 '튜닝'은 소모적인 과정임
- 만약 데이터의 특성에 따라 선택하면 좋을 알고리즘과 어울리는 파라미터들에 대한 가이드라인이 있다면 정말 편할텐데!

마치 Microsoft Azure Machine Learning: Algorithm Cheat Sheet 와 같이 말이다.




여기서 재미있는 사실은 이런 생각을 '많이 들' 했다라는 것이고,

데이터의 '특성'에 따라 선택하면 좋을 알고리즘과 어울리는 파라미터들을 '합리적'으로 '잘 찾는' 문제 = **AutoML 이라고 한다**는 것이다.
(게다가 AutoML 의 구현체들은 https://github.com/automl 에서 찾을 수 있다)

- 여기에서는 그 중에서도 가장 별을 많이 받은 auto-sklearn 을 Binary Classification 사례에 적용해보고
- 기존에 이 문제를 풀면서 튜닝했던 모델 "가입권유 막 하지 말자구요" 의 Performance 와 비교해본 후,
- 좋다면 왜 좋은지? 어떻게 동작하길래 좋은지? 등에 대해 알아보도록 하자

. . .

## 정기예금상품 가입권유 데이터를 auto-sklearn 으로?

정기예금상품 가입에 대한 확률모델을 만들 때, 아래와 같이 Random Search 코드를 작성했던 기억이 있다.

```
from sklearn.model_selection import
RandomizedSearchCV
```

```
from sklearn.metrics import make_scorer,
precision_score
import numpy as np
from pprint import pprint

n_estimators = [int(x) for x in np.linspace(start
= 100, stop = 500, num = 10)]
max_features = ['auto', 'sqrt']
max_depth = [int(x) for x in np.linspace(10, 110,
num = 11)]
max_depth.append(None)
min_samples_split = [2, 5, 10]
min_samples_leaf = [1, 2, 4]
bootstrap = [True, False]

# 파라미터 조합 생성
random_grid = {'n_estimators': n_estimators,
               'max_features': max_features,
               'max_depth': max_depth,
               'min_samples_split':
min_samples_split,
               'min_samples_leaf':
min_samples_leaf,
               'bootstrap': bootstrap}

pprint(random_grid)

# 생성된 조합에 대한 실험 시작!
from sklearn.ensemble import
RandomForestClassifier
rf = RandomForestClassifier()
rf_random = RandomizedSearchCV(
              estimator = rf,
              param_distributions = random_grid,
              n_iter = 100,
              cv = 3,
              verbose = False,
              scoring =
make_scorer(precision_score),
              random_state=42, n_jobs = -1)

rf_random.fit(x_train, y_train)
```

코드를 보면 알겠지만,

- 정말 감으로 파라미터 스페이스를 잡고
- 이 중에 100개의 조합을 Random 하게 선택해서 돌린 후
- Precision 을 토대로 각 조합에 대한 평가를 진행했었다.

물론 이 방법이 나쁜 성능을 가져다 주지는 않았지만, 뭔가 “감"과 “우연”에 의존하는 방법이 아닐까? 라는 “불만족스러움" 이 자리 했던 것은 사실이었던 거 같다.

그래서 시도해보았다. 앞서 말 한 대로, 이 AutoML 의 구현체 중 가장 별이 많은 auto-sklearn 을 돌려보는 것 말이다.

참고로 auto-sklearn 을 돌리는 것은 정말 쉬웠다.

**“세상에 .fit 함수 하나면 최적의 모델을 선택해주다니!”**

```
from autosklearn.classification import
AutoSklearnClassifier
from sklearn.model_selection import
train_test_split

# 5분 탐색을 하도록 해보자
automl =
AutoSklearnClassifier(time_left_for_this_task =
300,
                                  tmp_folder =
"./log/")
automl.fit(x_train, y_train)
```

정말 library 를 import 하는 코드라인을 제외하고, 단 두 줄이면 Random Search 코드와 같이, “감"에 의존하는 면 (파라미터 범위를 셋업할 때를 생각해보자) 이 있다고 생각했던 코드를 대체할 수 있었으니 말이다.

- 참고로 time_left_for_this_task 옵션은 꽤 중요하다. 파라미터를 찾는 전체 과정에 소요되는 총 시간을 결정할 수 있는 옵션이기 때문이다.
  (늘 그렇지만 시간이라는 자원은 너무나 소중하기에! 잘 챙겨두도록 하자)

- tmp_folder 옵션은 .fit 을 콜 한 후 일어나는 모든 과정에서 기록되는 산출물을 저장하는 root file path 정도 된다고 생각하면 된다.

(실제로, 코드의 동작을 세세하게 알 수 있는 로그 역시 이 폴더 밑에 쌓이기 때문에 가장 먼저 챙기지 않을 수 없었다.)

**"코드 실행의 결과는?"**

5분남짓을 돌렸던 결과는 가히 재미있었다.

```
model, weight : 0.32000000000000006
----------
SimpleClassificationPipeline({'categorical_encoding:__choice__': 'one_hot_encoding', 'classifier:random_forest:max_features': 0.5, 'balancing:strategy': 'none', 'categorical_encoding:one_hot_encoding:use_minimum_fraction': 'True', 'classifier:random_forest:bootstrap': 'True', 'classifier:random_forest:criterion': 'gini', 'classifier:random_forest:min_samples_leaf': 1, 'rescaling:__choice__': 'standardize', 'classifier:random_forest:min_weight_fraction_leaf': 0.0, 'classifier:__choice__': 'random_forest', 'imputation:strategy': 'mean', 'categorical_encoding:one_hot_encoding:minimum_fraction': 0.01, 'classifier:random_forest:n_estimators': 100, 'classifier:random_forest:min_samples_split': 2, 'classifier:random_forest:max_depth': 'None', 'classifier:random_forest:max_leaf_nodes': 'None', 'classifier:random_forest:min_impurity_decrease': 0.0, 'preprocessor:__choice__': 'no_preprocessing'},
dataset_properties={
  'task': 1,
  'target_type': 'classification',
  'signed': False,
  'multiclass': False,
  'multilabel': False,
  'sparse': False})


model, weight : 0.18000000000000002
----------
SimpleClassificationPipeline({'classifier:gradient_boosting:subsample': 0.9448890820738562, 'classifier:gradient_boosting:learning_rate': 0.1958974686405233, 'preprocessor:polynomial:interaction_only': 'False', 'classifier:gradient_boosting:min_samples_leaf': 6, 'classifier:gradient_boosting:min_samples_split': 4, 'classifier:gradient_boosting:min_impurity_decrease': 0.0, 'classifier:gradient_boosting:criterion': 'mse', 'classifier:__choice__': 'gradient_boosting', 'preprocessor:polynomial:degree': 2, 'classifier:gradient_boosting:max_leaf_nodes': 'None', 'classifier:gradient_boosting:loss': 'deviance', 'classifier:gradient_boosting:max_features': 0.33885235607979314, 'preprocessor:__choice__': 'polynomial', 'categorical_encoding:__choice__': 'one_hot_encoding', 'balancing:strategy': 'weighting', 'categorical_encoding:one_hot_encoding:use_minimum_fraction': 'False', 'rescaling:__choice__': 'none', 'classifier:gradient_boosting:n_estimators': 125, 'imputation:strategy': 'median', 'classifier:gradient_boosting:max_depth': 5, 'preprocessor:polynomial:include_bias': 'False', 'classifier:gradient_boosting:min_weight_fraction_leaf': 0.0},
dataset_properties={
  'task': 1,
  'target_type': 'classification',
  'signed': False,
  'multiclass': False,
  'multilabel': False,
  'sparse': False})


model, weight : 0.12000000000000001
----------
SimpleClassificationPipeline({'classifier:gradient_boosting:subsample': 0.3870344708308441, 'classifier:gradient_boosting:learning_rate': 0.018356703878357986, 'preprocessor:polynomial:interaction_only': 'True', 'classifier:gradient_boosting:min_samples_leaf': 12, 'classifier:gradient_boosting:min_samples_split': 3, 'classifier:gradient_boosting:min_impurity_decrease': 0.0, 'classifier:gradient_boosting:criterion': 'mse', 'classifier:__choice__': 'gradient_boosting', 'preprocessor:polynomial:degree': 2, 'categorical_encoding:one_hot_encoding:minimum_fraction': 0.3837398524575939, 'classifier:gradient_boosting:max_leaf_nodes': 'None', 'classifier:gradient_boosting:loss': 'deviance', 'classifier:gradient_boosting:max_features': 0.9690352514774068, 'preprocessor:__choice__': 'polynomial', 'categorical_encodin
```

