FLAML: A **F**ast **L**ibrary for **A**uto**ML** & Tuning

github.com/microsoft/FLAML

Chi Wang[1], Qingyun Wu[2], Susan Xueqing Liu[3], Luis Quintanilla[1]

1. Microsoft
2. Penn State University
3. Stevens Institute of technology

What Will You Learn

How to use FLAML to (1) find accurate ML models with low computational resources for common ML tasks; (2) tune hyperparameters generically

How to leverage the flexible and rich customization choices

Finish the last mile for deployment

Create new applications

Code examples, demos, use cases

Research & Development opportunities

Agenda

First half (9:30 AM – 11:10 AM)

- **Overview of AutoML and FLAML (9:30 AM)**
- Task-oriented AutoML with FLAML (9:45 AM)
- ML.NET demo (10:30 AM)
- Tune user defined functions with FLAML (10:40 AM)

Break, Q & A

Second half (11:20 AM – 12:30 AM)

- Zero-shot AutoML (11:20 AM)
- Time series forecasting
- Natural language processing (11:45 AM)
- Online AutoML (12:00 PM)
- Fair AutoML
- Challenges and open problems


TURN TO CLEAR VISION
QUARTERS ONLY
50¢
TO OPERATE TURN HANDLE ONE FULL TURN
2115

Overview

7 Cross-industry Disruptive Trends in Next Few Decades – McKinsey (2021)

- Applied AI

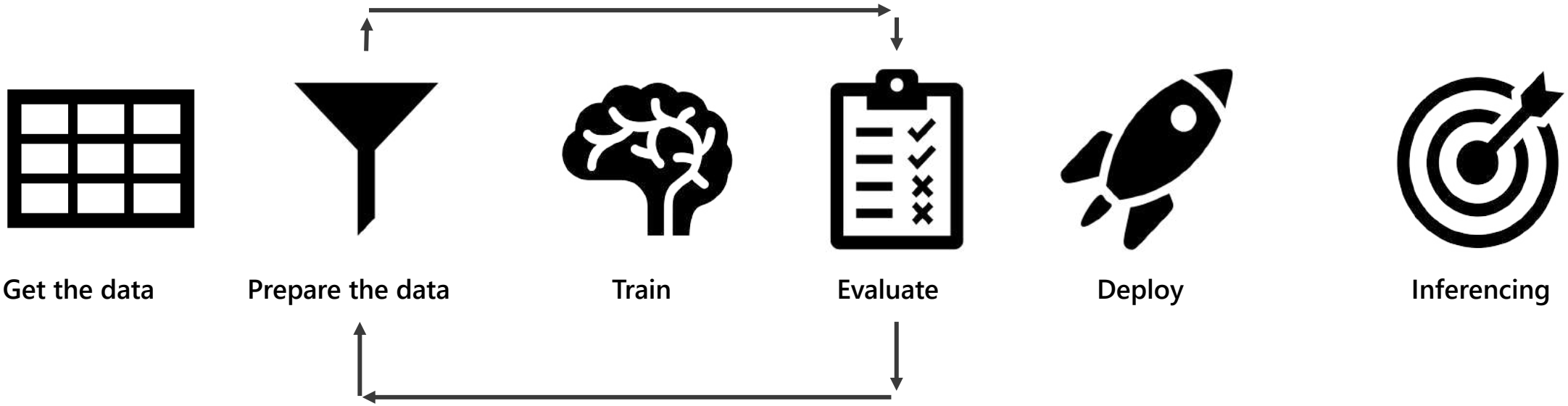
  >75% of all digital-service touch points will see improved usability, enriched personalization, and increased conversion

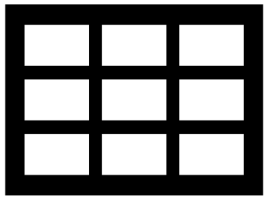
- Future of programming

  ~30x reduction in the working time required for software development and analytics

Companies need to master MLOps to make the most of both trends

Machine Learning Workflow





Get the data

Lots of decisions to make to optimize model performance

- Select learner
- Tune hyperparameters
- …

Deploy

Inferencing

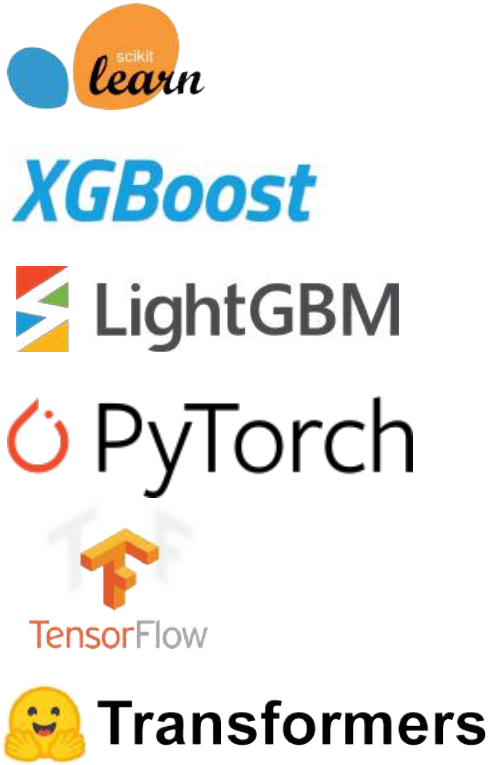
Model Training

Model Consumption

Lots of decisions to make:

- **Select learner**
- Tune hyperparameters
- …

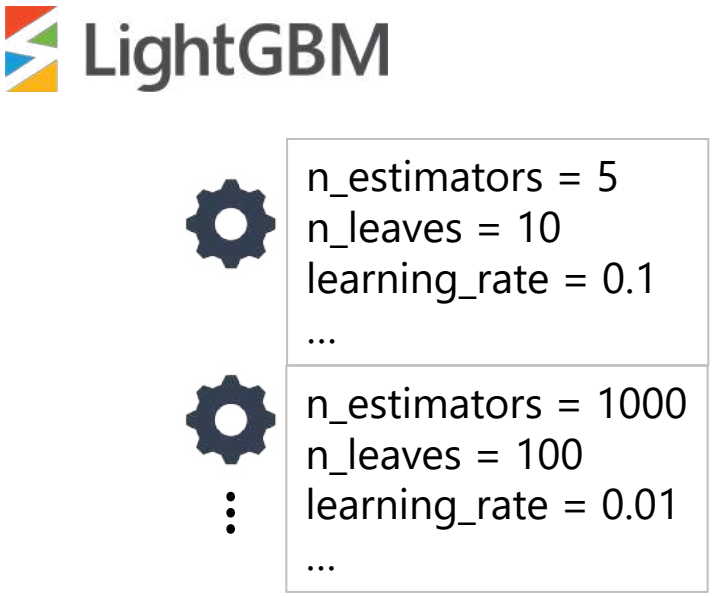
Optimize model performance



Lots of decisions to make:

- Select learner
- **Tune hyperparameters**
- …

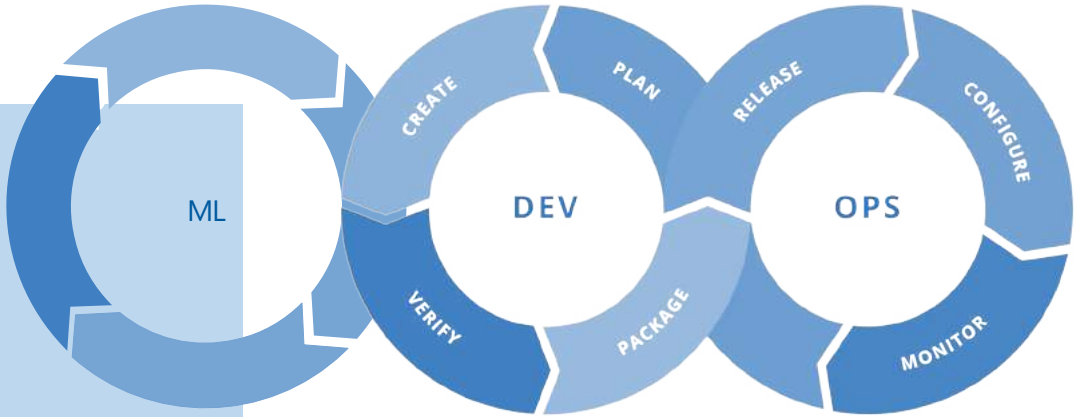
Optimize model performance

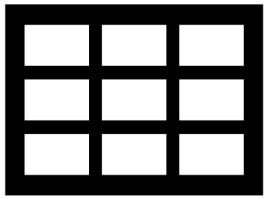


Lots of decisions to make:

- Select learner
- Tune hyperparameters
- …

Optimize model performance





Get the data

Lots of decisions to make:
- Select learner
- Tune hyperparameters
- …

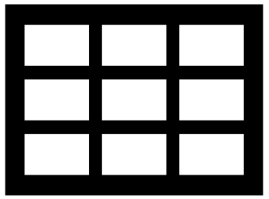
Optimize model performance

Deploy

Inferencing

Model Training

Model Consumption



Get the data

AutoML

Deploy

Inferencing

Model Training

Model Consumption

	AutoML	Autonomous data platform
Market Size Forecast in 2030	$14.8 Billion	$4.8 Billion

Benefits of AutoML

- Enables and empowers novices
- Standardizes the ML workflow for better reproducibility, code maintainability, knowledge sharing
- Prevents suboptimal results due to idiosyncrasies of ML Innovators
- Builds models more effectively and efficiently
- Enables rapid prototyping
- Fosters learning

[Xin et al. 2021] Xin, D., Wu, E. Lee, Y., Salehi N., and Parameswaran, A. Whither AutoML? Understanding the Role of Automation in Machine Learning Workflows. CHI 2021.

Deficiencies of AutoML

- Lacks comprehensive end-to-end support
- Causes system failures due to compute intensive workloads
- Lacks customizability
- Lacks transparency and interpretability

Deficiencies of AutoML

- Lacks comprehensive end-to-end support
- **Causes system failures due to compute intensive workloads**
- Lacks customizability
- Lacks transparency and interpretability

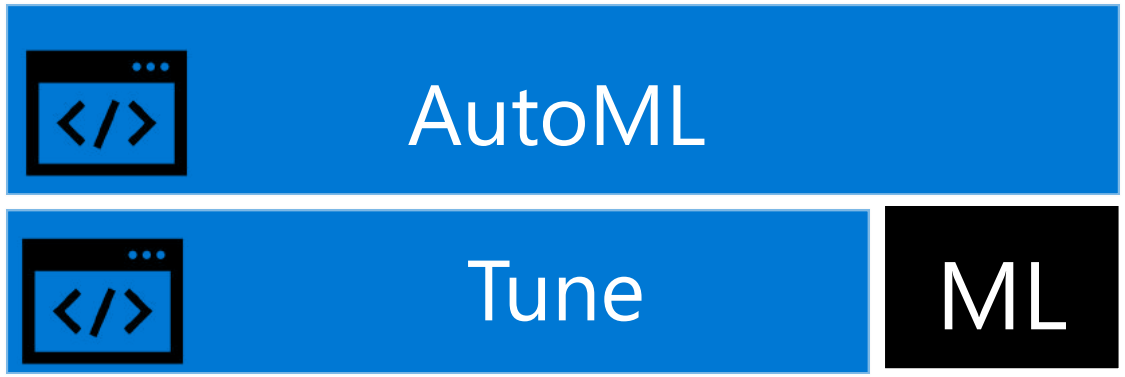
No usable model returned
Revert to manual development

Deficiencies of AutoML

- Lacks comprehensive end-to-end support
- Causes system failures due to compute intensive workloads
- **Lacks customizability** (Or users need to make too many hard choices)
- Lacks transparency and interpretability

Deficiencies of AutoML

- Lacks comprehensive end-to-end support → Offer both AutoML and Tune API & **extensible**
- **Causes system failures due to compute intensive workloads** → **Fast** 
- **Lacks customizability** → **Economical**  **Smooth customization**
- Lacks Transparency and Interpretability



FLAML

Offer both AutoML and Tune API & **extensible**

Fast

Economical

Smooth customization

Python Library *flaml*

- Task-oriented AutoML

```
from flaml import AutoML
automl = AutoML()
automl.fit(X_train, y_train, task="classification", time_budget=60)
```

Customization is easy

```
automl.add_learner("mylgbm", MyLGBMEstimator)
automl.fit(X_train, y_train, task="classification", metric=custom_metric, estimator_list=["mylgbm"])
```

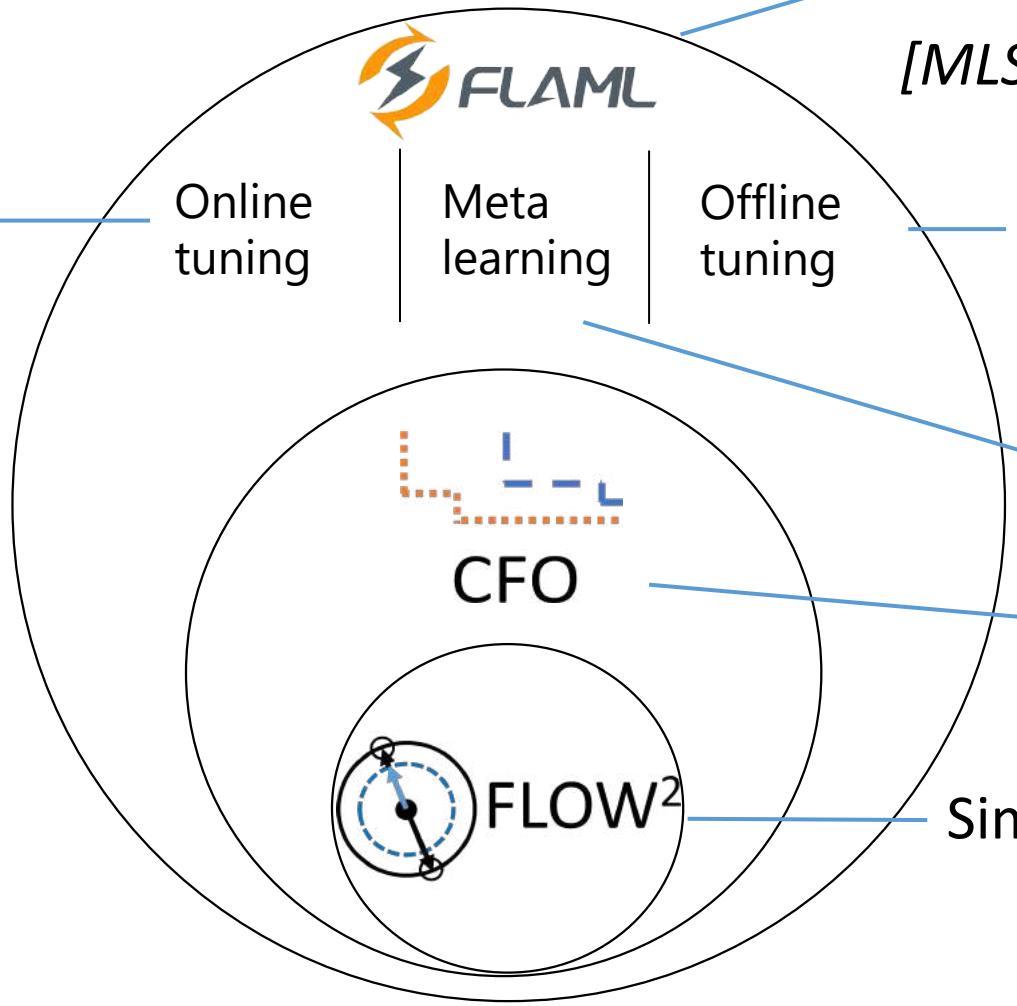
- Tune user-defined function

```
analysis = tune.run(
    train_lgbm, metric="mse", mode="min", config=config_search_space,
    low_cost_partial_config=low_cost_partial_config, time_budget_s=3, num_samples=-1,
)
```

Economical Tuning & AutoML

✓*Leverage joint impact of multiple factors on* ***cost & error***

- ✓*Hyperparameter*
- ✓*Learner*
- ✓*Sample size*
- ✓*Resampling strategy: CV or Holdout*



Fast library for AutoML and tuning *[MLSys'21]*

ChaCha (Champion-Challenger) *[ICML'21]*

BlendSearch (local + global search) *[ICLR'21]*

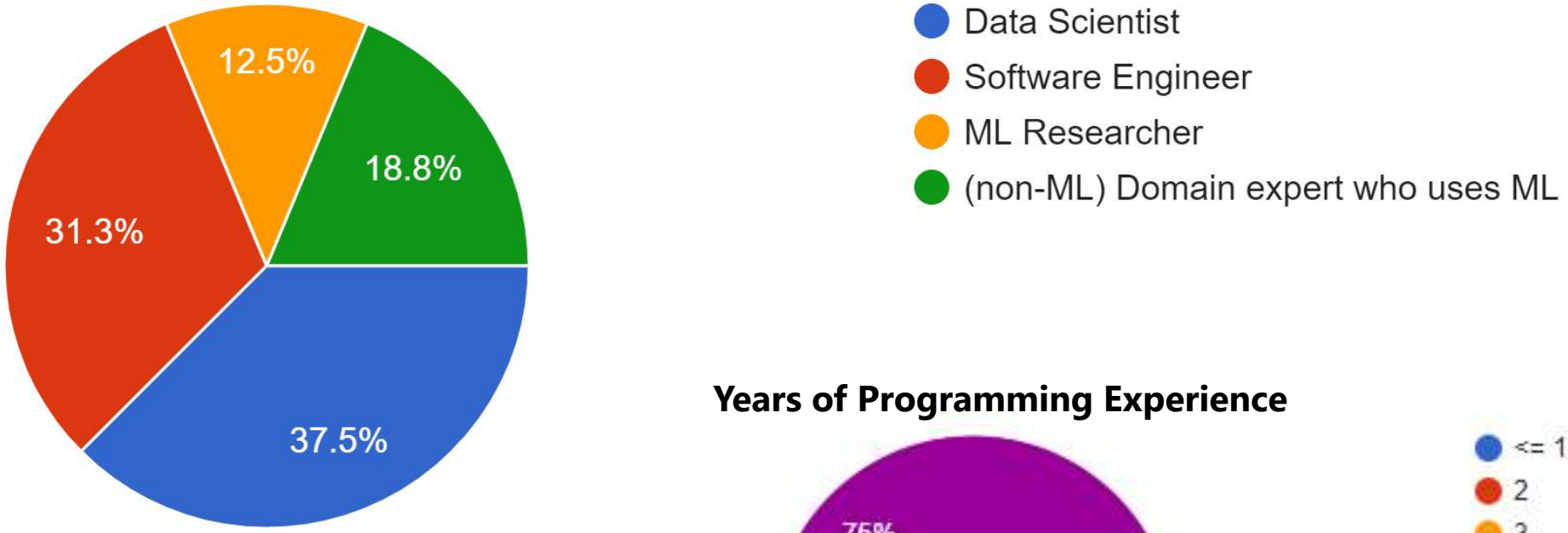
Zero-shot *[KDD-AutoML'22]*

Cost-frugal optimization *[AAAI'21]*

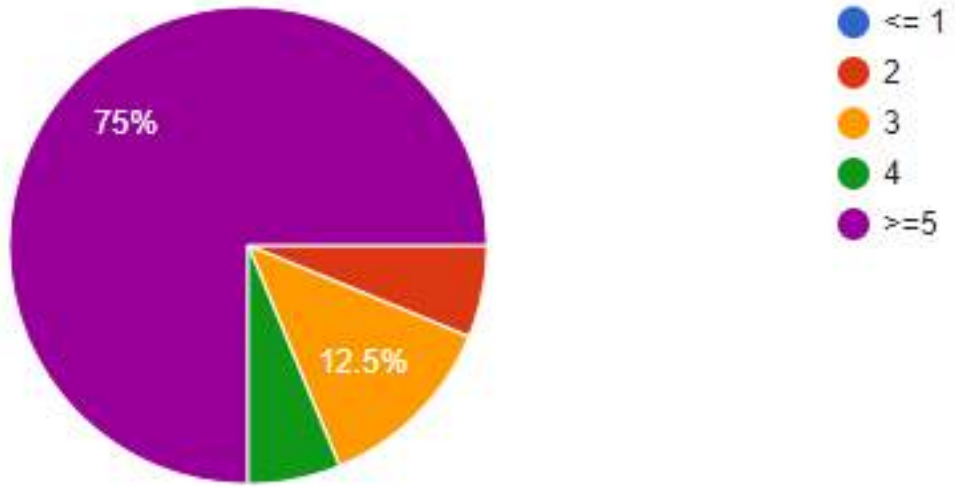
Simple and provable local search

Users' Background

Job Title



Years of Programming Experience



Example Use Cases

APPLICATION	WHO	FIELD
AutoML tool for .NET developers (Visual Studio, ML.NET)	Microsoft	SDE
Suspicious behavior detector	Microsoft	Security
Credit assessment	GLOBAL LENDING SERVICES	Finance
Auto finance functions (classification/regression)	WARBURG PINCUS	Finance
Hardware demand forecast	Microsoft	Supply chain
A/B testing with automated causal inference (auto-causality)	wise	CRM
Air quality estimate	MIT NASA ExxonMobil	Science
Pricing	wtw	Insurance