.fit 실행의 결과

위의 결과는 8개의 classifier 들(e.g., Random Forest, Gradient Boosting 등) 이 서로 다른 weight 로 결합된 앙상블 모델 이 추천된 것이다.

게다가 이 앙상블 모델의 정확도를 살펴보면 다음과 같았다.

| | AUC | Accuracy | Precision | Recall |
|---|---|---|---|---|
| **0** | 0.913284 | 0.843648 | 0.819646 | 0.864177 |

정확도

“가입권유 막 하지 말자구요” 에서 사용한
GradientBoostingClassifier 를 Tuning 한 결과가

- AUC : 0.92, Precision : 0.83, Recall : 0.84

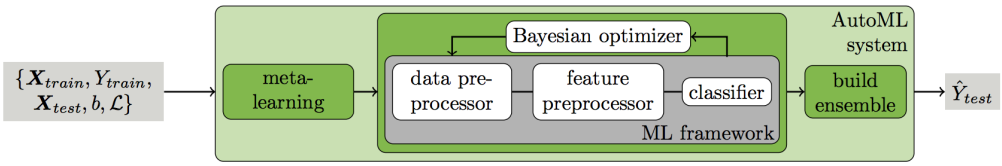
였다는 것을 기억한다면, 성능차이는 그렇게 크지 않다라고 확인할 수 있다.

Tuning 에 고민했던 시간을 생각해본다면, “왜 진즉, auto-sklearn 을 사용하지 않았을까" 라는 생각마저 든다.

## 그런데, 이런 결과는 어떻게 나오는 걸까요?

Efficient and Robust Automated Machine Learning 에서는 아래의 세 단계를 거쳐, auto-sklearn 이 동작한다라고 설명하고 있다.

- 데이터가 들어오면, 우선 이 데이터에 어울릴 알고리즘 / 파라미터들을 알려주는 meta-learning process 가 진행된다.
- 그리고 이 meta-learning process 의 산출물로 알고리즘 / 파라미터 셋들을 추천해준다.
- 마지막으로 앙상블 기법을 활용해 추천된 알고리즘 / 파라미터 셋들의 최적화를 진행한다.




Architecture (https://papers.nips.cc/paper/5872-efficient-and-robust-automated-machine-learning.pdf)

즉, 이 과정은 “들어온 데이터의 특성에 맞는 알고리즘 / 파라미터를 추천해주고 그것들을 앙상블해서 최종 모델을 만들어내는 과정" 이라고 요약할 수 있다.

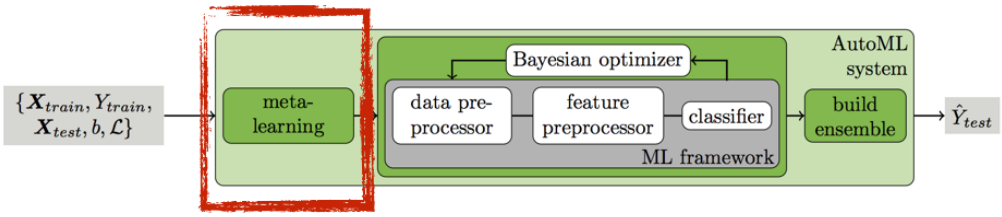
하지만 어떻게 알고리즘 및 파라미터들을 추천 해주며, 그것들을 어떻게 앙상블을 하는지? 에 대한 궁금증은 아직 해소되지 않았기에 ...

각 단계를 좀 더 살펴보며 이 궁금증을 해소시켜보도록 하자.

**“meta-learning 은 어떻게 들어온 데이터에 알맞은 알고리즘 / 파라미터들을 알려줄까요?”**

논문에서는 meta-learning 의 과정을 아래와 같은 문장으로 소개하고 있다.

```
"More specifically, for a large number of
datasets, we collect both performance data and a
set of meta-features, i.e., characteristics of the
dataset that can be computed efficiently and that
help to
determine which algorithm to use on a new dataset"
```




Meta-Learning

즉,

- 사전에 데이터 별로 가지고 있는 meta-feature 들과 알고리즘 / 파라미터에 따른 퍼포먼스를 기록해둔다.
- 새로운 데이터가 들어왔을 때, 그것의 meta-feature 들과 기존 데이터들의 meta-feature 들 간의 “유사도” 를 측정한 후
- 가장 유사한 기존 데이터들을 찾아낸다. 그리고 기존 데이터에 매칭되어 있는 알고리즘 / 파라미터 조합을 준다.
  (이미 기존 데이터에 알맞은 알고리즘 / 파라미터 조합을 알고 있기 때문에 그것을 바로 알려줄 수 있다.)

참고로 “사전에 가지고 있는 데이터들” 은 OpenML 데이터 셋 을 의미한다.

"데이터를 직접 돌려가며 이 과정을 경험해보자"

정기예금상품가입권유 데이터를 토대로 이 과정이 어떻게 진행되었는지 살펴보았다. (아래의 표는 정기예금상품-가입권유 데이터의 meta-features 가 뽑힌 결과를 나타낸 것이다.)

| | value |
|---|---|
| ClassEntropy | 0.997569 |
| ClassProbabilityMax | 0.529018 |
| ClassProbabilityMean | 0.500000 |
| ClassProbabilityMin | 0.470982 |
| ClassProbabilitySTD | 0.029018 |
| DatasetRatio | 0.006419 |
| InverseDatasetRatio | 155.791667 |
| KurtosisMax | 126.328167 |
| KurtosisMean | 18.894389 |
| KurtosisMin | -1.990604 |
| KurtosisSTD | 30.171476 |
| Landmark1NN | 0.714774 |
| LandmarkDecisionNodeLearner | 0.716773 |
| LandmarkDecisionTree | 0.778421 |
| LandmarkLDA | 0.799414 |
| LandmarkNaiveBayes | 0.715167 |
| LandmarkRandomNodeLearner | 0.529018 |
| LogDatasetRatio | -5.048520 |
| LogInverseDatasetRatio | 5.048520 |
| LogNumberOfFeatures | 3.871201 |
| LogNumberOfInstances | 8.919721 |
| NumberOfCategoricalFeatures | 0.000000 |
| NumberOfClasses | 2.000000 |
| NumberOfFeatures | 48.000000 |

정기예금상품-가입권유 데이터의 Meta Features

오! 뭐랄까 "데이터의 생김새" 라고 부를 만한 통계적인 특징들이 꽤 뽑혀져 나오는 것을 확인할 수 있다.

이 특징들의 면면은 크게 4가지로 구분할 수 있다.

- 행의 개수, 열의 개수, 변수 별 결측치의 수 등과 같이 데이터 그 자체를 descriptive 하게 묘사하는 **Simple Meta Features**
- 변수 별 분포 특징 (막 특정 값으로 수렴하는지? 혹은 어떤 값으로 치우쳐있는지 등, Kurtosis-Skewness Max, Mean, Min, STD) 등과 같이 데이터의 통계적인 특징을 묘사하는 **Statistical Meta Features**
- 예측하고자 하는 '타겟' 이 한쪽으로 치우쳐있는지? 등 (ClassEntropy, ClassProbabilityMax, Mean, Min, STD) Information-Theoretic 한 (e.g., 엔트로피 같은) 메트릭으로 데이터를 묘사하는 **Information-Theoretic Meta Features**
- Naive Bayes, Decision Tree 등 복잡도가 크지 않은 모델이 판단하는 Accuracy 로 데이터를 묘사하는 **Landmarking Features**

즉, 새로 입력된 데이터는

- Simple Meta Features
- Statistical Meta Features
- Information-Theoretic Meta Features
- Landmarking Features

를 나타내는 46개의 특징들로 표현되며 이 특징들을 이용해서 auto-sklearn 이 가지고 있는 기존의 데이터 '들' 과의 '유사도'를 아래와 같이 계산할 수 있는 것이었다!

**"46개의 특징들을 이용해서 '가장 가까운 거리'를 가지고 있는 기존 데이터셋들을 찾기"**

```
[('847_acc', 1.4400319734004678, '2'),
 ('904 acc', 1.7337023784411743, '24'),
```

```
('743_acc', 1.8073405202831938, '23'),
('806_acc', 1.8514474059324668, '21'),
('833_acc', 1.8822403382912076, '3'),
('849_acc', 1.9434928053410636, '17'),
('797_acc', 1.9481076146955585, '15'),
('741_acc', 2.0509299640789767, '8'),
('866_acc', 2.1153044107541752, '10'),
('60_acc', 2.1618595610977631, '11'),
('718_acc', 2.2451719458075892, '16'),
('772_acc', 2.886855092030411, '12'),
('991_acc', 2.8992078354129998, '22'),
('871_acc', 2.9304215667152351, '7'),
('846_acc', 3.1551113554603449, '14'),
('734_acc', 3.9935959597310244, '18'),
('23_acc', 4.7845055758650572, '4'),
('1020_acc', 4.7879684285318591, '20'),
('14_acc', 4.9394438345084506, '19'),
('391_acc', 6.3275805010419672, '5'),
('392_acc', 6.9517398390770273, '13'),
('401_acc', 7.0581106937283469, '6'),
('1116_acc', 7.4708241685525696, '9'),
('1000_acc', 7.4727586097284817, '1')]
```

정기예금상품-가입권유 데이터와 유사한 기존 데이터 셋들

위의 표는 48개의 특징 들로 표현된 정기예금상품 가입권유데이터와 가장 유사한 “기존 데이터 셋들” 을 구한 결과이다. (참고로 ‘유사함의 정도’ 는 ‘Minkowski Distance’ 를 이용해서 측정한다.)

결과는 차례대로,

- 데이터 셋 이름, Minkowski Distance, 이 데이터 셋을 최적으로 돌릴 수 있는 알고리즘 / 파라미터 조합에 대한 인덱스

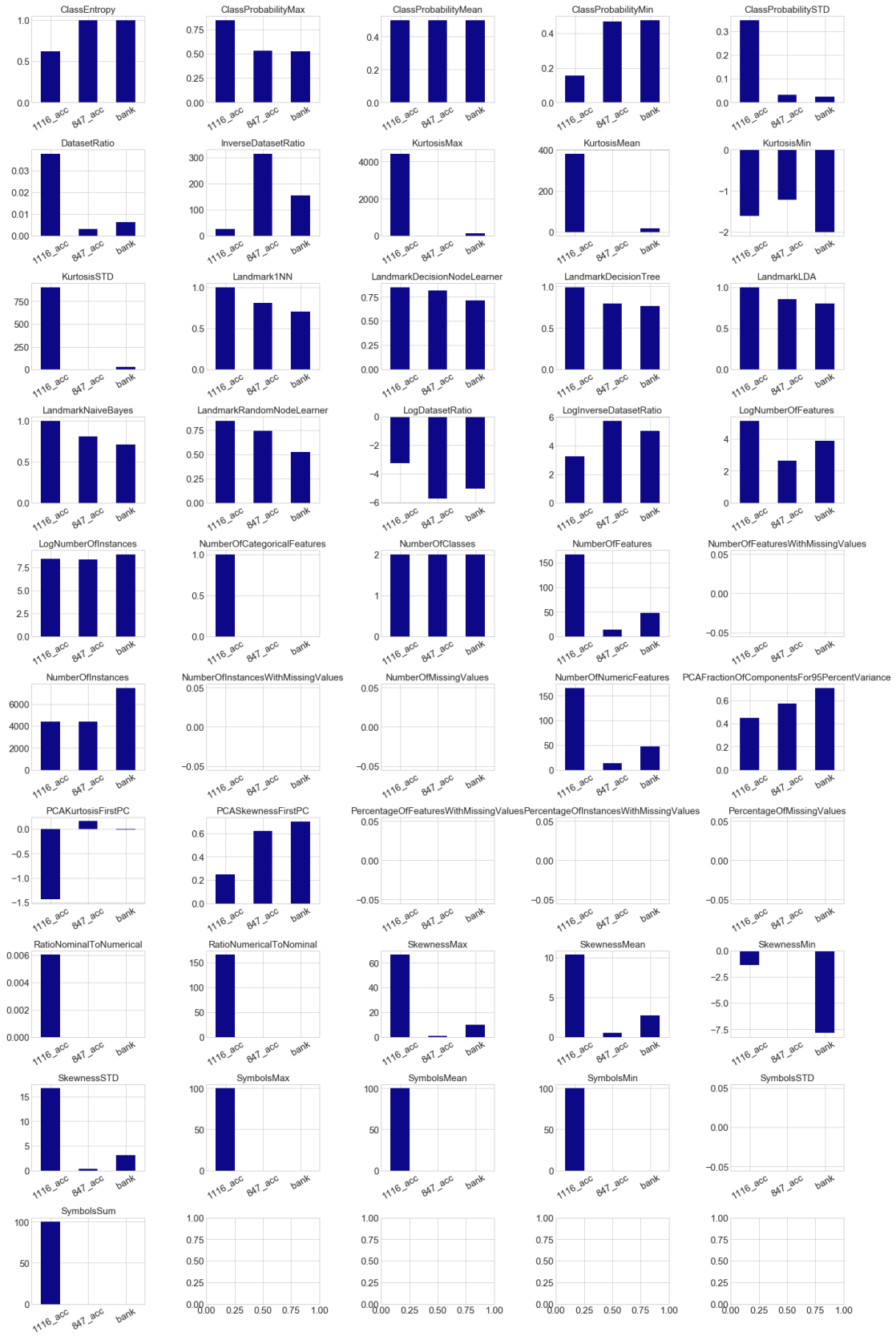
를 의미한다.

즉, 847_acc 라는 이름의 데이터 셋이 정기예금상품 가입권유데이터와 가장 유사한 특성을 가지고 있으며 그것을 최적으로 돌려주는 알고리즘 / 파라미터 조합에 대한 인덱스는 2번이다! 라는 것을 알 수 있는 것이다.