- Impact: Accuracy, Productivity, R&D

Fast AutoML with FLAML + Ray Tune

Microsoft Researchers have developed FLAML (Fast Lightweight AutoML) which can now utilize Ray Tune for distributed hyperparameter tuning to scale up FLAML's resource-efficient & easily parallelizable algorithms across a cluster

Michael Galarnyk Aug 25 · 9 min read

[Blog Post in *Towards Data Science*]

Agenda

First half (9:30 AM – 11:10 AM)

- Overview of AutoML and FLAML (9:30 AM)
- **Task-oriented AutoML with FLAML (9:45 AM)**
- ML.NET demo (10:30 AM)
- Tune user defined functions with FLAML (10:40 AM)

Break, Q & A

Second half (11:20 AM – 12:30 AM)

- Zero-shot AutoML (11:20 AM)
- Time series forecasting
- Natural language processing (11:45 AM)
- Online AutoML (12:00 PM)
- Fair AutoML
- Challenges and open problems

Overview of AutoML and Tune in FLAML

Task-oriented AutoML

Tune user-defined function

Agenda and Resources to Be Used in This Tutorial

First half of the tutorial:

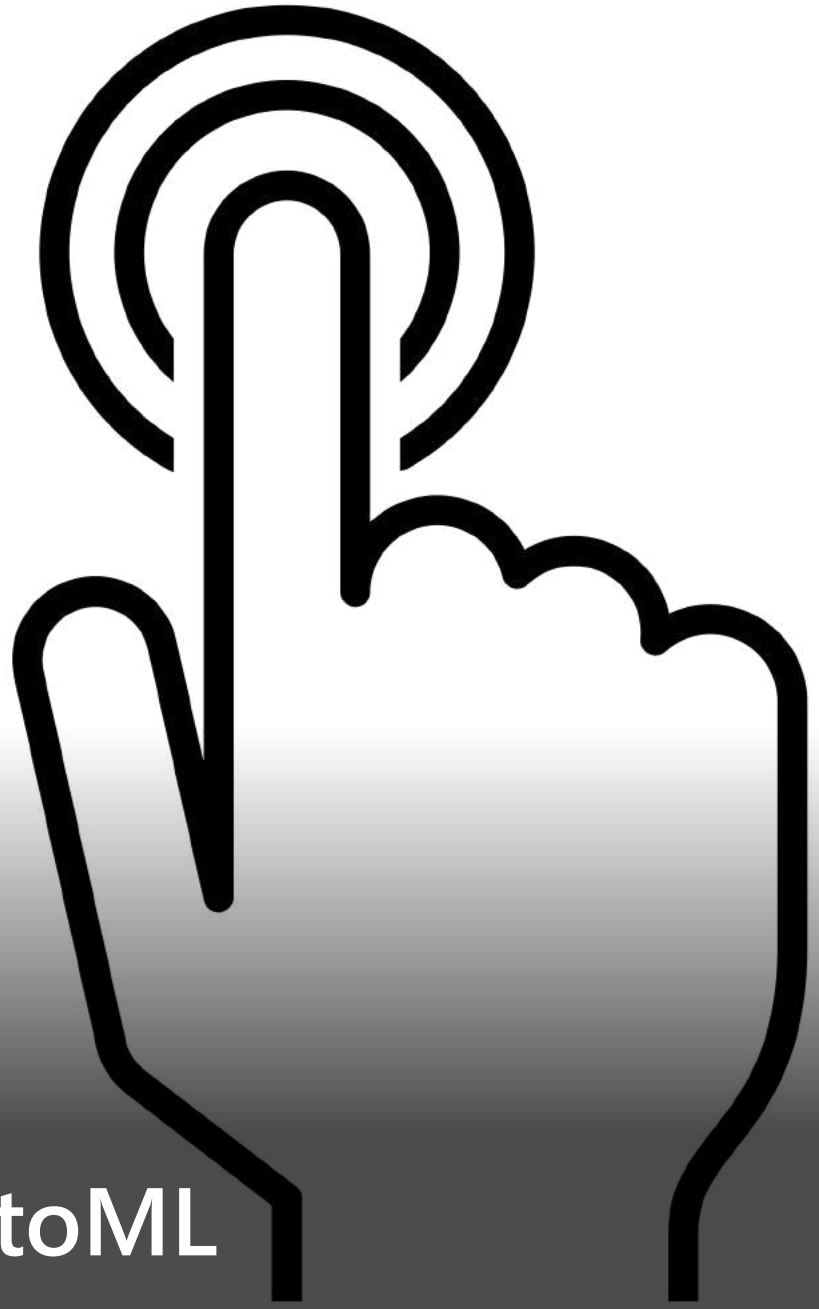
- Overview of AutoML and FLAML
- Task-oriented AutoML with FLAML
  - Notebook: A classification task with AutoML
  - Notebook: A regression task with AuotML using LightGBM as the learner
- ML.Net demo
- Tune user defined functions with FLAML
  - Notebook: Basic tuning procedures and advanced tuning options
  - Notebook: Tune pytorch
- Q & A

Second half of the tutorial:

- Zero-shot AutoML
  - Notebook: Zeroshot AutoML
- Time series forecasting
  - Notebook: AutoML for Time Series Forecast tasks
- Natural language processing
  - Notebook: AutoML for NLP tasks
- Online AutoML
  - Notebook: Online AutoML with Vowpal Wabbit
- Fair AutoML
- Challenges and open problems

https://github.com/microsoft/FLAML

https://github.com/microsoft/FLAML/tree/tutorial/tutorial



Task-oriented AutoML

What Is Task-oriented AutoML

Inputs:

- Resource budget
- ML task
  - ✓ Training data
  - ✓ Task type

Deploy

Inferencing

Model Training

Model Consumption

Task-oriented AutoML: Supported Tasks

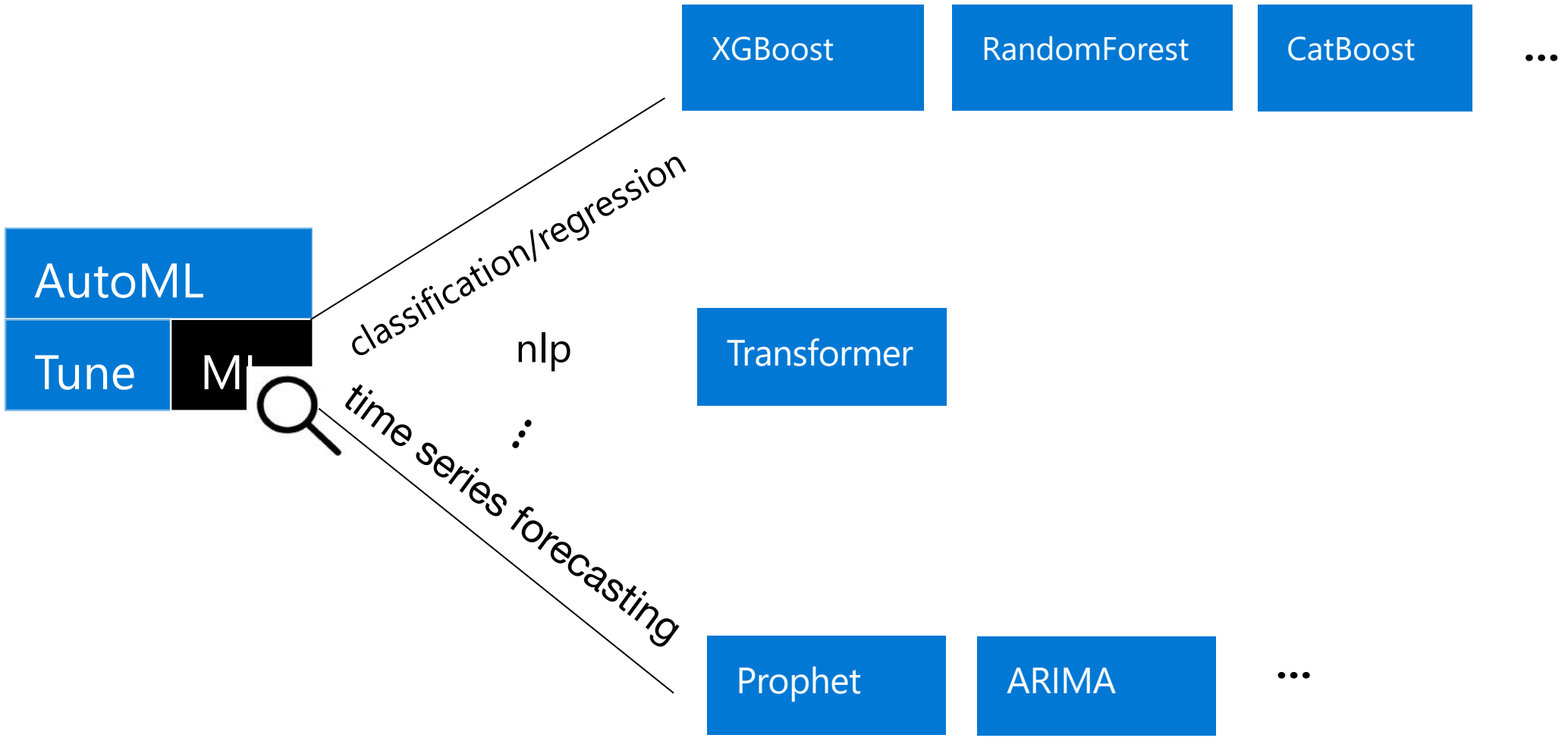
- Tasks (specified via `task`):
  - 'classification': classification.
  - 'regression': regression.
  - 'ts_forecast': time series forecasting.
  - 'ts_forecast_classification': time series forecasting for classification.
  - 'rank': learning to rank.
  - 'seq-classification': sequence classification.
  - 'seq-regression': sequence regression.
  - 'summarization': text summarization.
  - 'token-classification': token classification.
  - 'multichoice-classification': multichoice classification.

Required data format:
X: Numpy array or dataframe
y: Numpy array or series of labels in shape n*1.

Time series forecasting tasks

NLP tasks

Task-specific Built-in ML Estimators



Resources to Be Used

First half of the tutorial:

- Overview of AutoML and FLAML
- Task-oriented AutoML with FLAML
  - Notebook: A classification task with AutoML
  - Notebook: A regression task with AuotML using LightGBM as the learner
- ML.Net demo
- Tune user defined functions with FLAML
  - Notebook: Basic tuning procedures and advanced tuning options
  - Notebook: Tune pytorch
- Q & A

Second half of the tutorial:

- Zero-shot AutoML
  - Notebook: Zeroshot AutoML
- Time series forecasting
  - Notebook: AutoML for Time Series Forecast tasks
- Natural language processing
  - Notebook: AutoML for NLP tasks
- Online AutoML
  - Notebook: Online AutoML with Vowpal Wabbit
- Fair AutoML
- Challenges and open problems

https://github.com/microsoft/FLAML/tree/tutorial/tutorial

Task-oriented AutoML: A Basic Use Case

- Get data
- AutoML with FLAML
- AutoML Result

```
''' import AutoML class from flaml package '''
from flaml import AutoML
automl = AutoML()
```

```
'''The main flaml automl API'''
automl.fit(X_train=X_train, y_train=y_train, time_budget=60, task='classification')
```

Benefits of Task-oriented AutoML in FLAML

- Save manual efforts (human resource)
- **Effectiveness** and **efficiency**: use small computation resource to find models with good performance

Benefits of Task-oriented AutoML in FLAML

- Save manual efforts (human resource)
- **Effectiveness** and **efficiency**: use small computation resource to find models with good performance

First half of the tutorial:

- Overview of AutoML and FLAML
- Task-oriented AutoML with FLAML
  - Notebook: A classification task with AutoML
  - Notebook: A regression task with AuotML using LightGBM as the learner
- ML.Net demo
- Tune user defined functions with FLAML
  - Notebook: Basic tuning procedures and advanced tuning options
  - Notebook: Tune pytorch

Benefits of Task-oriented AutoML in FLAML

- Save manual efforts (human resource)
- **Effectiveness** and **efficiency**: use small computation resource to find models with good performance

Default LightGBM

```
default lgbm r2 = 0.8296179648694404
```

Optuna LightGBM Tuner

```
CPU times: user 5min 14s, sys: 16.8 s, total: 5min 31s
Wall time: 5min 32s
```

```
Optuna LightGBM Tuner r2 = 0.8444445782478855
```

FLAML's accuracy

```
print('flaml (4min) r2', '=', 1 - sklearn_metric_loss_score('r2', y_pred, y_test))
```

```
flaml (4min) r2 = 0.8505434326525669
```

Benefits of Task-oriented AutoML in FLAML

- Save manual efforts (human resource)
- **Effectiveness** and **efficiency**: use small computation resource to find models with good performance

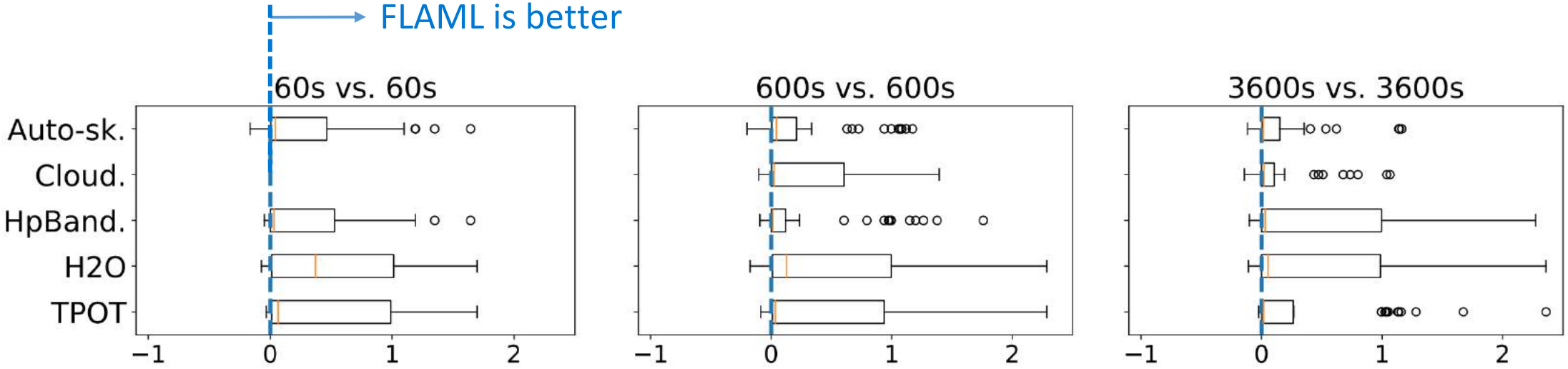


Figure 1. Box plot of scaled score difference between FLAML and other libraries when FLAML uses equal or smaller budget (**positive difference meaning FLAML is better**). *[MLSys'21]*

AutoML Benchmark

OpenML AutoML Benchmarking Framework

Benefits of Task-oriented AutoML in FLAML

- Save manual efforts (human resource)
- **Effectiveness** and **efficiency**: use small computation resource to find models with good performance
- Rich **customization choices** in FLAML

  Helpful in scenarios such as:

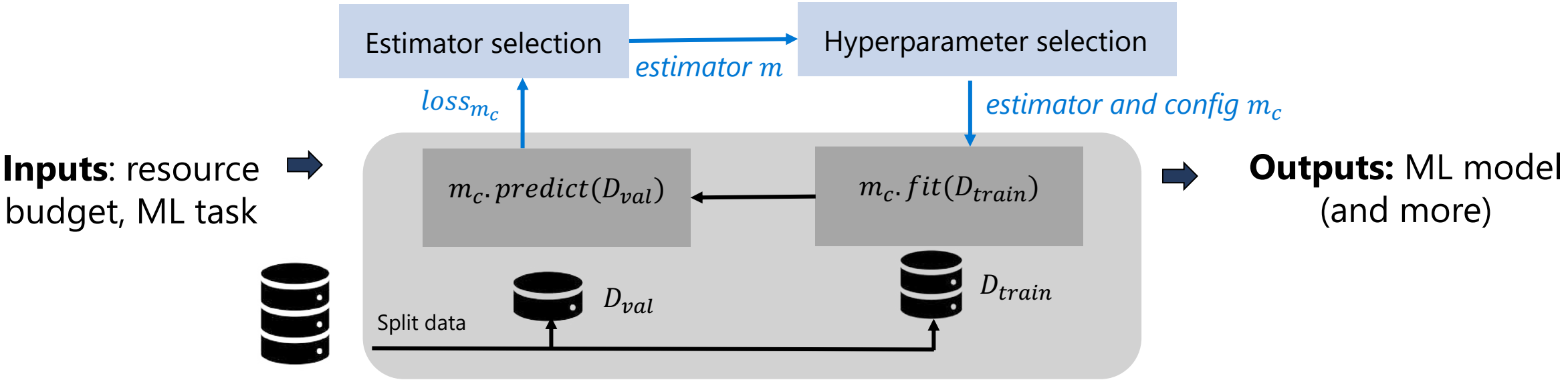
  Due to business/deployment requirements, you need to use