**“여기서 잠깐!, 847_acc 와 정기예금상품데이터 그리고 Random 한 다른 데이터 (1116_acc) 를 뽑아서 이 세 개의 데이터 특징 (46개의) 을 비교해본다면?”**

이건 뭐 당연한 결과이지만,

847_acc & 정기예금상품데이터는 어느 정도 유사한 데 비해, 1116_acc 는 그렇지 않다라는 것을 아래 그래프에서 확인할 수 있다.




847_acc, 정기예금상품-가입권유데이터, 1116_acc 데이터 간의 특징 비교

여튼, 이런 식으로 46개의 특징을 이용해서 정기예금상품-가입권유데이터의 데이터 특징과 가장 가까운 데이터 셋들과 그것들에 붙어있는 알고리즘 / 파라미터 조합을 아래와 같이 추천받는다.

```
Configuration:
  balancing:strategy, Value: 'none'
  categorical_encoding:__choice__, Value: 'no_encoding'
  classifier:__choice__, Value: 'random_forest'
  classifier:random_forest:bootstrap, Value: 'True'
  classifier:random_forest:criterion, Value: 'gini'
  classifier:random_forest:max_depth, Constant: 'None'
  classifier:random_forest:max_features, Value: 0.5
  classifier:random_forest:max_leaf_nodes, Constant: 'None'
  classifier:random_forest:min_impurity_decrease, Constant: 0
  classifier:random_forest:min_samples_leaf, Value: 1
  classifier:random_forest:min_samples_split, Value: 2
  classifier:random_forest:min_weight_fraction_leaf, Constant: 0
  classifier:random_forest:n_estimators, Constant: 100
  imputation:strategy, Value: 'mean'
  preprocessor:__choice__, Value: 'no_preprocessing'
  rescaling:__choice__, Value: 'minmax'

--------------------------------------------------


Configuration:
  balancing:strategy, Value: 'none'
  categorical_encoding:__choice__, Value: 'no_encoding'
  classifier:__choice__, Value: 'adaboost'
  classifier:adaboost:algorithm, Value: 'SAMME'
  classifier:adaboost:learning_rate, Value: 0.723185509
  classifier:adaboost:max_depth, Value: 5
  classifier:adaboost:n_estimators, Value: 357
  imputation:strategy, Value: 'mean'
  preprocessor:__choice__, Value: 'extra_trees_preproc_for_classification'
  preprocessor:extra_trees_preproc_for_classification:bootstrap, Value: 'True'
  preprocessor:extra_trees_preproc_for_classification:criterion, Value: 'gini'
  preprocessor:extra_trees_preproc_for_classification:max_depth, Constant: 'None'
  preprocessor:extra_trees_preproc_for_classification:max_features, Value: 0.5
  preprocessor:extra_trees_preproc_for_classification:max_leaf_nodes, Constant: 'None'
  preprocessor:extra_trees_preproc_for_classification:min_impurity_decrease, Constant: 0
  preprocessor:extra_trees_preproc_for_classification:min_samples_leaf, Value: 5
  preprocessor:extra_trees_preproc_for_classification:min_samples_split, Value: 8
  preprocessor:extra_trees_preproc_for_classification:min_weight_fraction_leaf, Constant: 0
  preprocessor:extra_trees_preproc_for_classification:n_estimators, Constant: 100
  rescaling:__choice__, Value: 'standardize'
```

추천된 알고리즘 및 파라미터 조합들

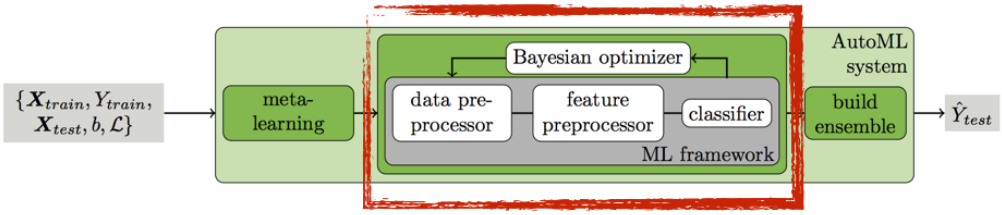
**“근데 기존 데이터 셋들에 대한 알고리즘, 파라미터를 어떻게 찾았을까?”**

알고리즘 및 파라미터에 대한 조합들에 대한 경우의 수는 굉장히 많다. 아니 ‘굉장히’ 라는 말보다는 ‘무한하다’라는 표현이 더 어울릴 지 모른다.

이렇게 많은 조합들 가운데에서, ‘기존 데이터 셋들’ 을 잘 분류하거나, 잘 예측하는 알고리즘 및 파라미터 조합은 어떻게 찾았을까? 설마 그 많은 조합을 다 돌려보고 일일이 기록해 둔 것은 아니겠지?

**“Bayesian Optimization”**

auto-sklearn 에서는 Bayesian Optimization 을 이용해서 기존 데이터 셋들에 대한 ‘최적의 조합’을 찾고, 기록해둔다.




Bayesian Optimization 과정

그 과정을 다시 말하면 다음과 같다.

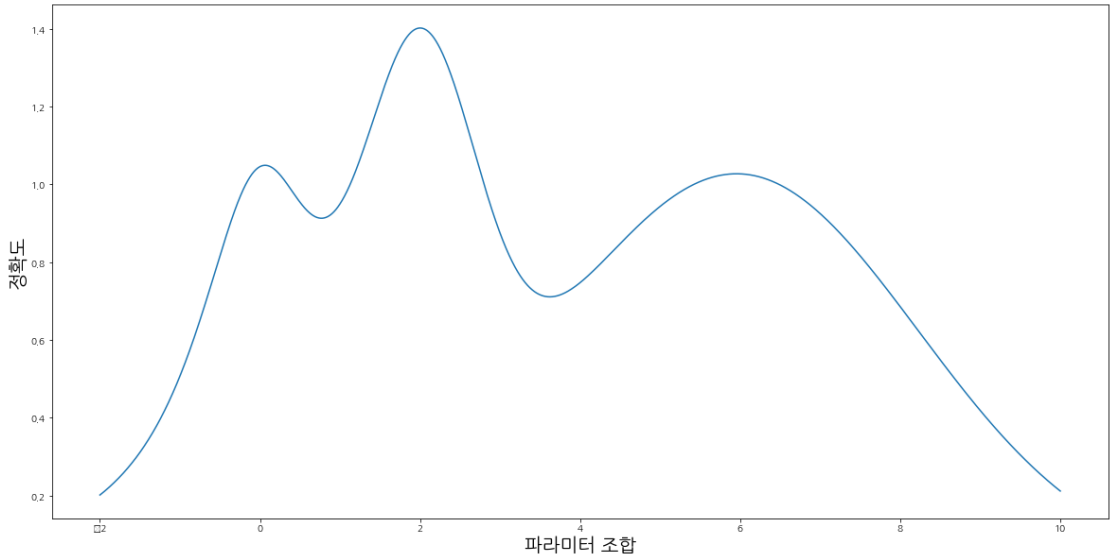
- 찾아 볼 알고리즘 및 파라미터 조합들을 마련해두고,
- **Bayesian Optimization** (여기서는 **SMAC = Sequential Model-Based Optimization for General Algorithm Configuration**) 을 이용해서 가장 정확도가 우수한 알고리즘 및 파라미터 조합을 선택한다.
- 그리고 그 결과를 기존 데이터 셋의 인덱스와 함께 저장한다.

**“Bayesian Optimization 방법으로 어떻게 최적의 알고리즘 및 파라미터 조합을 찾는다라는 거지?”**

특정 데이터 셋에 대해,

- 모든 알고리즘 및 파라미터 조합을 토대로 각각 모델링을 진행했고,
- 그 모델이 주는 정확도를 역시 각각 기록했다고 가정해보자.

이 알고리즘 및 파라미터 조합 & 정확도 ‘들’은 아래와 같은 그래프로 표현될 수 있다.




특정 데이터 셋의 '모든 알고리즘 & 파라미터 조합'에 따른 '정확도'

결국 이 그래프를 안다라는 것은,

- 적어도 특정 데이터 셋에 대해 어떤 알고리즘 및 파라미터 조합이 가장 정확도가 우수한지? 를 알 수 있다라는 의미 또는,
- 이 데이터 셋과 유사한 데이터 셋에 대해서도 가장 정확도가 우수한 알고리즘 및 파라미터 조합을 찾아줄 수 있다.

라고 해석할 수 있다.

때문에 "특정 데이터 셋에 대해 이 그래프를 그릴 수 있는 어떤 함수"를 찾을 수만 있다면, 이 데이터 셋을 가장 높은 정확도로 분류 혹은 예측해 낼 수 있는 알고리즘 및 파라미터 조합을 찾을 수 있다.

**"그런데 이 함수를 어떻게 구할 수 있을까?"**

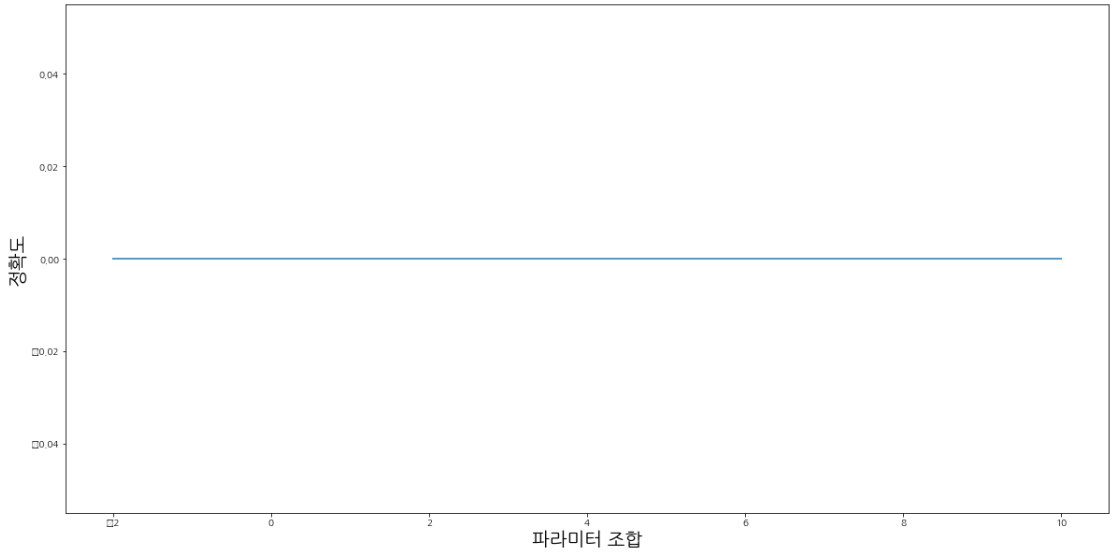
함수를 설명하는 어떤 식이 있다면 그 식을 한꺼번에 구해내기란 여간 어려운 일이 아니다. 때문에, Bayesian Optimization 방법은 아래와 같은 '전략' 을 활용한다.

- 이 함수는 어떤 기본 모양을 가지고 있다. (그게 어떤 직선이든, 2차 곡선이든 ..., 물론 이건 도메인에 따라서 결정할 수 있다., 그래서 Prior Distribution 이라는 말을 쓰기도 한다.)

- 이 기본 모양이 최종적인 이 함수를 얼마나 잘 대변하고 있는지, 관측치를 하나, 둘 넣어본다. 당연히 잘 맞지 않겠지? 그래서 이 관측치들을 중심으로 기본 모양을 변화시켜준다.
- 임의의 관측치에 대한 오차가 일정 이하의 수준으로 떨어질 때까지 위의 과정을 반복한다.

자 그럼 이 전략을 "특정 데이터 셋에 대해, 모든 알고리즘 및 파라미터 조합에 대한 정확도를 알 수 있는 함수" 를 구하는 데에 사용해보자.

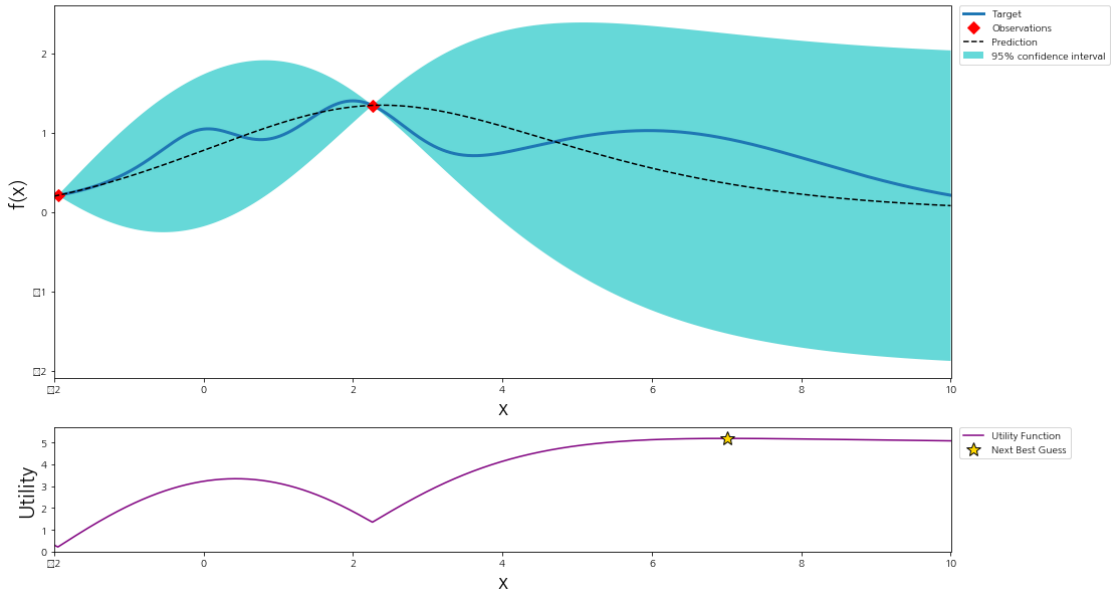
먼저 '어떤 알고리즘 및 파라미터 조합'에 대해서도 '정확도'가 0인 그래프가 있다고 가정해보자. 그리고 이 그래프를 우리가 구하고자 하는 함수의 기본모양으로 활용하자.




어떤 알고리즘 & 파라미터에 대해서도 정확도 = 0 인 그래프

**"그리고 2개의 관측치 (알고리즘 및 파라미터 조합과 그 때의 정확도) 를 반영하면서 기본 모양을 변화시켜보자"**

2개의 관측치를 토대로 그래프의 기본모양은 아래와 같이 '업데이트' 된다.




2개의 관측치가 반영된 함수의 모양

갑자기 그래프가 2 종류가 나오는데, 여기서는 위의 그래프만 '일단' 보기로 하자. 그래프는 아래와 같은 구성요소로 이루어져 있다.