  - Special **ML learner** (e.g., a domain specific one), or/and
  - **Custom metrics,** or/and
  - **Various constraints** (e.g., in terms of computation resource, model complexity, inference time)

  …

Task-oriented AutoML: Overview

Empowered by our cost-frugal Hyperparameter Optimization (HPO) algorithms



The Cost-Frugal HPO Problem

Validation loss under hyperparameter configuration x.

$$\mathbf{x}^* = \arg\min_{\mathbf{x} \in \mathcal{X}} f(\mathbf{x})$$

Each time you query f(x), you need to pay a cost of $g(x)$

Hyperparameter configuration

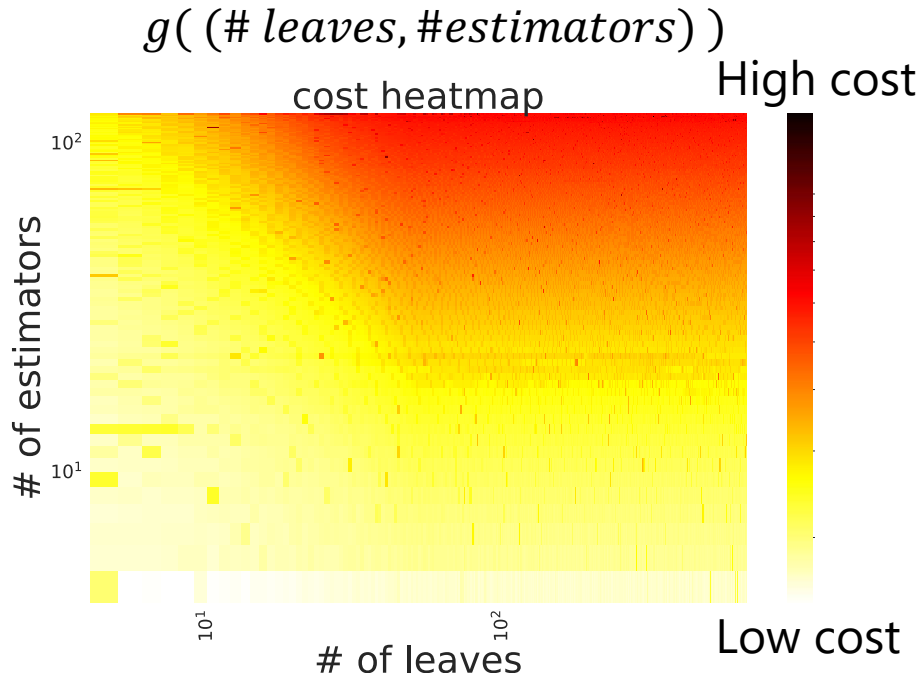
- Two important properties:
  - f(x) is a **black-box function**
  - Function value evaluation is **expensive**
- Vanilla HPO: find x^* with small number of iterations
- Cost-frugal HPO: find x^* while keeping total cost $\sum g(\boldsymbol{x_i})$ small

Insights On The Cost-Frugal HPO Problem

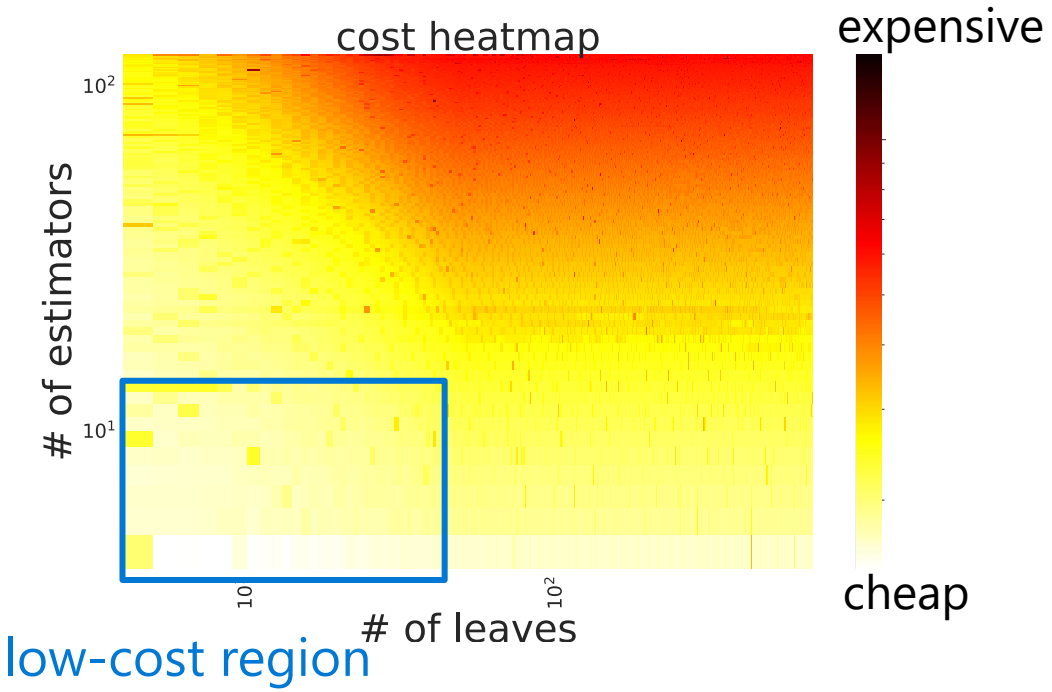
- Vanilla HPO: find x^* with small number of iterations
- Cost-frugal HPO: find x^* while keeping total cost $\sum g(\boldsymbol{x_i})$ small

1. If g(x) is constant, low cost ⇔ small #iterations
Bayesian Optimization optimizes for this case

2. It is common to encounter **cost-related hyperparameters**
(Examples: the number of estimators and leaves in gradient boosted trees; the number of layers and neurons in DNN)



Assumptions about Cost-related HPs



1. Lipschitz continuous

2. Easy to know low-cost configurations before we start the optimization

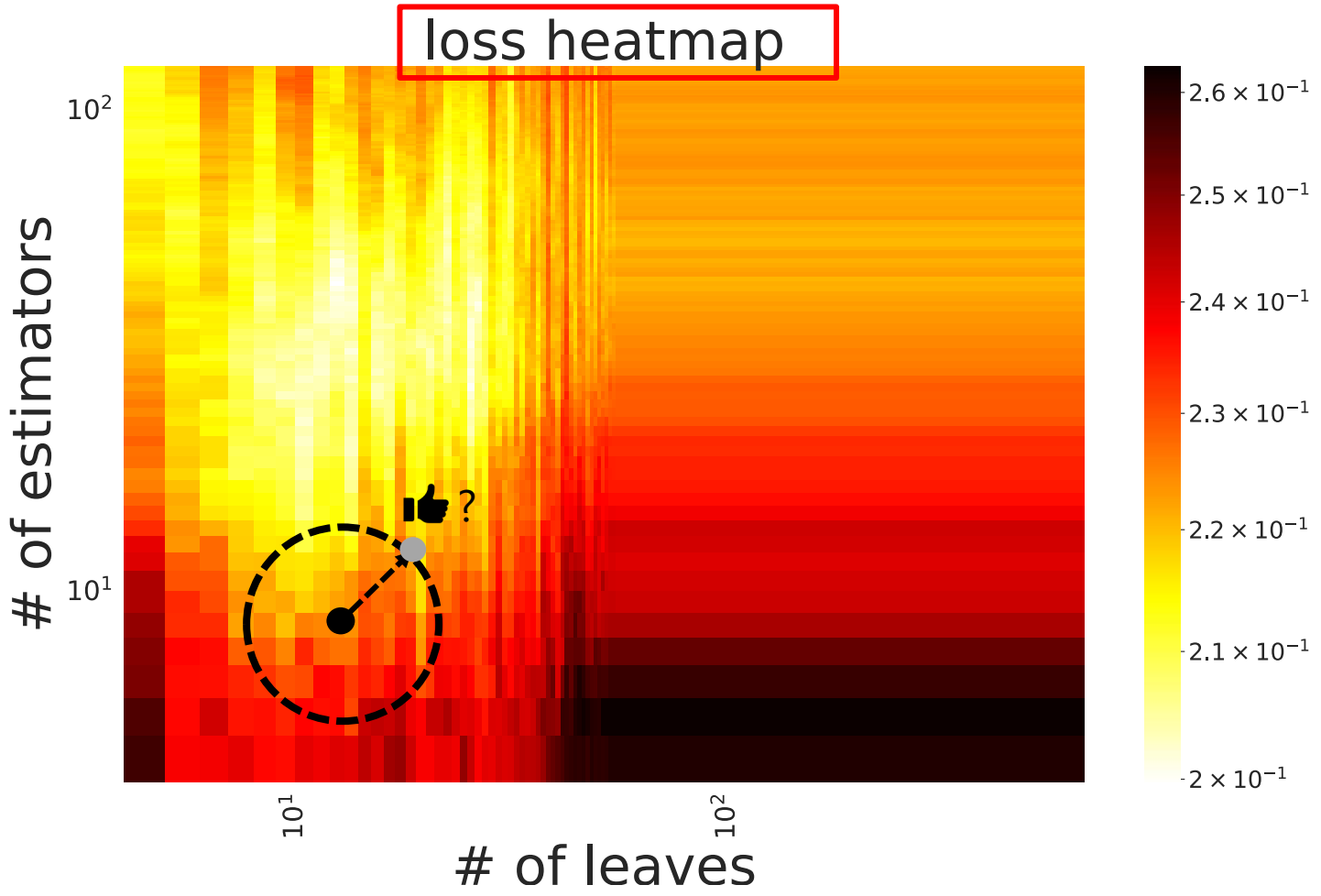
Cost-Frugal HPO Algorithm Design

- To avoid high-cost points until necessary -> Low-cost starting point + local search
- To find low-loss points -> Follow loss-descent directions
- Cannot directly use gradient-based method: no gradient available.

- Surprise: function values are enough to find the right directions for efficient convergence.

- More surprise: sign comparison between function values is enough!!

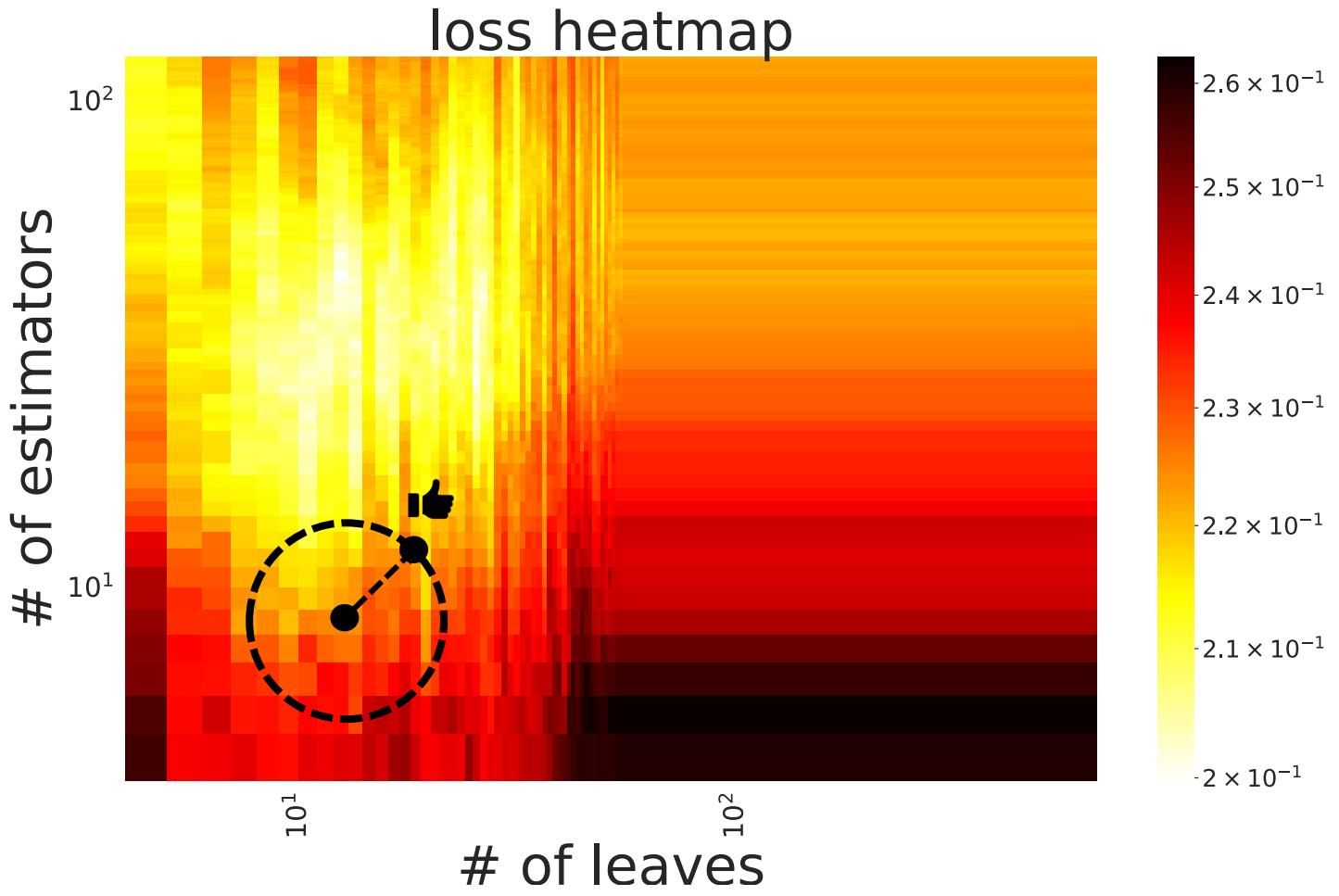
A Cost-Frugal HPO Algorithm (CFO)[*AAAI'21*]



Repeat the following steps after each *move*:

1. **Uniformly** sample a direction from **a local unit sphere;**
2. **Compare;**
3. ***Move*** *(and break)* or try the **opposite direction**;
4. ***Move*** *(and break)* or stay.

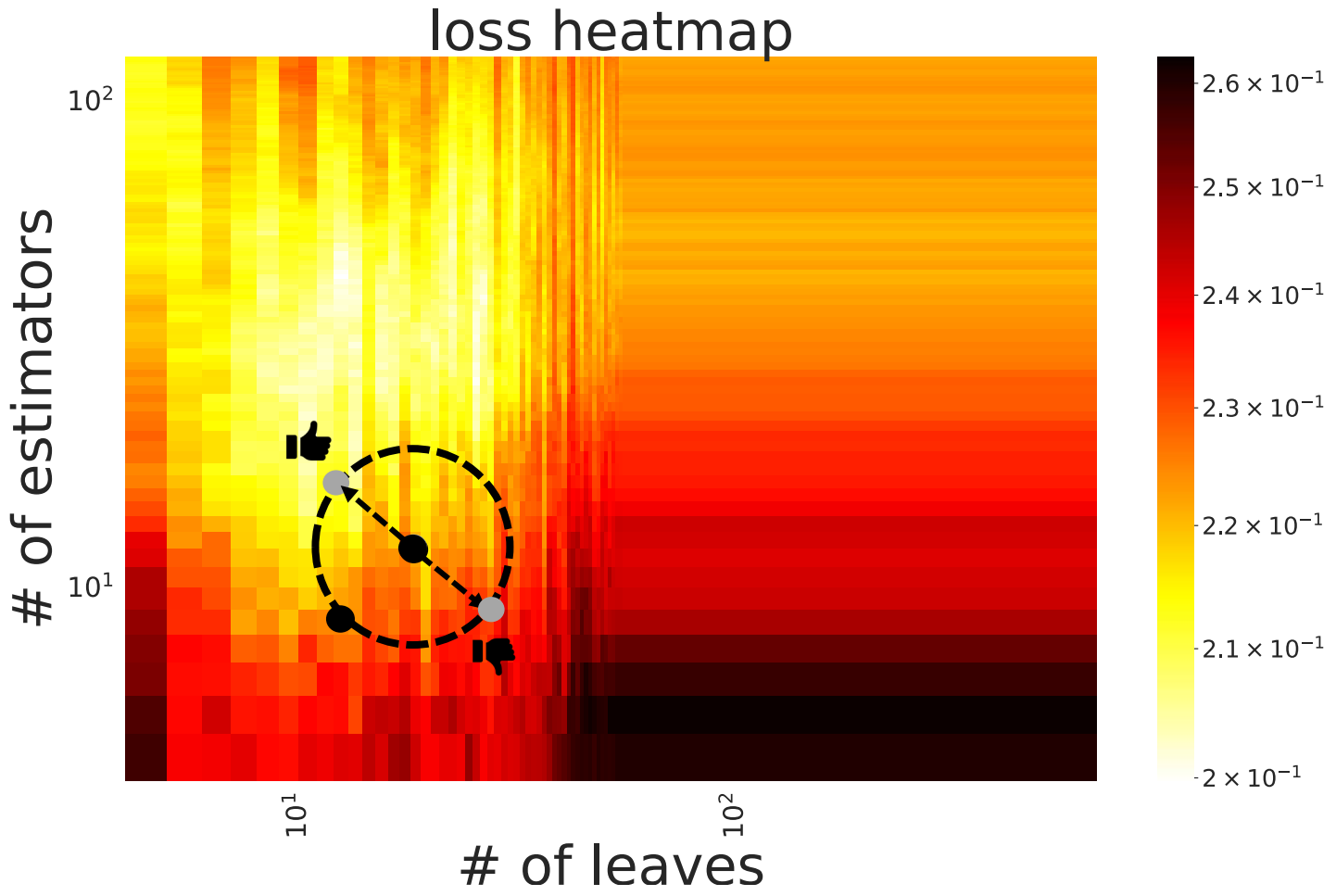
A Cost-Frugal HPO Algorithm (CFO)[*AAAI'21*]



Repeat the following steps after each *move*:

1. **Uniformly** sample a direction from **a local unit sphere;**
2. **Compare;**
3. ***Move*** *(and break)* or try the **opposite direction**;
4. ***Move*** *(and break)* or stay.