- 빨간색 점 : 2개의 관측치
- 파란색 선 : 우리가 알고 싶은 함수
- 점선 : 두 관측치를 토대로 임의의 알고리즘 및 파라미터 조합 & 정확도를 예측한 선 (다시 말해, 두 관측치를 토대로 우리가 알고 싶은 함수를 추정한 선)
- Cyan 색으로 칠해진 영역 : 각 알고리즘 및 파라미터 조합에 대해 '정확도'가 예측될 수 있는 영역 (점선으로 표현된 값은 예측된 정확도들의 평균, 이 영역은 95%의 신뢰구간이라고 보면 된다)

역시 2개의 관측치 (알고리즘 및 파라미터 조합 & 그 때의 정확도) 를 이용해, 함수 전체의 모양이 잘 추정되지 않는 것은 사실이다.

- 그래서 한 개의 관측치를 더 추가해서, 함수 모양을 다시 변화시켜보는 행위를 해 볼 수 있다.

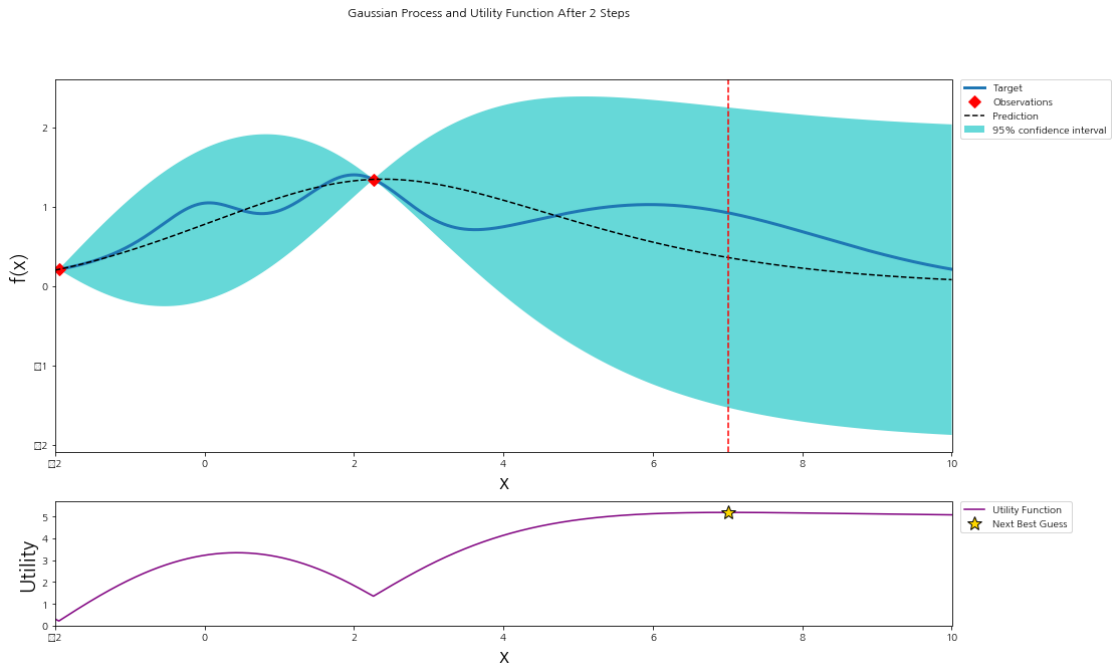
- 이 때, 어떤 관측치를 더 추가해야 소위 ROI (Return of Investment) 가 나오게 될까?

**“가장 점선이 출렁일 수 있는 가능성이 큰 지점을 관측하는 게 어떨까?”**

당연히 그러는게 좋을 거 같다.

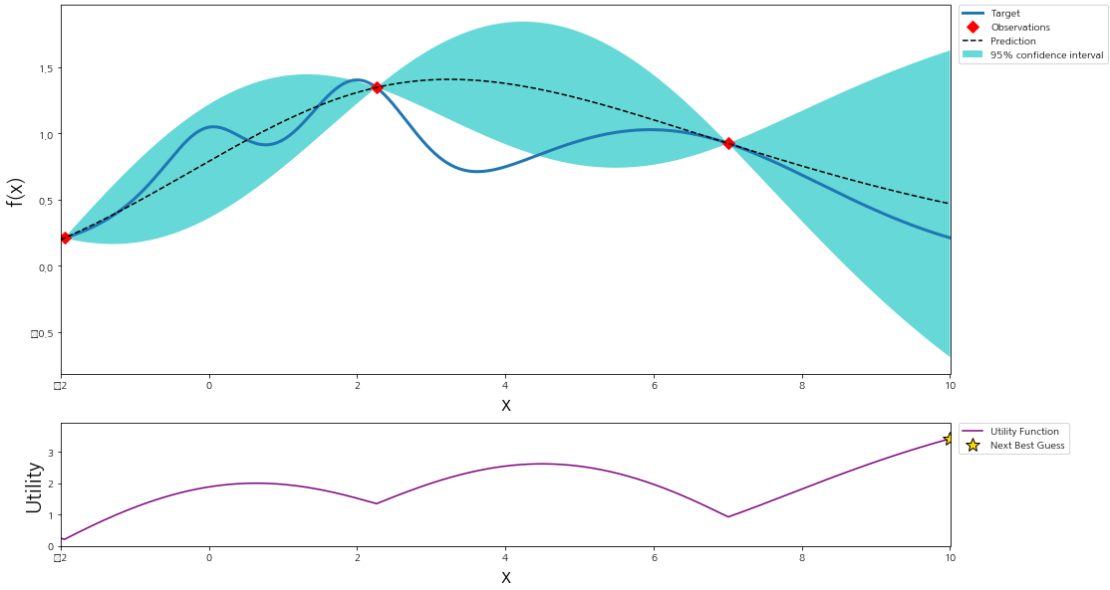
함수의 모양이 다양하게 변할 수 있는 부분을 먼저 관측하는 것이, 함수의 모양을 딱 잡아나가는 데에 가장 큰 기여를 할 테니까 말이다.

다시 말해, 위의 그래프에서 Cyan 영역이 위 아래로 가장 긴 지점을 고르자라는 말!, 바로 아래와 같이 빨간 색 점선이 그려진 지점 말이다.




어떤 지점을 우선 관측해야 할까? 바로 함수의 모양이 가장 많이 변화할 수 있는 지점! 바로 그 지점이다.

자 그럼 이 빨간 색 점선에 해당하는 관측치(알고리즘 및 파라미터 조합 & 정확도)를 구해서 함수 모양을 다시 ‘업데이트’ 해보자.




새로운 관측치를 통해 업데이트 된 함수의 모양

오! 2 개의 관측치로 그렸을 때보다 우리가 원하는 함수의 모양에 가까워 진 함수의 모양을 볼 수 있다.

결국 이런 식으로 함수의 모양을 추정하다보면 몇 개의 지점에 대한 관찰 만으로도 '특정 데이터 셋의 정확도를 가장 높일 수 있는 알고리즘 및 파라미터 조합을 발견하게 하는 함수' 를 '잘 추정' 해낼 수 있게 된다.

또한 이렇게 '특정 데이터 셋 별로 잘 추정된 함수'를 이용해서 그 데이터 셋을 높은 정확도로 분류 및 예측해줄 수 있는 알고리즘 및 파라미터를 찾을 수 있다라는 말!

그리고 auto-sklearn 은 기존 데이터 셋에 대해 높은 정확도를 가지는 알고리즘 및 파라미터 조합을 찾아서 기록해두는 것이고 말이다.