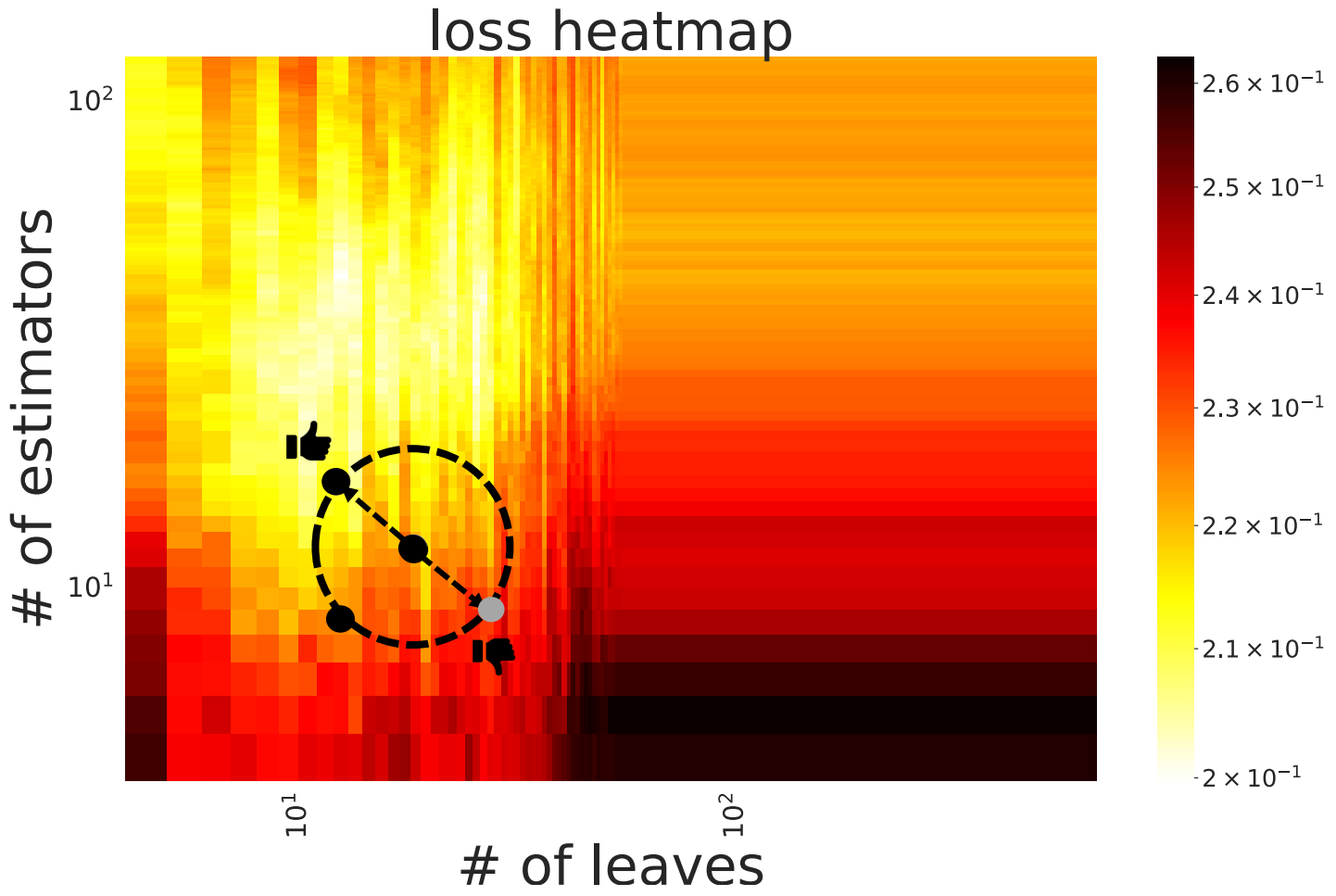
A Cost-Frugal HPO Algorithm (CFO)[*AAAI'21*]



Repeat the following steps after each *move*:

1. **Uniformly** sample a direction from **a local unit sphere;**
2. **Compare;**
3. ***Move*** *(and break)* or try the **opposite direction**;
4. ***Move*** *(and break)* or stay.

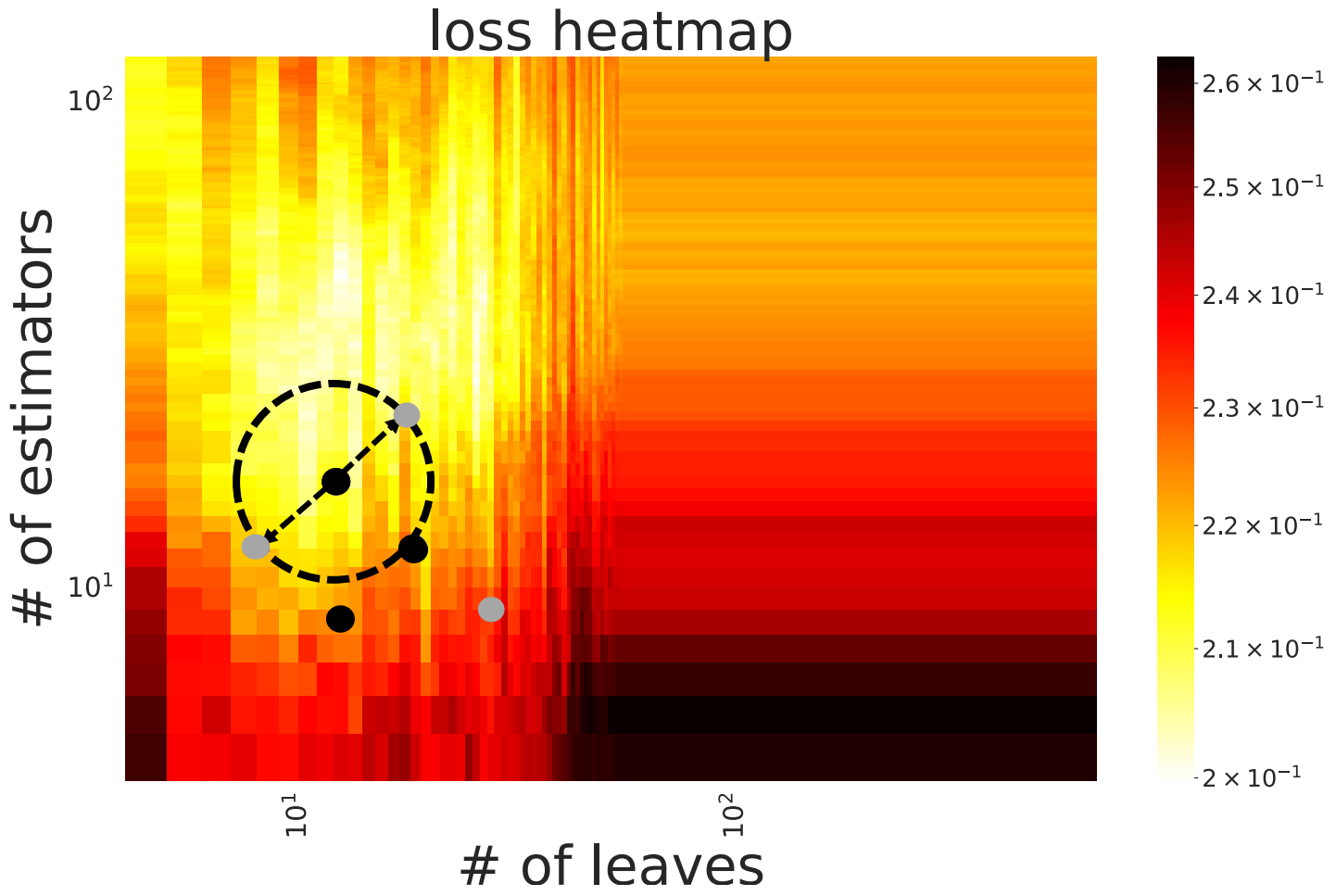
A Cost-Frugal HPO Algorithm (CFO)[*AAAI'21*]



Repeat the following steps after each *move*:

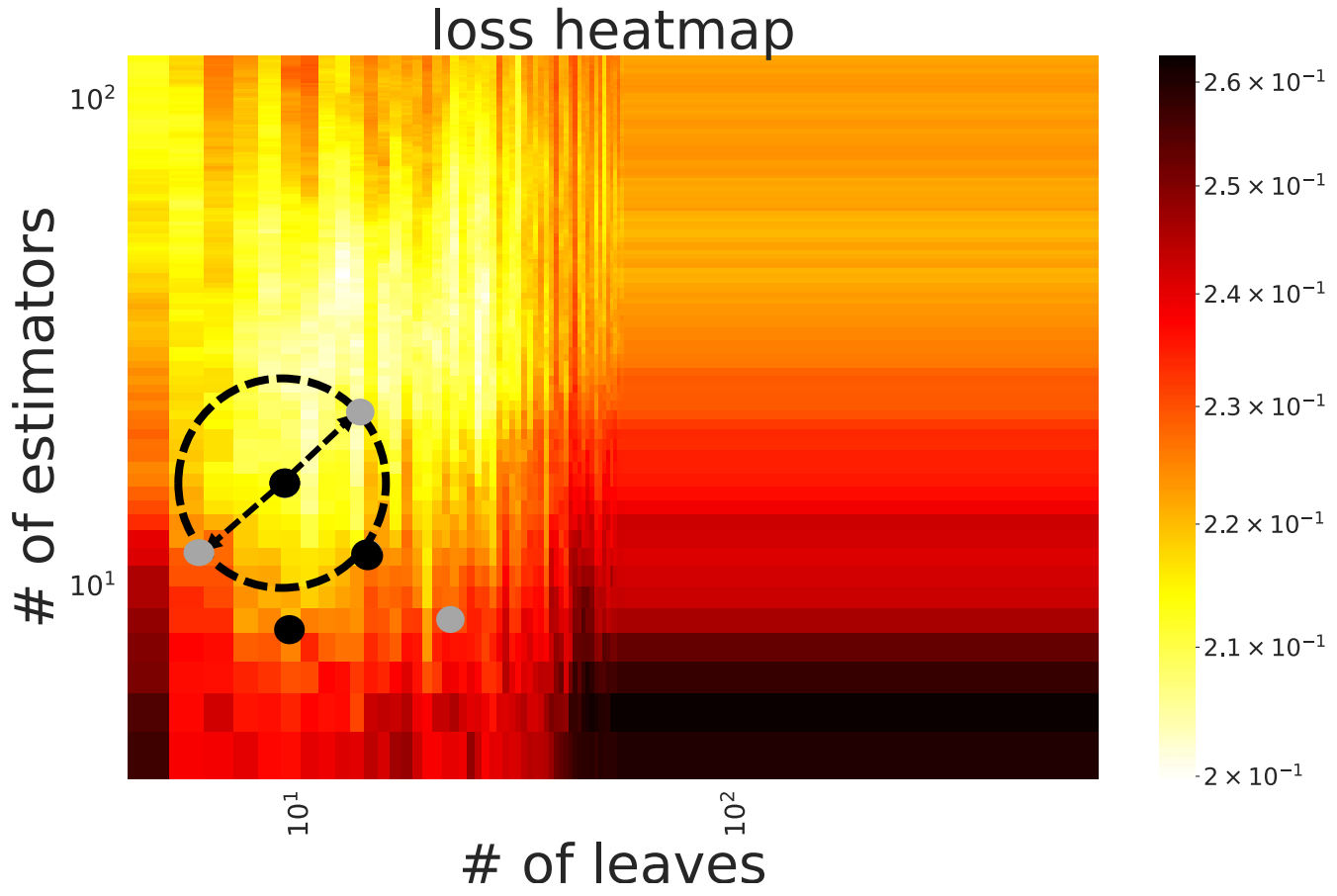
1. **Uniformly** sample a direction from **a local unit sphere;**
2. **Compare;**
3. ***Move*** *(and break)* or try the **opposite direction**;
4. ***Move*** *(and break)* or stay.

A Cost-Frugal HPO Algorithm (CFO)[*AAAI'21*]



Repeat the following steps after each *move*::

1. **Uniformly** sample a direction from **a local unit sphere;**
2. **Compare;**
3. ***Move (and break)*** **or try the opposite direction;**
4. *Move (and break)* or stay.



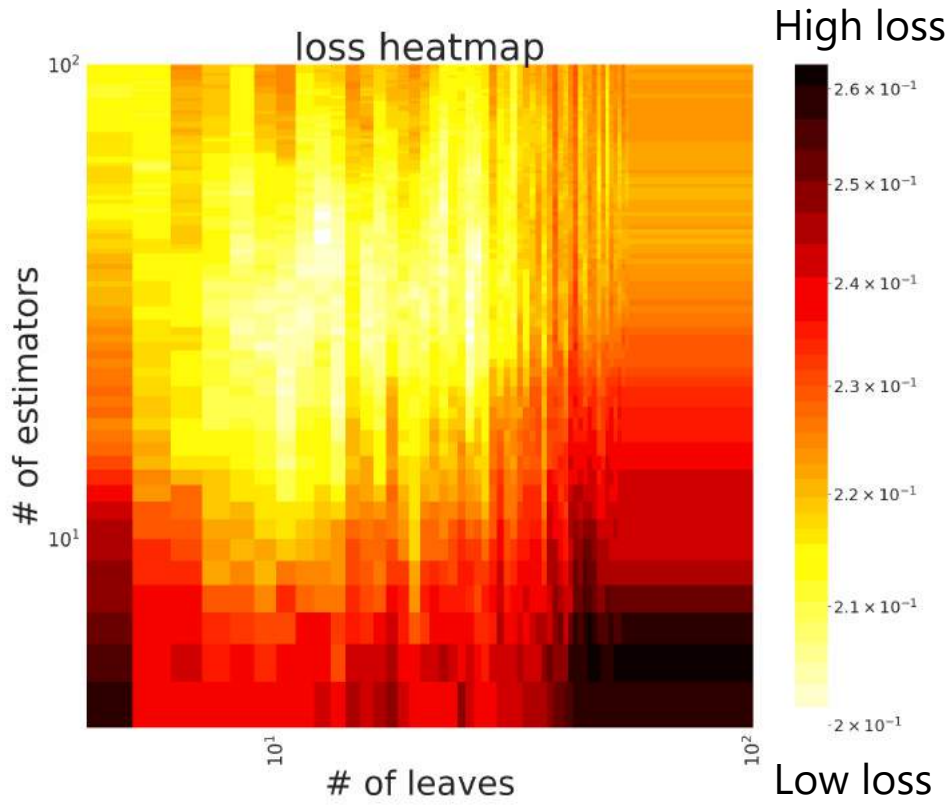
Implications:

At any iteration,

- The **incumbent** has the best loss
- The **evaluated point** in the next iteration is in the neighboring area of the incumbent =>the cost is not far away from the incumbent

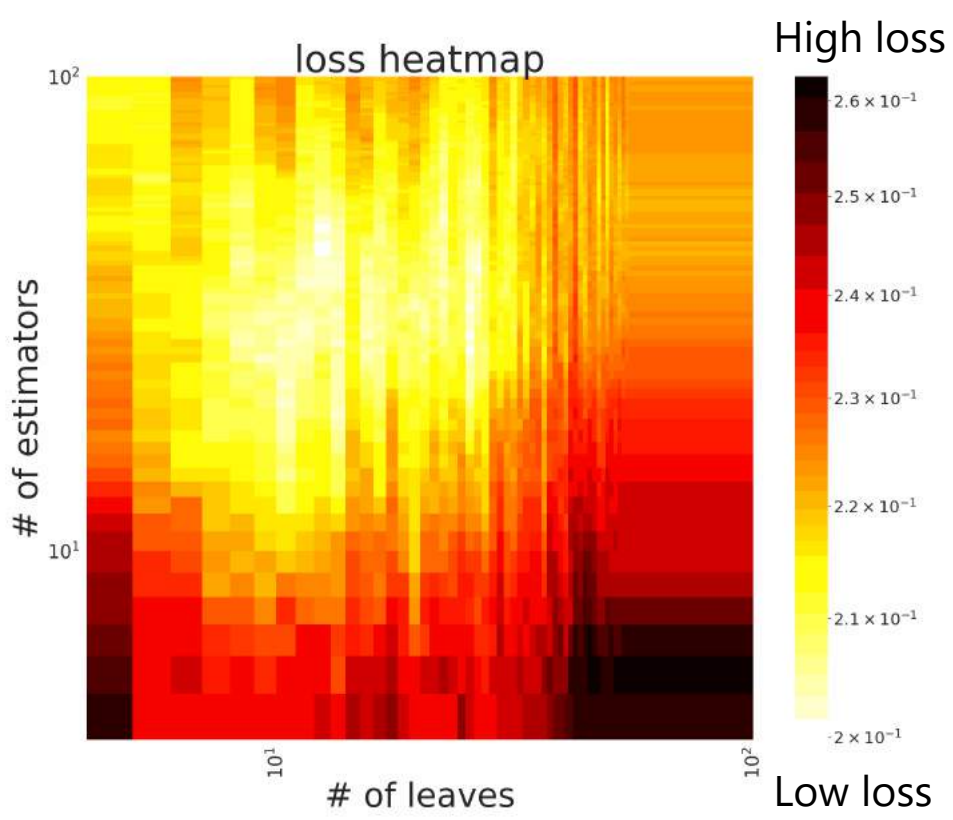
A Cost-Frugal HPO Algorithm (CFO)[*AAAI'21*]

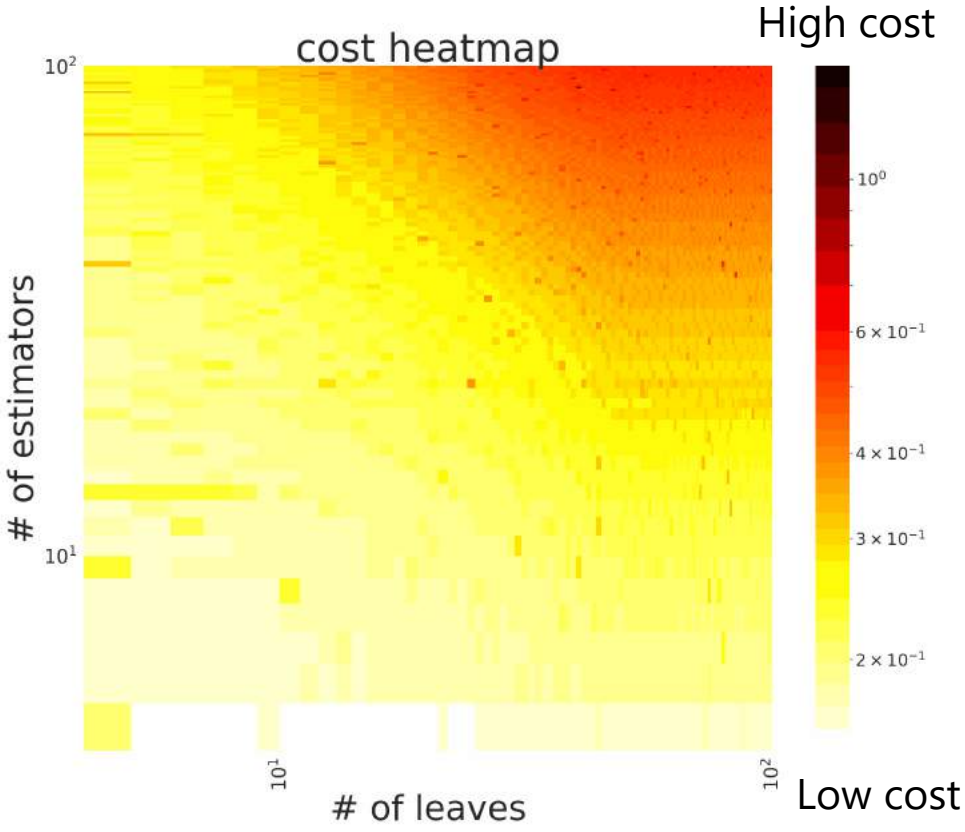
- **Theoretical guarantees on:**
  - Convergence rate