자, 그럼 이제 마지막 궁금증을 풀어볼 차례이다.

**"유사한 기존 데이터들에 매칭된 알고리즘 및 파라미터들을 어떻게 조합할까?"**

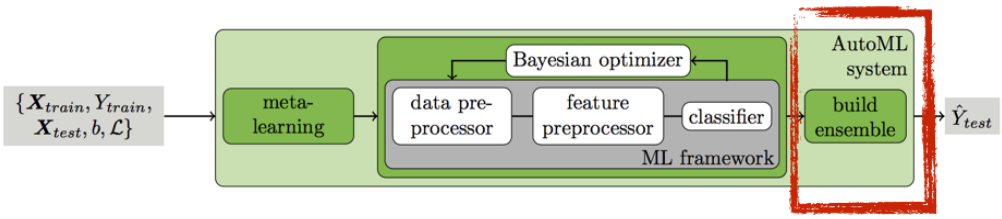
논문에서는 아래와 같이 '단일 모델 보다 모델들을 앙상블해서 이용하는 것이 훨씬 효과적이다'라고 주장하고 있다.

```
It is well known that ensembles often outperform
individual models, and that effective ensembles
can be created from a library of models.
```

때문에 auto-sklearn 에서도,

- 알고리즘 및파라미터 조합으로 이루어진 개별 모델들을 추천한 후,
- 그것들을 앙상블해서 최종모델을 만드는

흐름을 탄다.




Ensemble 과정

**"앙상블은 어떻게 할까?"**

추천된 알고리즘 및 파라미터 조합에 대한 앙상블은 Ensemble selection from libraries of models 에서 소개한 아래의 단계들을 토대로 진행하고 있다.

- 추천된 "알고리즘 및 파라미터 조합"을 가지는 모델들의 정확도를 구한다.
- 이 정확도를 기반으로 정렬해서 이 모델 들 중 Top N 을 고른다.
- 이 Top N의 모델들을 하나 씩 합치면서 정확도를 본다.
  ("합친다" = 합쳐지는 두 모델이 각각 예측한 확률의 평균을 구한 후, 그 평균 확률 스코어를 기반으로 제대로 예측을 하는

지를 평가하는 과정, 참고로 Top N 의 모델들은 중복되어서 합칠 수 있다.)

- 예측 정확도를 본다. 그리고 Top N 의 모델들을 합치는 과정으로 다시 돌아간다. (정해진 반복의 수 혹은 프로세싱 시간 등으로 루프를 제한할 수 있다.)

정기예금상품 가입 데이터로, 예를 들어보자.

이 데이터와 유사한 기존 데이터가 가지고 있는 알고리즘 및 파라미터 조합들을 추천 받은 그 상황을 떠올려보자.

```
Configuration:
  balancing:strategy, Value: 'none'
  categorical_encoding:__choice__, Value: 'no_encoding'
  classifier:__choice__, Value: 'random_forest'
  classifier:random_forest:bootstrap, Value: 'True'
  classifier:random_forest:criterion, Value: 'gini'
  classifier:random_forest:max_depth, Constant: 'None'
  classifier:random_forest:max_features, Value: 0.5
  classifier:random_forest:max_leaf_nodes, Constant: 'None'
  classifier:random_forest:min_impurity_decrease, Constant: 0
  classifier:random_forest:min_samples_leaf, Value: 1
  classifier:random_forest:min_samples_split, Value: 2
  classifier:random_forest:min_weight_fraction_leaf, Constant: 0
  classifier:random_forest:n_estimators, Constant: 100
  imputation:strategy, Value: 'mean'
  preprocessor:__choice__, Value: 'no_preprocessing'
  rescaling:__choice__, Value: 'minmax'

-------------------------------------------------


Configuration:
  balancing:strategy, Value: 'none'
  categorical_encoding:__choice__, Value: 'no_encoding'
  classifier:__choice__, Value: 'adaboost'
  classifier:adaboost:algorithm, Value: 'SAMME'
  classifier:adaboost:learning_rate, Value: 0.723185509
  classifier:adaboost:max_depth, Value: 5
  classifier:adaboost:n_estimators, Value: 357
  imputation:strategy, Value: 'mean'
  preprocessor:__choice__, Value: 'extra_trees_preproc_for_classification'
  preprocessor:extra_trees_preproc_for_classification:bootstrap, Value: 'True'
  preprocessor:extra_trees_preproc_for_classification:criterion, Value: 'gini'
  preprocessor:extra_trees_preproc_for_classification:max_depth, Constant: 'None'
  preprocessor:extra_trees_preproc_for_classification:max_features, Value: 0.5
  preprocessor:extra_trees_preproc_for_classification:max_leaf_nodes, Constant: 'None'
  preprocessor:extra_trees_preproc_for_classification:min_impurity_decrease, Constant: 0
  preprocessor:extra_trees_preproc_for_classification:min_samples_leaf, Value: 5
  preprocessor:extra_trees_preproc_for_classification:min_samples_split, Value: 8
  preprocessor:extra_trees_preproc_for_classification:min_weight_fraction_leaf, Constant: 0
  preprocessor:extra_trees_preproc_for_classification:n_estimators, Constant: 100
  rescaling:__choice__, Value: 'standardize'
```

추천된 알고리즘 및 파라미터 조합들

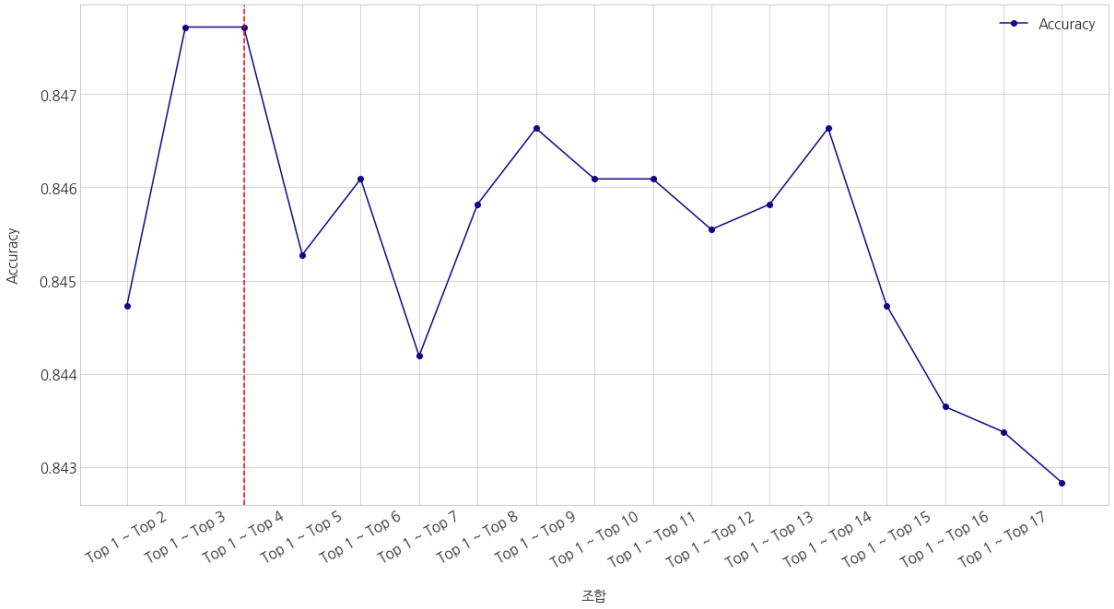
그리고 이 알고리즘 및 파라미터 조합이 탑재된 모델들에 대한 정확도를 각각 구한다면 아래와 같다.