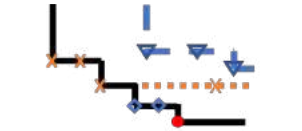
A Cost-Frugal HPO Algorithm (CFO)[*AAAI'21*]

- **Theoretical guarantees on:**
  - Convergence rate
  - The total evaluation cost

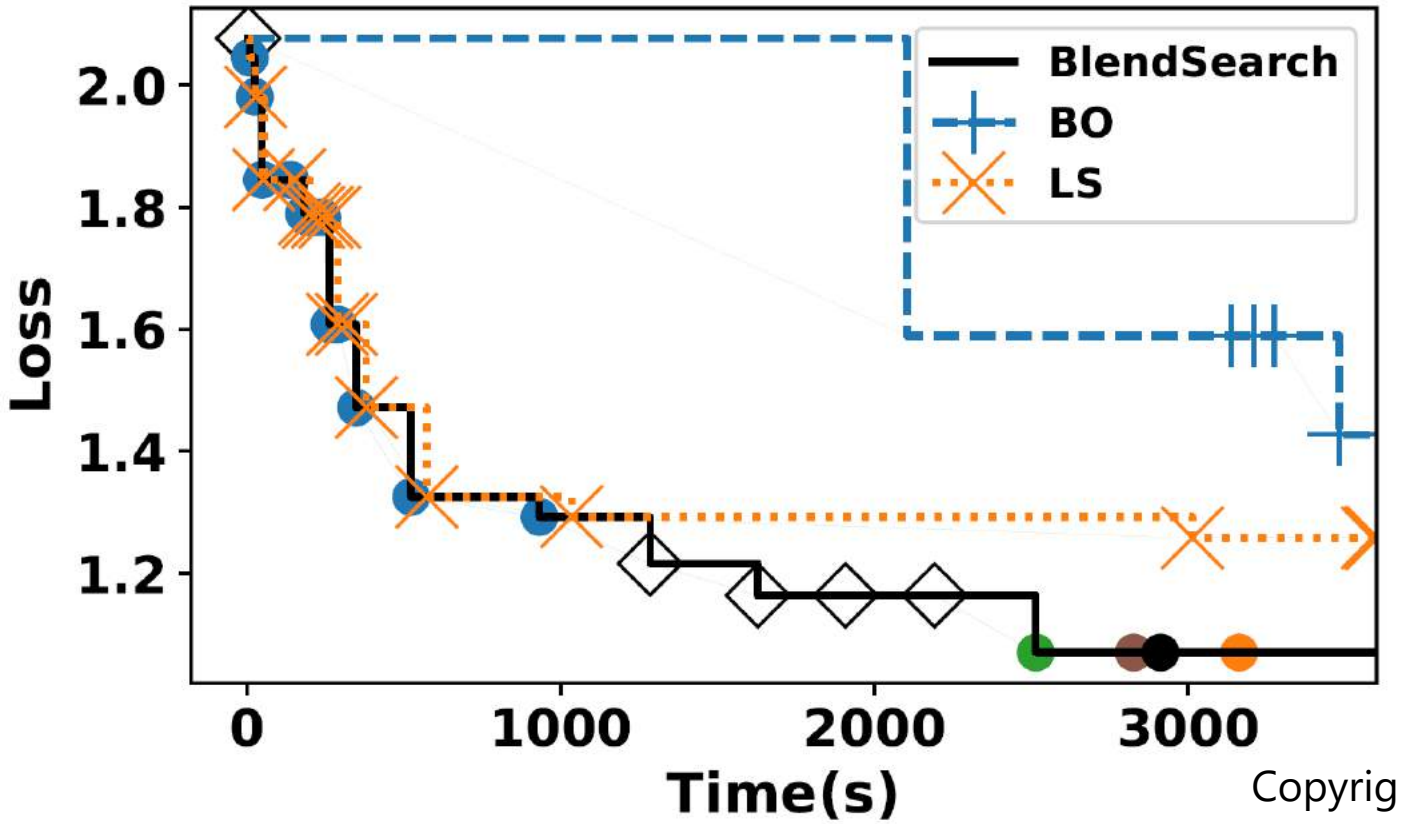


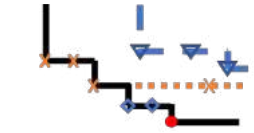


Combine local search and global search



- LS – low cost; may get trapped in local optima
- Global search– able to explore the whole space; high cost

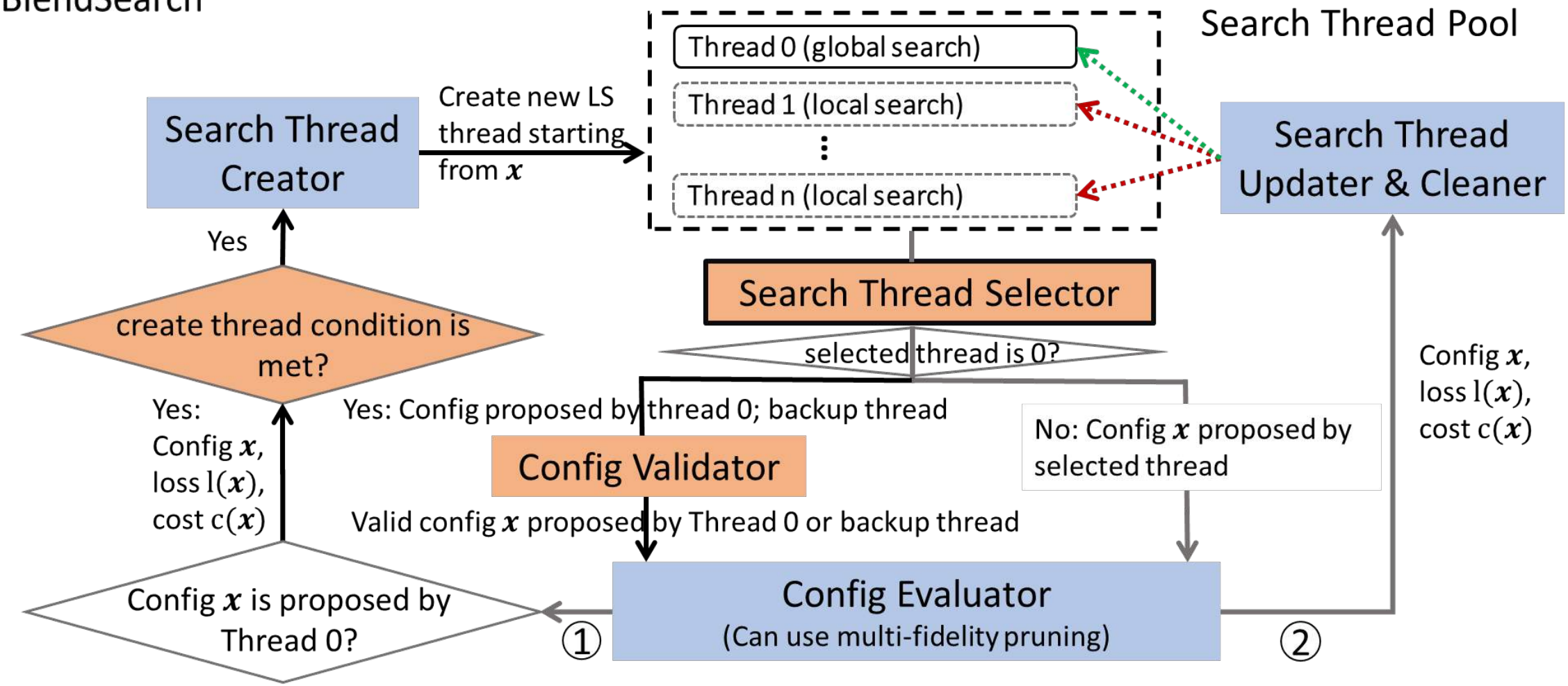




BlendSearch

Framework

- Economical Hyperparameter Optimization With Blended Search Strategy. Chi Wang, Qingyun Wu, Silu Huang, Amin Saied. ICLR 2021.



Benefits of Task-oriented AutoML in FLAML

- Save manual efforts (human resource)
- **Effectiveness** and **efficiency**: use small computation resource to find models with good performance
- Rich **customization choices** in FLAML

  Helpful in scenarios such as:

  Due to business/deployment requirements, you need to use

  - Special **ML learner** (e.g., a domain specific one), or/and
  - **Custom metrics,** or/and
  - **Various constraints** (e.g., in terms of computation resource, model complexity, inference time)

  …

An Example Use Case That Needs Customization

- Application domain: Security
- Overall objective: Find a good model to detect suspicious behaviors.

Comments about customization (in blue).
Comments about performance (in green).

"

- *I am able to use *** classifiers as the custom learner and use search space for random forest, lr and lightgbm.*

...

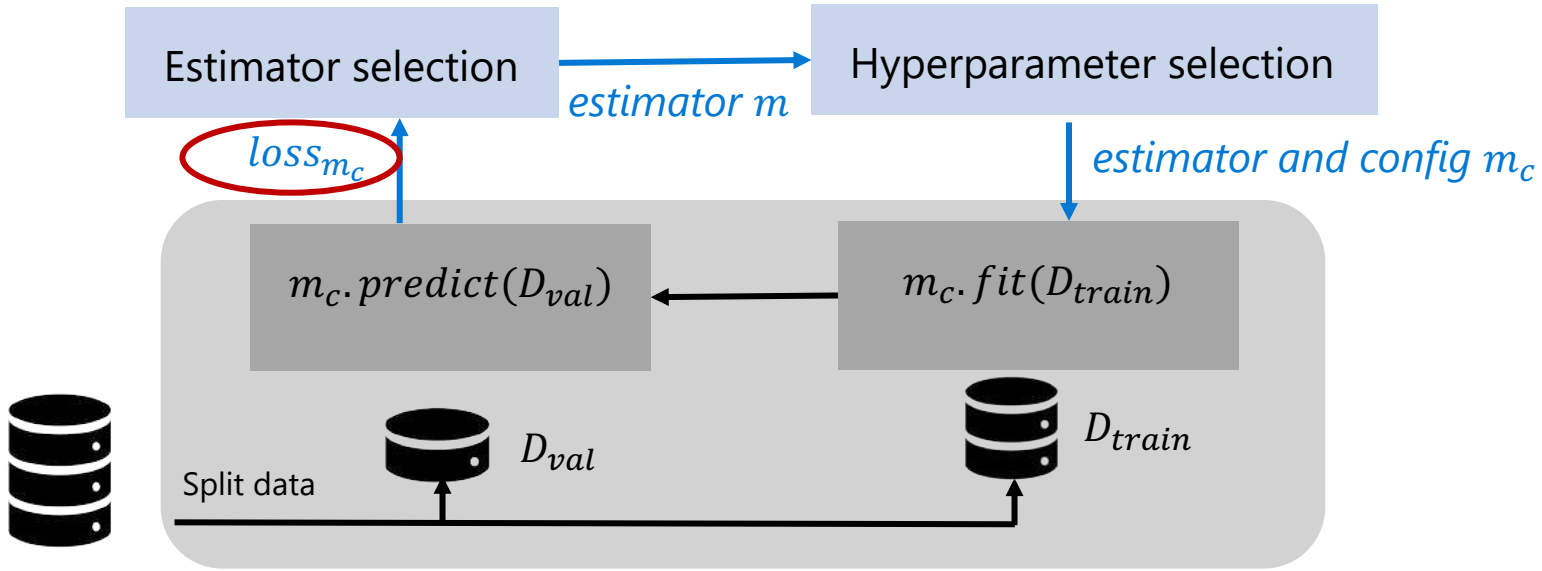
- *It is useful for me to optimize the hyperparameters in a short time. So I appreciate that, and to be able to customize the metrics. My job is to create detectors using models, my last few models were optimized using flaml. It is a corporate level product.*

...

- *Adding 0.5% positive precision increase to "suspicious behavior detector" while adding 3.5% more true positives (positive recall) Adding 0.8% positive precision increase to "suspicious remote behavior detector" while adding 24% more true positives (positive recall). Contributed to ~2000 detections weekly for both detectors above.*

"

Task-oriented AutoML: Optimization Metric



"
I appreciate that I am able to customize the metrics. My job is to create detectors using models, my last few models were optimized using flaml. "

Built-in Optimization Metrics

- *metric*
  - 'accuracy': 1 - accuracy as the corresponding metric to minimize.
  - 'log_loss': default metric for multiclass classification.
  - 'r2': 1 - r2_score as the corresponding metric to minimize. Default metric for regression.
  - 'rmse': root mean squared error.
  - 'mse': mean squared error.
  - 'mae': mean absolute error.
  - 'mape': mean absolute percentage error.
  - 'roc_auc': minimize 1 - roc_auc_score. Default metric for binary classification.
  - 'roc_auc_ovr': minimize 1 - roc_auc_score with `multi_class="ovr"`.
  - 'roc_auc_ovo': minimize 1 - roc_auc_score with `multi_class="ovo"`.
  - 'f1': minimize 1 - f1_score.
  - 'micro_f1': minimize 1 - f1_score with `average="micro"`.
  - 'macro_f1': minimize 1 - f1_score with `average="macro"`.
  - 'ap': minimize 1 - average_precision_score.
  - 'ndcg': minimize 1 - ndcg_score.
  - 'ndcg@k': minimize 1 - ndcg_score@k. k is an integer.

Built-in Optimization Metrics

- *metric*

```
automl = AutoML()


settings = {
    "time_budget": 600,
    "metric": 'accuracy',
    "task": 'classification',
}


'''The main flaml automl API'''
automl.fit(X_train=X_train, y_train=y_train, **settings)
```

User-defined Metric Function

- *metric*

```
automl = AutoML()
settings = {
    "time_budget": 10,  # total running time in seconds
    "metric": custom_metric,  # pass the custom metric funtion here
    "task": 'classification',  # task type
}
automl.fit(X_train=X_train, y_train=y_train, **settings)
```

```python
def custom_metric(X_val, y_val, estimator, labels, X_train, y_train,
                  weight_val=None, weight_train=None, config=None,
                  groups_val=None, groups_train=None):
    from sklearn.metrics import log_loss
    import time
    start = time.time()
    y_pred = estimator.predict_proba(X_val)
    pred_time = (time.time() - start) / len(X_val)
    val_loss = log_loss(y_val, y_pred, labels=labels,
                        sample_weight=weight_val)
    y_pred = estimator.predict_proba(X_train)
    train_loss = log_loss(y_train, y_pred, labels=labels,
                          sample_weight=weight_train)
    alpha = 0.5
    return val_loss * (1 + alpha) - alpha * train_loss, {
        'val_loss': val_loss, "train_loss": train_loss, "pred_time": pred_time
    }
    # two elements are returned:
    # the first element is the metric to minimize as a float number,
    # the second element is a dictionary of the metrics to log
```

Optimization metric

Metrics to log

"*It does allow us to build our models fast; esp. it allows us to* ***control overfitting***.
*We did observe that the models we built using the packages are simpler (in terms of #trees, depth, and #features in the models) and more robust (over time), compared to other models we built in the past using similar tools like ***(another HPO service).*
"

The Power of a User-defined Metric Function

> *"Closing the gap between the loss function we optimize in ML and the product metrics we really want to optimize."*
> --Carlos Guestrin at KDD '19, on *4 Perspectives in Human-Centered Machine Learning*

- User-defined Metric Function in FLAML:

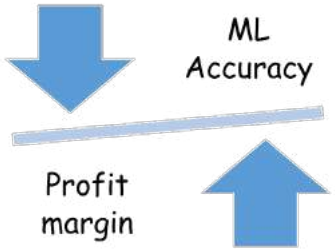
Allow creative metrics toward the *ultimate* objectives

Metrics beyond typical ML predictive performance metrics

For example, business objectives, such as *profit or revenue*

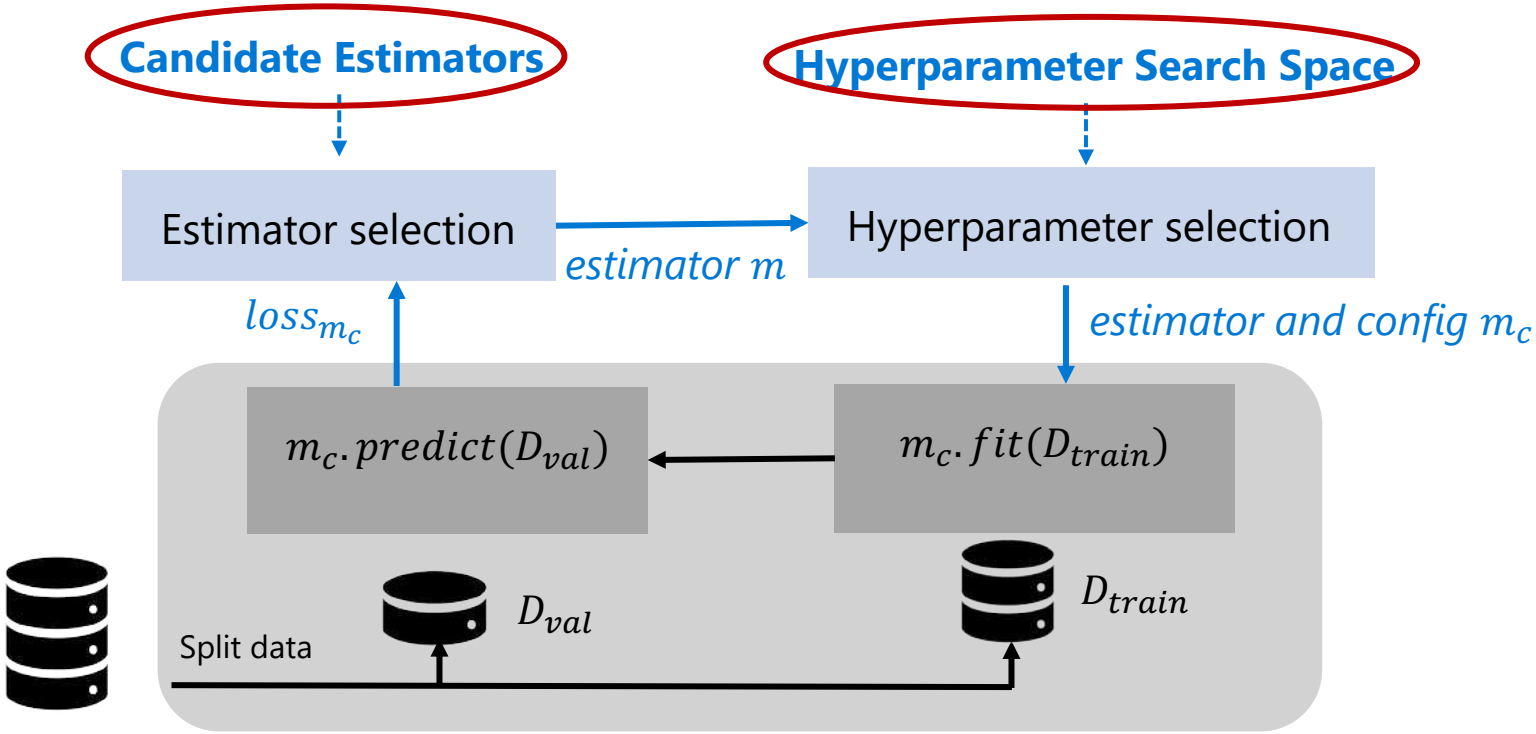
Concrete examples:

1. A heuristic objective to control overfitting: $obj = Loss_{val} * (1 + \alpha) - \alpha * Loss_{train}$
2. Integrating business optimization with a machine learning model



Task-oriented AutoML: Estimators and Search Space

*"I am able to use *** classifiers as the custom learner and use search space for random forest, lr and lightgbm.*



Estimators

estimator_list

- Use built-in estimators with default search space
  - ✓ Classification/regression task:
    "lgbm", "xgboost", "rf", "extra_tree", "catboost", "lrl2"/"lrl1", "kneighbor",
  - ✓ Time series forecasting task: *"prophet", "arima", "sarimax"*
  - ✓ NLP task: *"transformer"*
- Add custom estimators

```
automl.fit(
    X_train=X_train,
    y_train=y_train,
    task="classification",
    time_budget=5,
    estimator_list=["xgboost", "lgbm"],
)
```

Adding a Custom Estimator

1. **Build a custom estimator by inheriting flaml.model.BaseEstimator or a derived class.**

```
class BaseEstimator:
    """The abstract class for all learners.

    Typical examples:
    * XGBoostEstimator: for regression.
    * XGBoostSklearnEstimator: for classification.
    * LGBMEstimator, RandomForestEstimator, LRL1Classifier, LRL2Classifier:
        for both regression and classification.
    """
```

...

```
    def fit(self, X_train, y_train, budget=None, **kwargs):…

    def predict(self, X, **kwargs):…

    def predict_proba(self, X, **kwargs):…
```

Adding a Custom Estimator

1. **Build a custom estimator by inheriting [flaml.model.BaseEstimator](#) or a derived class.**

```python
class MyRegularizedGreedyForest(SKLearnEstimator):
    def __init__(self, task='binary', **config):
        '''Constructor

        Args:
            task: A string of the task type, one of
                'binary', 'multiclass', 'regression'
            config: A dictionary containing the hyperparameter names
                and 'n_jobs' as keys. n_jobs is the number of parallel threads.
        '''

        super().__init__(task, **config)

        '''task=binary or multi for classification task'''
        if task in CLASSIFICATION:
            from rgf.sklearn import RGFClassifier

            self.estimator_class = RGFClassifier
        else:
            from rgf.sklearn import RGFRegressor

            self.estimator_class = RGFRegressor
```

```python
@classmethod
def search_space(cls, data_size, task):
    '''[required method] search space

    Returns:
        A dictionary of the search space.
        Each key is the name of a hyperparameter, and value is
            a dict with its domain (required) and low_cost_init_value,
            init_value, cat_hp_cost (if applicable).
            e.g.,
            {'domain': tune.randint(lower=1, upper=10), 'init_value': 1}.
    '''
    space = {
        'max_leaf': {'domain': tune.lograndint(lower=4, upper=data_size[0]),
                     'init_value': 4,
                     'low_cost_init_value': 4},
        'n_iter': {'domain': tune.lograndint(lower=1, upper=data_size[0]),
                     'init_value': 1,
                     'low_cost_init_value': 1},
        'n_tree_search': {'domain': tune.lograndint(lower=1, upper=32768),
                     'init_value': 1,
                     'low_cost_init_value': 1},
        'learning_rate': {'domain': tune.loguniform(lower=0.01, upper=20.0)},
        'min_samples_leaf': {'domain': tune.lograndint(lower=1, upper=20),
                     'init_value': 20},
    }
    return space
```

Adding a Custom Estimator

1. Build a custom estimator by inheriting flaml.model.BaseEstimator or a derived class.

2. Give the custom estimator a name and add it in AutoML.

```
automl = AutoML()
automl.add_learner(learner_name='RGF', learner_class=MyRegularizedGreedyForest)
```

Adding a Custom Estimator

1. Build a custom estimator by inheriting flaml.model.BaseEstimator or a derived class.

2. Give the custom estimator a name and add it in AutoML.

3. Tune the newly added custom estimator depending on your needs.

```
settings = {
    "time_budget": 10,  # total running time in seconds
    "metric": 'accuracy',
    "estimator_list": ['RGF', 'lgbm', 'rf', 'xgboost'],  # list of ML learners
    "task": 'classification',  # task type
}

automl.fit(X_train=X_train, y_train=y_train, **settings)
```

Search Space

```
space = {
    'max_leaf': {
        'domain': tune.lograndint(lower=4, upper=data_size[0]),
        'init_value': 4,
        'low_cost_init_value': 4},
    'n_iter': {
        'domain': tune.lograndint(lower=1, upper=data_size[0]),
        'init_value': 1,
        'low_cost_init_value': 1},
    'n_tree_search': {
        'domain': tune.lograndint(lower=1, upper=32768),
        'init_value': 1,
        'low_cost_init_value': 1},
    'learning_rate': {
        'domain': tune.loguniform(lower=0.01, upper=20.0)},
    'min_samples_leaf': {
        'domain': tune.lograndint(lower=1, upper=20),
        'init_value': 20},
}
```

Each hyperparameter is associated with a dict with the following fields:

- "**domain**", specifies the possible values of the hyperparameter and their distribution.
- "**init_value**" (optional), which specifies the initial value of the hyperparameter.
- "**low_cost_init_value**" (optional), which specifies the value of the hyperparameter that is associated with low computation cost.

Search Space

```
space = {
    'max_leaf': {
        'domain': tune.lograndint(lower=4, upper=data_size[0]),
        'init_value': 4,
        'low_cost_init_value': 4},
    'n_iter': {
        'domain': tune.lograndint(lower=1, upper=data_size[0]),
        'init_value': 1,
        'low_cost_init_value': 1},
    'n_tree_search': {
        'domain': tune.lograndint(lower=1, upper=32768),
        'init_value': 1,
        'low_cost_init_value': 1},
    'learning_rate': {
        'domain': tune.loguniform(lower=0.01, upper=20.0)},
    'min_samples_leaf': {
        'domain': tune.lograndint(lower=1, upper=20),
        'init_value': 20},
}
```

Each hyperparameter is associated with a dict with the following fields:

- "**domain**", specifies the possible values of the hyperparameter and their distribution.
- "**init_value**" (optional), which specifies the initial value of the hyperparameter.
- "**low_cost_init_value**" (optional), which specifies the value of the hyperparameter that is associated with low computation cost.

Cost-related hyperparameter

Cost-related Hyperparameter in the Search Space

```python
class LGBMEstimator(BaseEstimator):
    """The class for tuning LGBM, using sklearn API."""

    ITER_HP = "n_estimators"
    HAS_CALLBACK = True
    DEFAULT_ITER = 100

    @classmethod
    def search_space(cls, data_size, **params):
        upper = max(5, min(32768, int(data_size[0])))  # upper must be larger than lower
        return {
            "n_estimators": {
                "domain": tune.lograndint(lower=4, upper=upper),
                "init_value": 4,
                "low_cost_init_value": 4,
            },
            "num_leaves": {
                "domain": tune.lograndint(lower=4, upper=upper),
                "init_value": 4,
                "low_cost_init_value": 4,
            },
            "min_child_samples": {
                "domain": tune.lograndint(lower=2, upper=2**7 + 1),
                "init_value": 20,
            },
            "learning_rate": {
                "domain": tune.loguniform(lower=1 / 1024, upper=1.0),
                "init_value": 0.1,
            },
```

Resources to Be Used in This Tutorial

First half of the tutorial:

- Overview of AutoML and FLAML
- Task-oriented AutoML with FLAML
  - Notebook: A classification task with AutoML
  - Notebook: A regression task with AuotML using LightGBM as the learner
- ML.Net demo
- Tune user defined functions with FLAML
  - Notebook: Basic tuning procedures and advanced tuning options
  - Notebook: Tune pytorch
- Q & A

Second half of the tutorial:

- Zero-shot AutoML
  - Notebook: Zeroshot AutoML
- Time series forecasting
  - Notebook: AutoML for Time Series Forecast tasks
- Natural language processing
  - Notebook: AutoML for NLP tasks
- Online AutoML
  - Notebook: Online AutoML with Vowpal Wabbit
- Fair AutoML
- Challenges and open problems

- https://github.com/microsoft/FLAML/tree/tutorial/tutorial

Customize the Search Spaces (of Existing Estimators)

Option 1. Create a new estimator with a revised search space.

Option 2. A shortcut to override the search space of existing estimators via *custom_hp*.

```
settings = {
    "time_budget": 60,
    "metric": 'r2',
    "custom_hp": custom_hp,
    "task": 'regression',  # task type
}
automl.fit(X_train=X_train, y_train=y_train,
           **settings)
```

Customize the Search Spaces (of Existing Estimators)

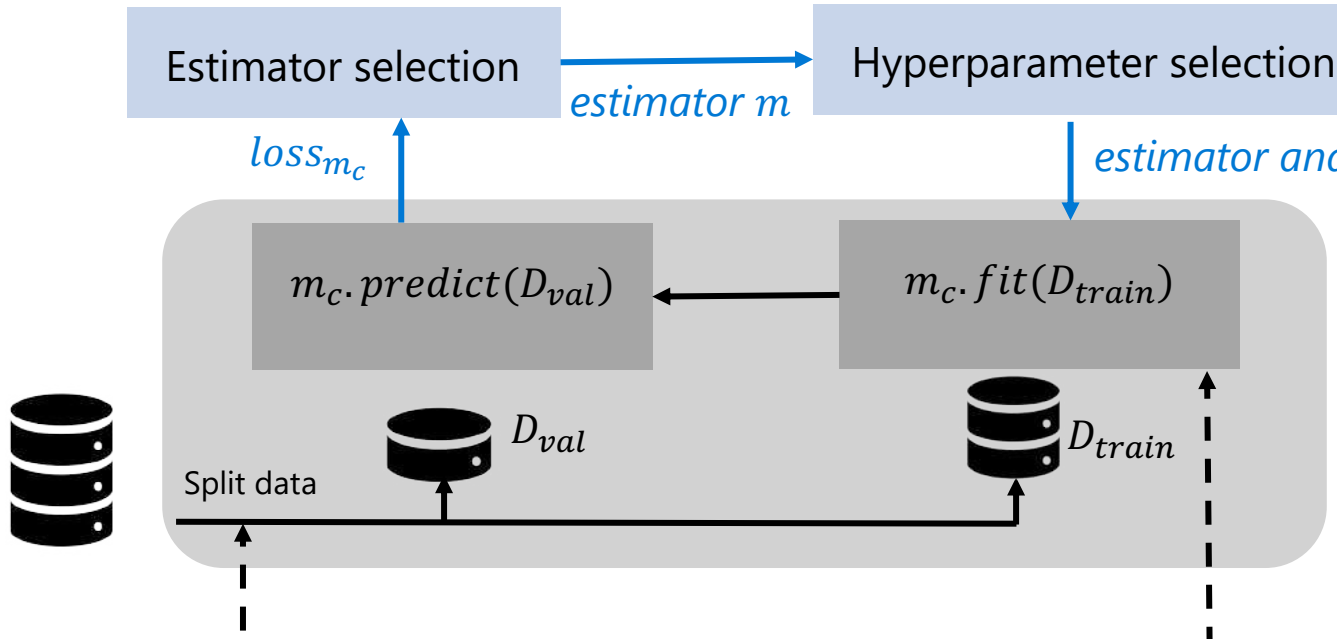
```
custom_hp = {
    "xgboost": {
        "n_estimators": {
            "domain": tune.lograndint(lower=4, upper=100),
            "low_cost_init_value": 4,
        },
    },
    "rf": {
        "max_leaves": {
            "domain": None,  # disable search
        },
    },
    "lgbm": {
        "subsample": {
            "domain": tune.uniform(lower=0.1, upper=1.0),
            "init_value": 1.0,
        },
        "subsample_freq": {
            "domain": 1,  # subsample_freq must > 0 to enable subsample
        },
    },
}
```

Using a different search range for "n_estimators"

Disable search by setting "domain" to None

Setting a constant value

Task-oriented AutoML: ML Procedure and Ensemble




```
automl.fit(
    X_train, y_train, task="classification",
    "ensemble": {
        "final_estimator": LogisticRegression(),
        "passthrough": False,
    },
)
```

eval_method: A string of resampling strategy, one of ['auto', 'cv', 'holdout'].
split_ratio, n_splits,
split_type: ["auto", 'stratified', 'uniform', 'time', 'group']
X_val, y_val

***fit_kwargs*:** Provide additional key word arguments to pass to fit() function of the candidate learners, such as sample_weight.
fit_kwargs_by_estimator: The user specified keywords arguments, grouped by estimator name.

Task-oriented AutoML: Advanced Functionalities

- Required inputs
  - X_train, y_train, task, *time_budget/max_iter*
- Logging
  - *log_file_name*
  - *log_type*
- Constraints
  - *train_time_limit*
  - *pred_time_limit*
  - *metric_constraints*
- Warm start
  - *starting_points*
- Parallel tuning
  - *n_concurrent_trials*

Task-oriented AutoML: Advanced Functionalities

- Required inputs
  - X_train, y_train, task, *time_budget/max_iter*
- Logging
  - *log_file_name:* A string of the log file name
  - *log_type:* "better" or "all"
- Constraints
  - *train_time_limit*
  - *pred_time_limit*
  - *metric_constraints*
- Warm start
  - *starting_points*
- Parallel tuning
  - *n_concurrent_trials*

"The team that built out the manual process were only able to review two or three different models. I was able to look at 50.
I was able to report on more insights, more understanding of the variable inputs than they were in the same amount of time, more understanding around why the model performed the way that it did." (P10)

[Xin et al. 2021]

Task-oriented AutoML: Logging

{"**record_id**": 12, "**iter_per_learner**": 2, "**logged_metric": {"pred_time"**: 3.085368373117264e-07}, **"trial_time**": 0.05657434463500976 6, "**wall_clock_time**": 1.4277665615081787, "**validation_loss**": 0.407244741798933 33, "**config**": {"n_estimators": 4, "max_leaves": 4, "learning_rate": 0.0385913619213208 2, "subsample": 1.0, "colsample_bylevel": 0.814847411062700 4, "colsample_bytree": 0.97772348004424 23, "reg_alpha": 0.0009765625, "reg_lambda": 5.525802807180917, "min_child_weight": 0.011999696534212 02, "FLAML_sample_size": 10000}, "**learner": "xgboost**", "sample_size": 10000}

Task-oriented AutoML: Logging with MLFlow

```
experiment = mlflow.set_experiment("flaml")
with mlflow.start_run() as run:
    automl.fit(X_train=X_train, y_train=y_train, **settings)
    # log the model
    mlflow.sklearn.log_model(automl, "automl")
```

Retrieve logs

```
mlflow.search_runs(experiment_ids=[experiment.experiment_id], filter_string="params.learner = 'xgboost'")
```

Load the model

```
automl = mlflow.sklearn.load_model(f"{run.info.artifact_uri}/automl")
print(automl.predict_proba(X_test))
print(automl.predict(X_test))
```

Task-oriented AutoML: More Constraints

- Required inputs
  - X_train, y_train, task, *time_budget/max_iter*
- Logging
  - *log_file_name:* A string of the log file name
  - *log_type:* "better" or "all"
- Constraints
  - *train_time_limit:* Training time constraint in seconds
  - *pred_time_limit:* Predict time constraint in seconds
  - *metric_constraints:* A list of constraints on certain metrics
- Warm start
  - *starting_points*
- Parallel tuning
  - *n_concurrent_trials*

Constraints of 4 Different Types

1. Constraints on the AutoML process: *time_budget, max_iter*

```
automl.fit(..., time_budget=60)
```

```
automl.fit(..., max_iter=10)
```

2. Constraints on the constructor arguments of the estimators.

dmlc XGBoost

- **monotone_constraints** (*Optional[Union[Dict[str, int], str]]*) – Constraint of variable monotonicity. See tutorial for more information.

```
from flaml.model import XGBoostSklearnEstimator
class MonotonicXGBoostEstimator(XGBoostSklearnEstimator):
    @classmethod
    def search_space(cls, data_size, task):
        space = super().search_space(data_size)
        space.update({"monotone_constraints": {"domain": "(1, -1)"}})
        return space
```

Constraints of 4 Different Types

3. Constraints on the models tried in AutoML: *train_time_limit, pred_time_limit*

```
automl.fit(..., train_time_limit=60)
```

Or/and

```
automl.fit(..., pred_time_limit=1)
```

Constraints of 4 Different Types

4. Constraints on the metrics of the ML model tried in AutoML: metric_constraints

```
automl = AutoML()
metric_constraints = [("train_loss", "<=", 0.1), ("val_loss", "<=", 0.1)]
settings = {
    "time_budget": 10,  # total running time in seconds
    "metric": custom_metric,  # pass the custom metric funtion here
    "task": 'classification',  # task type
    "train_time_limit": 1,
    "pred_time_limit": 0.1,
    "metric_constraints": metric_constraints,
}
automl.fit(X_train, y_train, **settings)
```

Constraints of 4 Different Types

4. Constraints on the metrics of the ML model tried in AutoML: metric_constraints

```
def custom_metric(X_val, y_val, estimator, labels, X_train, y_train,
                  weight_val=None, weight_train=None, config=None,
                  groups_val=None, groups_train=None):
    from sklearn.metrics import log_loss
    import time
    start = time.time()
    y_pred = estimator.predict_proba(X_val)
    pred_time = (time.time() - start) / len(X_val)
    val_loss = log_loss(y_val, y_pred, labels=labels,
                        sample_weight=weight_val)
    y_pred = estimator.predict_proba(X_train)
    train_loss = log_loss(y_train, y_pred, labels=labels,
                          sample_weight=weight_train)
    alpha = 0.5
    return val_loss * (1 + alpha) - alpha * train_loss, {
        "val_loss": val_loss, "train_loss": train_loss, "pred_time": pred_time
    }
    # two elements are returned:
    # the first element is the metric to minimize as a float number,
    # the second element is a dictionary of the metrics to log
```

Task-oriented AutoML: Warm Start

- Required inputs
  - X_train, y_train, task, *time_budget/max_iter*
- Logging
  - *log_file_name:* A string of the log file name
  - *log_type:* "better" or "all"
- Constraints
  - *train_time_limit*
  - *pred_time_limit*
  - *metric_constraints*
- Warm start
  - *starting_points:* Starting hyperparameter config for the estimators
- Parallel tuning
  - *n_concurrent_trials*

Warm Start

- ***starting_points:***

A dictionary of config start with or a str to specify the starting hyperparameter config for the estimators | default="data".

If str:
- if "data", use **data-dependent defaults**;
- if "data:path", use data-dependent defaults which are stored at path;
- if "static", use data-independent defaults.

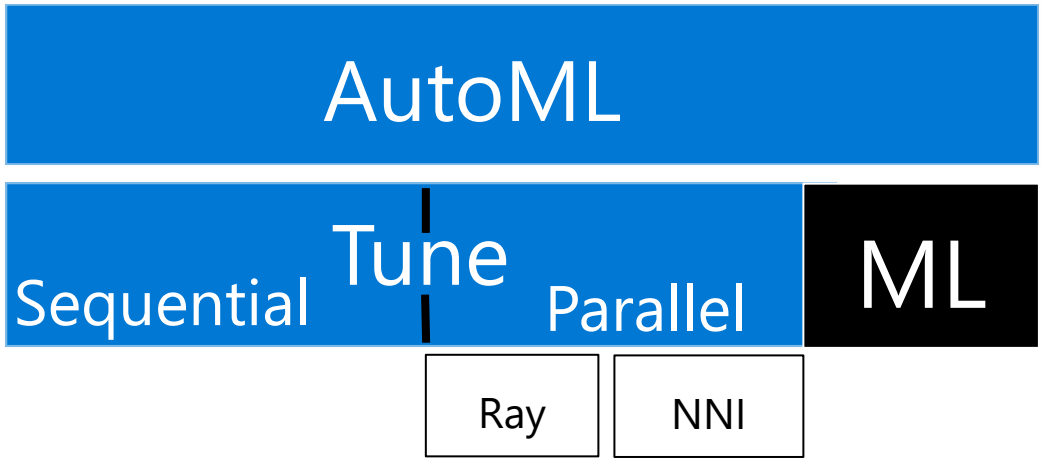
Will be covered later in zero-shot AutoML

```
from flaml import AutoML
automl1 = AutoML()
automl1.fit(X_train, y_train, time_budget=30)
automl2 = AutoML()
automl2.fit(X_train, y_train, time_budget=60, starting_points=automl1.best_config_per_estimator)
```

Task-oriented AutoML: Parallel Tuning

- Required inputs
  - X_train, y_train, task, *time_budget/max_iter*
- Logging
  - *log_file_name:* A string of the log file name
  - *log_type:* "better" or "all"
- Constraints
  - *train_time_limit*
  - *pred_time_limit*
  - *metric_constraints*
- Warm start
  - *starting_points*
- Parallel tuning
  - *n_concurrent_trials:* The number of concurrent trials

Parallel Tuning




```
%pip install flaml[ray,blendsearch]
```

```
import ray
from flaml import AutoML
ray.init(num_cpus=4)
automl = AutoML()
automl.fit(X_train, y_train, time_budget=30, n_jobs=2, n_concurrent_trials=2)
```

Easy Parallel Tuning Is a Desirable Feature

*So I work in Research and that entire application I wrote myself over the last 9 months. My job (and the teams) currently is to research a next state application stack for our existing products. In particular the models out of this product *** have been implemented in FLAML (this is our own GLM and GBM) which is exactly why we liked FLAML because it allowed us to easily extend the estimators, something *** (an alternative HPO service) does not allow.* ***Anyway the reason we love FLAML and Ray is to get large scale parallel tuning, something the product team are struggling with because it's desktop and C# based, so to have FLAML out of the box distribute across an auto scaled cluster removes at least 18 months work for us.***

-- A S&P 500 company

Parallel vs Sequential Tuning

Heads-up: Parallel tuning is not necessarily more desirable than sequential tuning. 

Things to Consider:

- Different overhead and trial time.

E.g., when parallel tuning is used (ray backend is used), there will be a certain computation overhead that is larger than sequential tuning.

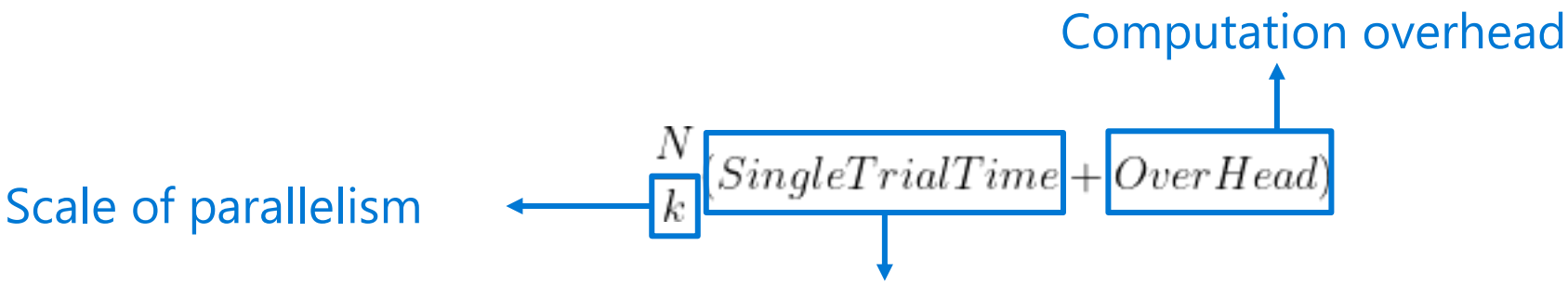
- Availability of parallel resources
- Different randomness

Find more in this doc:

https://microsoft.github.io/FLAML/docs/Use-Cases/Task-Oriented-AutoML#parallel-tuning

Parallel vs Sequential Tuning

A rough estimation of the wall-clock time needed to finish N trials:




```
 "trial_time": 0.21239686012268066, "wall_clock_time": 3.023219347000122,
 "trial_time": 0.2319955825805664, "wall_clock_time": 3.484747886657715, "
 "trial_time": 0.26106786727905273, "wall_clock_time": 3.5908350944519043,
 "trial_time": 0.25072693824768066, "wall_clock_time": 3.8085615634918213,
 "trial_time": 0.23816537857055664, "wall_clock_time": 4.002406597137451,
"trial_time": 0.23777341842651367, "wall_clock_time": 4.1051344871521, "va
 "trial_time": 0.2535979747772217, "wall_clock_time": 4.408732652664185, "
"trial_time": 0.28461432456970215, "wall_clock_time": 4.545221567153931, "
```


k(scale of parallelism) = 8,
SingleTrialTime ≈ 0.3
OverHead (parallel tuning) ≈ 2.6

➡ Sequential tuning is faster

use_ray in Sequential Tuning

- ***use_ray***: boolean or dict.
  - If boolean, default=False | Whether to use ray to run the training **in separate processes**. This can be used to **prevent OOM for large datasets**, but will incur more overhead in time.
  - If dict: the dict contains the keywords arguments to be passed to ray.tune.run

Suggested scenarios to use ray backend:

1. **Parallel tuning**
2. **Sequential tuning** with potential Out Of Memory (OOM) failure

AutoML Use Cases: 1. Credit Scoring and Fraud Detection in Financial Industry

Advantages of FLAML:

- Overall objective and constraints: Finding an "optimal" (gradient boosting or deep learning) model that
  - performs the **best** in **out-of-time (OOT)** period;
  - **control model complexity** (e.g., # of variables in the model, # of trees, tree depth);
  - maintain **model explainability**;
  - preferably meet the following criteria: (1**) do not over fit** the training data; (2) **perform consistently between training/holdout/OOT**; (3) **Simple and intuitive, and can pass regulatory exams**.

- Fast HPO
- Controlled by our HPO algorithm
- Allow custom metric

"

- *It does allow us to build our models fast; esp. it allows us to control overfitting.*
- *We did observe that the models we built using the packages are simpler (in terms of #trees, depth, and #features in the models) and more robust (over time), compared to other models we built in the past using similar tools like ***(another HPO service).*

"

AutoML Use Cases: 2. Investment Management

- Overall objectives and constraints: Find the **best model (customized estimator)** based on **business requirements** for deployment.
- Advantages of FLAML:
  - Saved dev time
  - High compatibility

"

Dev time saved 30 - 40 percent on average; for gigantic datasets the saving is even more; regarding the performance, AUC wise the lift is about 0.1 - 0.2 point

For us, a big part of R&D is testing new algorithms released in recent publications with internal data; thanks to the high compatibility of FLAML, we can conveniently incorporate these new algorithms into the existing pipeline and make apples-to-apples comparisons.

"

-- A large private equity firm

AutoML Use Cases: 3. Custom Retention and Growth Analysis

- **Overall objectives and constraints:** Find a good multiclass workload classification model (mainly xgboost) based on the usage and behavior pattern.
- **Advantages of FLAML:**
  - Fast (30 hours grid search -> minutes)
  - Can find accurate models

> "
> the model runs faster (in minutes) and we are able to try out various sampling techniques and additional derived attributes. Thus, we are using FLAML model in the production classification models. It really helped to improve our team productivity and reduced development iterations.
> "

AutoML Use Cases: 4. Security

- **Overall objective and constraints**: Find a good model to detect suspicious behaviors.
- **Advantages of FLAML:**
  - Support customized learner, custom metrics
  - Fast
  - Can find accurate models

"

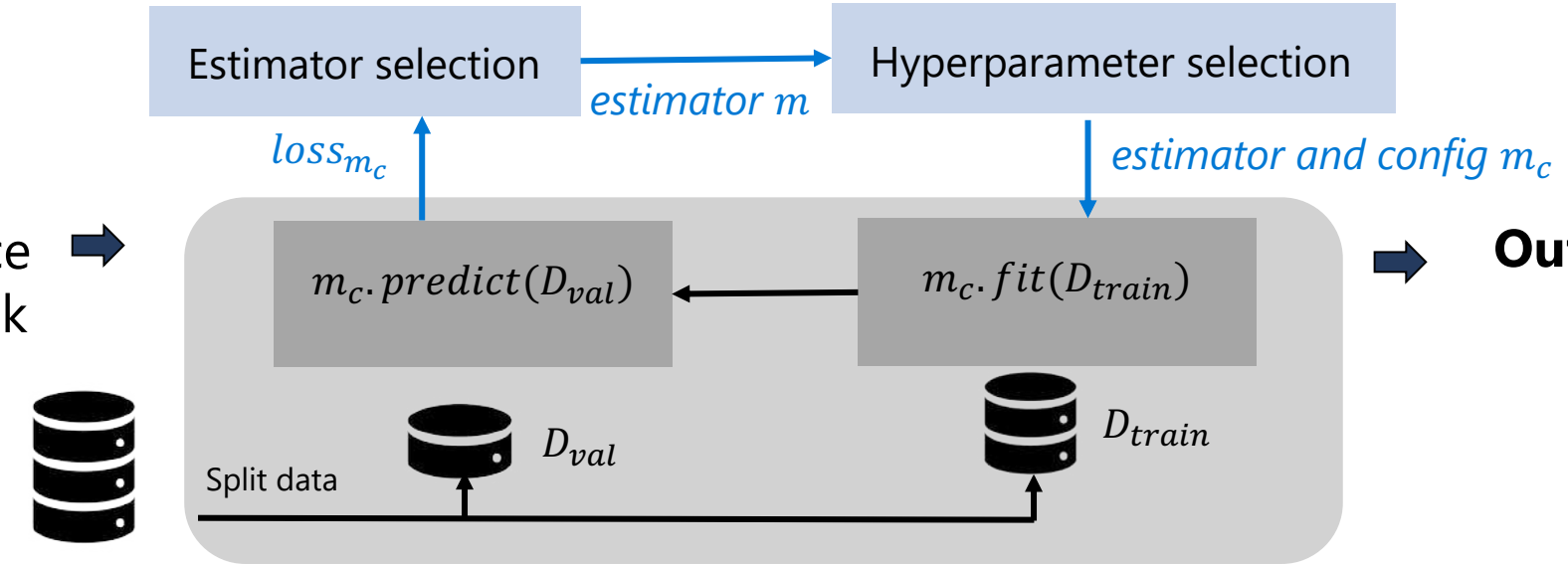
It is useful for me to optimize the hyperparameters in a short time. So I appreciate that, and to be able to customize the metrics. My job is to create detectors using models, my last few models were optimized using flaml. It is a corporate level product.

...

Adding 0.5% positive precision increase to "suspicious behavior detector" while adding 3.5% more true positives (positive recall). Adding 0.8% positive precision increase to "suspicious remote behavior detector" while adding 24% more true positives (positive recall). Contributed to ~2000 detections weekly for both detectors above.

"

Task-oriented AutoML: Q & A



Inputs: resource budget, ML task

Outputs: ML model (and more)

```
from flaml import AutoML
automl = AutoML()
automl.fit(X_train, y_train, task="classification", time_budget=60)
```

Agenda

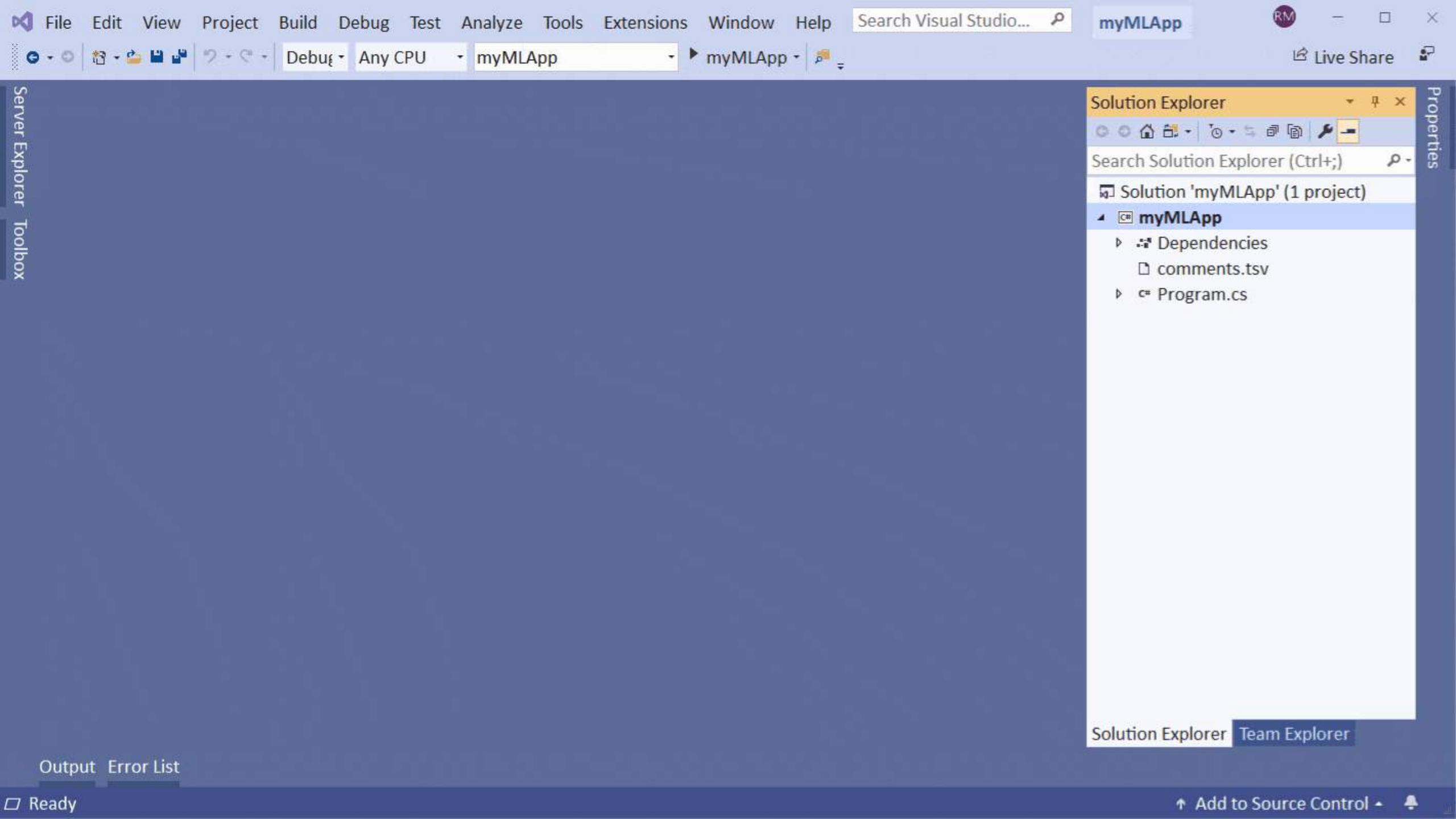
First half (9:30 AM – 11:10 AM)

- Overview of AutoML and FLAML (9:30 AM)
- Task-oriented AutoML with FLAML (9:45 AM)
- **ML.NET demo (10:30 AM)**
- Tune user defined functions with FLAML (10:40 AM)

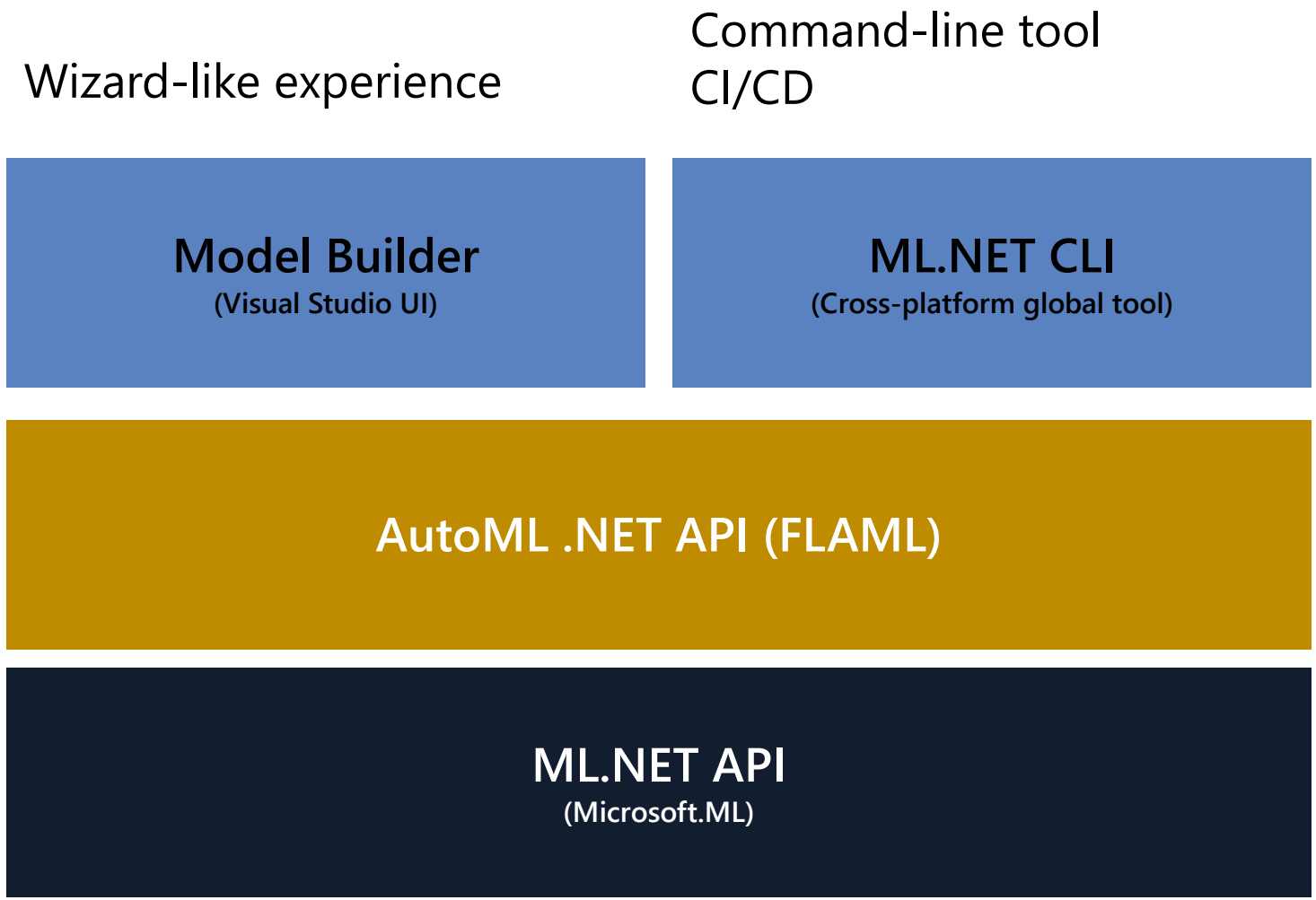
Break, Q & A

Second half (11:20 AM – 12:30 AM)