| AUC | Accuracy | Precision | Recall | idx | predictions |
|---|---|---|---|---|---|
| 0.846700 | 0.844463 | 0.804531 | 0.883169 | 0 | [0.33, 0.35, 0.64, 0.28, 0.53, 0.03, 0.03, 0.0... |

| | | | | | |
|---|---|---|---|---|---|
| 0.824989 | 0.823833 | 0.793798 | 0.843840 | 1 | [0.493048365171, 0.483201026368, 0.53894920682... |
| 0.846554 | 0.844734 | 0.808863 | 0.876229 | 2 | [0.456061705089, 0.342925733862, 0.79545112740... |
| 0.843475 | 0.840934 | 0.798122 | 0.884905 | 3 | [0.387922087822, 0.305746574318, 0.76504001612... |
| 0.841137 | 0.840119 | 0.812158 | 0.857721 | 4 | [0.348978880343, 0.369038737472, 0.77697798687... |

알고리즘 및 파라미터 조합 별 정확도

자, 이제 이 정확도를 높은 순으로 정렬한 후 각 모델들을 Top 1 부터 ... 주욱 차례 차례 붙여보면서 정확도를 구해보는 거다. 그리고 Top 1 의 정확도보다 뛰어난 조합은 어떤 것인지 확인해보자.




Top 1 ~ ..., 차례대로 붙여가면서 구한 정확도

- Top 1 ~ Top 4 를 sequential 하게 조합한 결과가 가장 좋다고 나온다.
- 그리고 그 정확도는 .84771 이고,
- Top 1의 정확도인 .84473 를 넘어서는 결과를 보이고 있다.

그런데, 이 결과는 알고리즘 및 파라미터의 조합에 대한 중복 추출 및 결합을 고려하지 않은 결과이다.
(물론, 그럼에도 불구하고 Top 1을 이기는 괜찮은 조합을 찾긴 했지만)

**"중복 조합을 허용할 경우는?"**

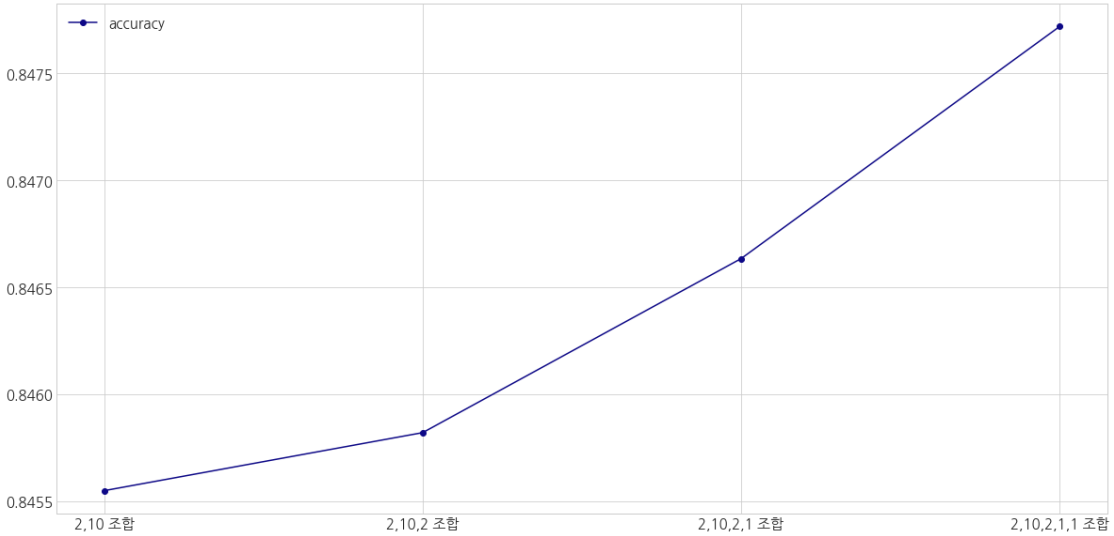
즉, 다음의 과정을 타게 되는데

- Top 1 모델의 확률 스코어와 Random 하게 고른 모델의 확률 스코어와의 평균값을 계산하고,
- 그 값을 토대로 정확도를 산출하고,
- 그 정확도가 기존에 높았던 정확도를 넘는 지를 확인한다. (그래서 높다면 정확도 및 그 정확도를 구할 수 있게 하는 앙상블 시퀀스를 기록한다)
- 이 과정을 N 번 반복한다. (여기서 N = 1000)

예로 든 데이터로 이 과정을 타게 되면,

- 2번 인덱스 (Top 1) + 10 번 인덱스 (Top 9) + 3개의 1번 인덱스 (Top 17)의 조합의 정확도가 .847로 가장 뛰어나다.

라는 결론을 얻게 된다.




중복 조합으로 얻은 정확도

**“그래서 얻은 최종 모델은 어떻게 되는거지?”**

위의 방법을 이용해서 얻은 모델은 아래와 같이 표현할 수 있다.

- 총 모델 = 2번 인덱스의 알고리즘 및 파라미터가 적용된 모델 * 0.4 + 10번 모델 * 0.2 + 1번 모델 * 0.4

(참고로 여기서 weight는 등장한 모델의 개수 / 총 모델의 개수이다.)

## 지금까지 한 이야기를 정리해보자

굉장히 길었지만, 이 글은 아래와 같이 정리할 수 있다.

- 풀려는 문제와 데이터에 따라, 알고리즘을 선택하고 그 알고리즘에 입력해주어야 하는 파라미터 들을 튜닝하는 과정은 굉장히 Time-Consuming 하는 소모적인 과정이다.
- 때문에 적당한 정확도가 보장되는 모델을 자동으로 만드는 방법이 없을까? 라는 고민을 했다.
- 그런데, 데이터의 특성에 따라 '선택하면 좋을 알고리즘 및 파라미터를 찾는 문제' 는 이미 연구되어 왔고 이를 AutoML 이라고 일컫고 있다라는 것
- 이 AutoML 의 구현체는 다양한데, 그 중 Auto-Sklearn 을 돌려보고 그 원리를 파악해봤다.

파악해 본 Auto-Sklearn 은,

- OpenML 의 데이터를 이용해 각 데이터의 특성에 맞는 최적의 알고리즘 및 파라미터 조합을 미리 계산해두고,
- 새로운 데이터가 들어왔을 때 이 데이터의 특성을 추출해서,
- 가장 유사한 특성을 가진 '기존의 데이터 들'을 뽑아, 그 데이터들에 매칭되어 있는 최적의 알고리즘 및 파라미터 조합을 알려준다.
- 그리고 그 알고리즘 및 파라미터 조합들을 '앙상블 기법' 을 활용해서 조합해주는 과정을 토대로 최종 모델을 만들어 준다.

## References


- Feurer et al., Efficient and Robust Automated Machine Learning, Advances in Neural Information Processing Systems 28 (NIPS 2015).

- J. Snoek, H. Larochelle, and R. P. Adams. Practical Bayesian optimization of machine learning algorithms. In Proc. of NIPS'12, pages 2960–2968, 2012.
- R. Caruana, A. Niculescu-Mizil, G. Crew, and A. Ksikes. Ensemble selection from libraries of models. In Proc. of ICML'04, page 18, 2004.
- https://github.com/automl/auto-sklearn
- https://github.com/fmfn/BayesianOptimization


## Code

이 글에서 소개된 그래프 및 여러 실험에 대한 코드는 여기에서 확인할 수 있다.