- Zero-shot AutoML (11:20 AM)
- Time series forecasting
- Natural language processing (11:45 AM)
- Online AutoML (12:00 PM)
- Fair AutoML
- Challenges and open problems


File Edit View Project Build Debug Test Analyze Tools Extensions Window Help
Search Visual Studio...
myMLApp
Live Share
Debug
Any CPU
myMLApp
myMLApp
Server Explorer
Toolbox
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'myMLApp' (1 project)
myMLApp
Dependencies
comments.tsv
Program.cs
Properties
Solution Explorer
Team Explorer
Output
Error List
Ready
Add to Source Control

ML.NET Platform



Demo: FLAML AutoML in .NET

Agenda

First half (9:30 AM – 11:10 AM)

- Overview of AutoML and FLAML (9:30 AM)
- Task-oriented AutoML with FLAML (9:45 AM)
- ML.NET demo (10:30 AM)
- **Tune user defined functions with FLAML (10:40 AM)**

Break, Q & A

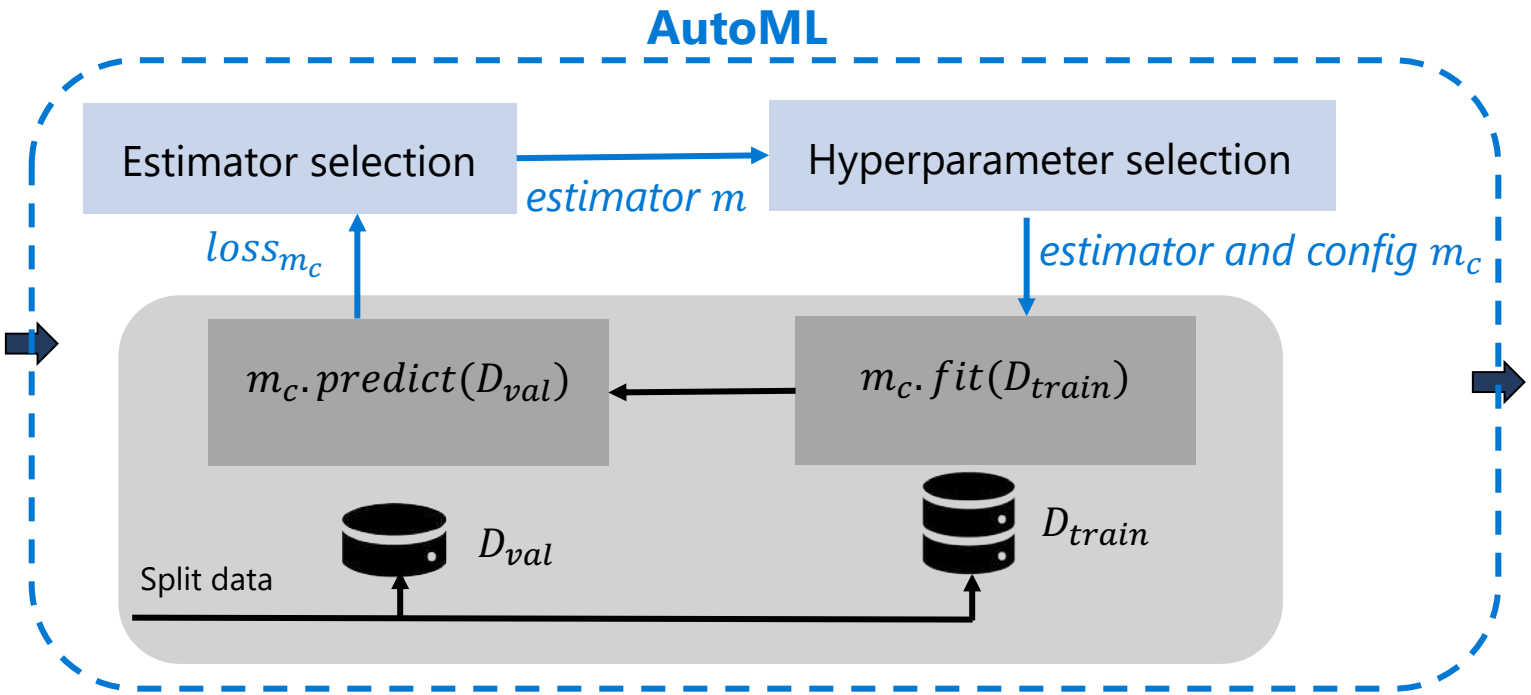
Second half (11:20 AM – 12:30 AM)

- Zero-shot AutoML (11:20 AM)
- Time series forecasting
- Natural language processing (11:45 AM)
- Online AutoML (12:00 PM)
- Fair AutoML
- Challenges and open problems

Tune User Defined Function

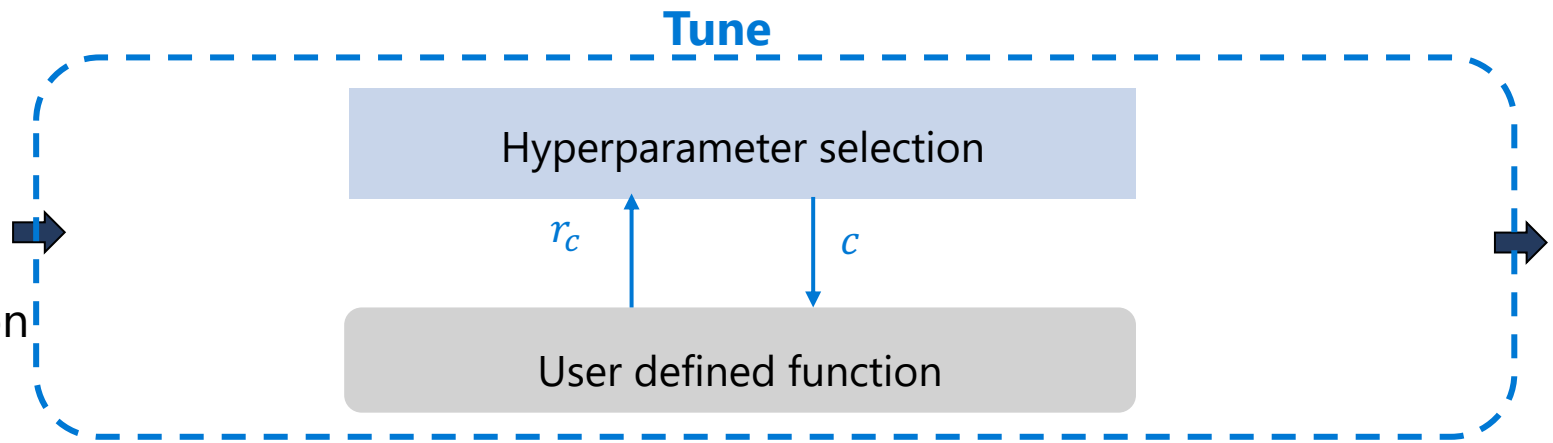
AutoML vs Tune

Inputs:
1. Resource budget
2. ML task



Outputs:
ML model
(and more)

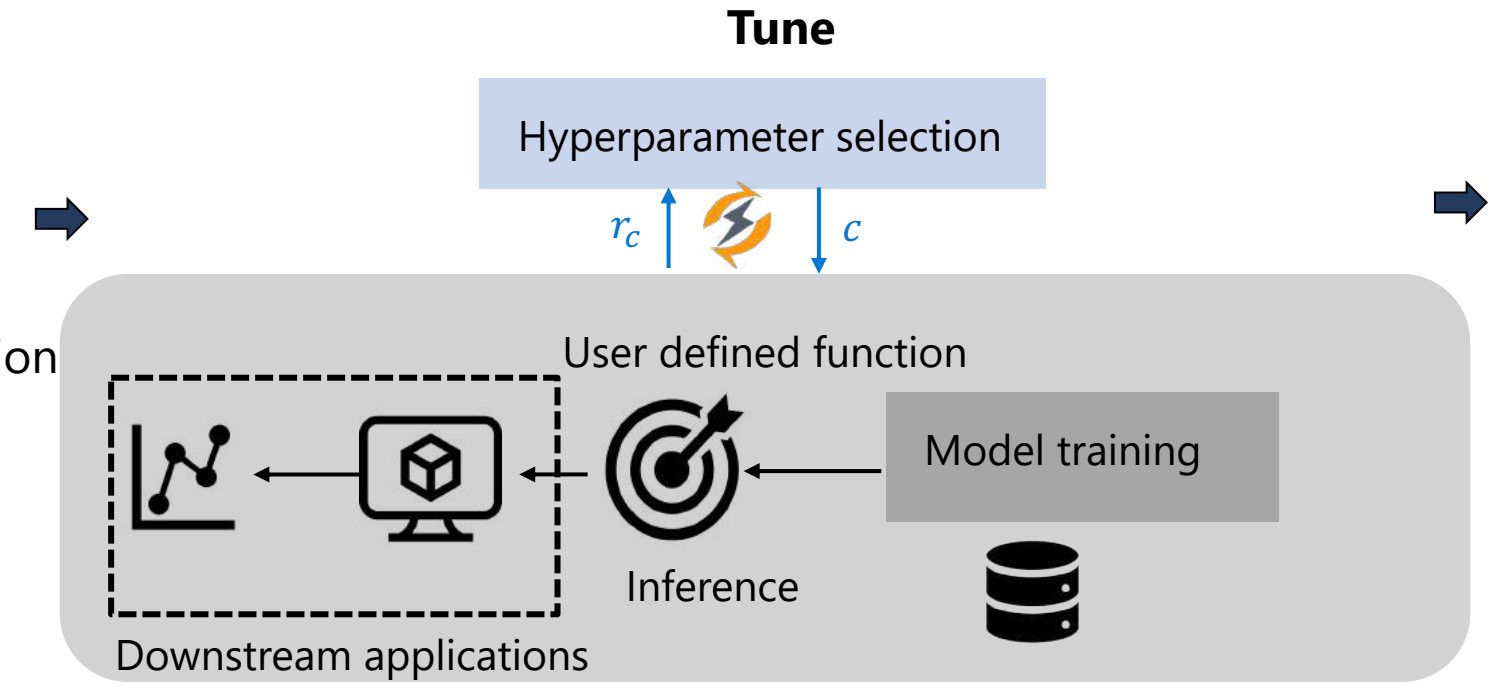
Inputs:
1. Resource budget
2. Search space
3. User defined function



Outputs:
Best config

Tune User Defined Function

Inputs:
1. Resource budget
2. Search space
3. User defined function



Outputs: Best config

Tune User Defined Function

It can be used to tune generic hyperparameters for:

- MLOps workflows, pipelines
- Mathematical/statistical models
- Algorithms
- Computing experiments
- Software configurations

…

Examples:

AzureML pipeline, MLflow pipeline

Casual models

RL policies

Simulations in environmental science

Database configurations

Resources to Be Used

First half of the tutorial:

- Overview of AutoML and FLAML
- Task-oriented AutoML with FLAML
  - Notebook: A classification task with AutoML
  - Notebook: A regression task with AuotML using LightGBM as the learner
- ML.Net demo
- Tune user defined functions with FLAML
  - Notebook: Basic tuning procedures and advanced tuning options
  - Notebook: Tune pytorch
- Q & A

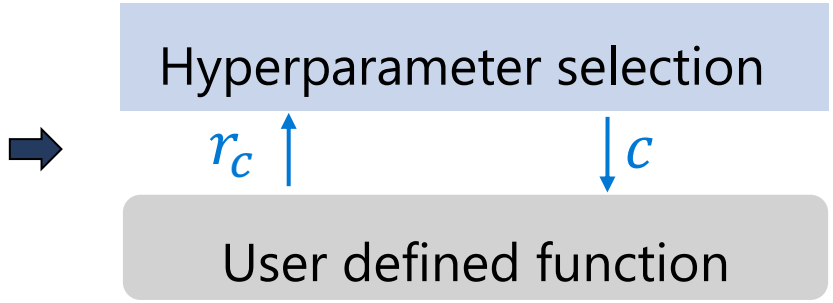
Second half of the tutorial:

- Zero-shot AutoML
  - Notebook: Zeroshot AutoML
- Time series forecasting
  - Notebook: AutoML for Time Series Forecast tasks
- Natural language processing
  - Notebook: AutoML for NLP tasks
- Online AutoML
  - Notebook: Online AutoML with Vowpal Wabbit
- Fair AutoML
- Challenges and open problems

- https://github.com/microsoft/FLAML/tree/tutorial/tutorial

Tune User Defined Function: A Basic Tuning Procedure

Inputs:
1. Resource budget
2. Search space
3. User defined function



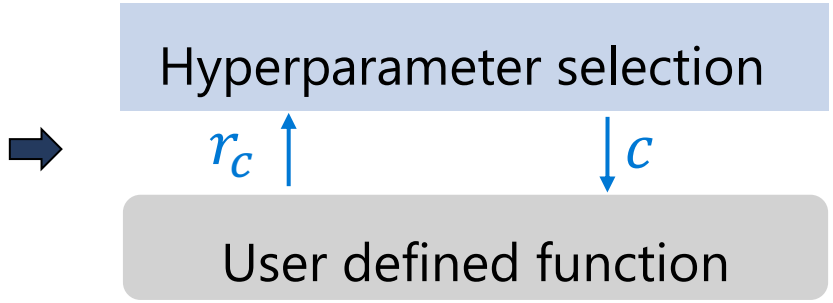
Outputs: Best config

```
''''Performs tuning'''
# require: pip install flaml[blendsearch]
analysis = tune.run(
    evaluate_config,  # the function to evaluate a config
    config=config_search_space,  # the search space defined
    metric="score",
    mode="min",  # the optimization mode, "min" or "max"
    num_samples=-1,  # the maximal number of configs to try, -1 means infinite
    time_budget_s=10,  # the time budget in seconds
)
```

Tune User Defined Function: A Basic Tuning Procedure

Inputs:

1. Resource budget
2. Search space
3. User defined function



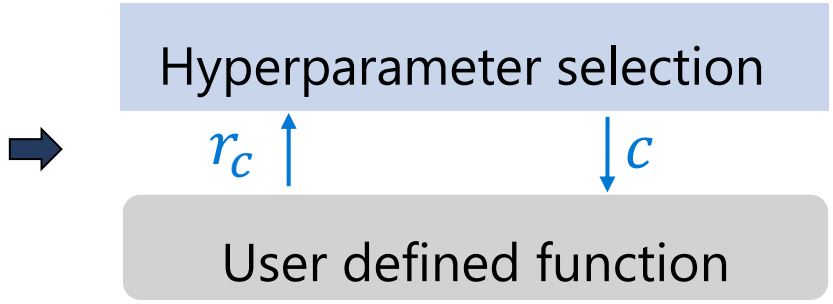
Outputs: Best config

```
config_search_space = {
    "x": tune.lograndint(lower=1, upper=100000),
    "y": tune.randint(lower=1, upper=100000)
}
```

Tune User Defined Function: A Basic Tuning Procedure

Inputs:

1. Resource budget
2. Search space
3. User defined function



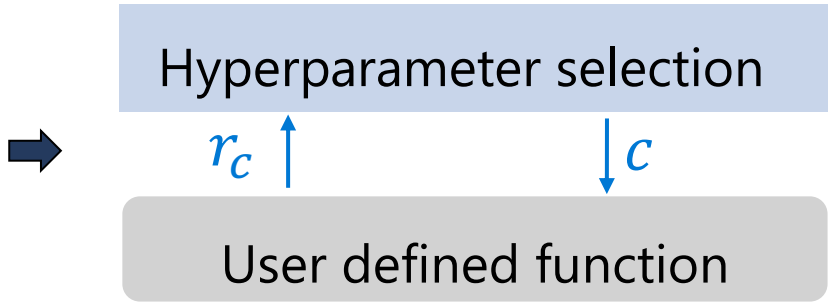
Outputs: Best config

- **Evaluation function**
- Optimization metric
- Optimization mode

```
'''Write a evaluation function'''
import time
def evaluate_config(config: dict):
    """evaluate a hyperparameter configuration"""
    score = (config["x"] - 85000) ** 2 - config["x"] / config["y"]
    # usually the evaluation takes an non-neglible cost
    # and the cost could be related to certain hyperparameters
    # here we simulate this cost by calling the time.sleep() function
    # here we assume the cost is proportional to x
    faked_evaluation_cost = config["x"] / 100000
    time.sleep(faked_evaluation_cost)
    # we can return a single float as a score on the input config:
    # return score
    # or, we can return a dictionary that maps metric name to metric value:
    return {"score": score, "evaluation_cost": faked_evaluation_cost,
            "constraint_metric": config["x"] * config["y"]}
```

Tune User Defined Function: A Basic Tuning Procedure

Inputs:
1. Resource budget
2. Search space
3. User defined function



Outputs: Best config

```
''''Performs tuning'''
# require: pip install flaml[blendsearch]
analysis = tune.run(
    evaluate_config,  # the function to evaluate a config
    config=config_search_space,  # the search space defined
    metric="score",
    mode="min",  # the optimization mode, "min" or "max"
    num_samples=-1,  # the maximal number of configs to try, -1 means infinite
    time_budget_s=10,  # the time budget in seconds
)
```

Tune User Defined Function: A Basic Tuning Procedure

```
'''Investigate results'''
print(analysis.best_result)
```

```
{'score': 138344643.26761267, 'evaluation_cost': 0.73238, 'constraint_metric': 7323726762,
'training_iteration': 0, 'config': {'x': 73238, 'y': 99999}, 'config/x': 73238, 'config/y':
99999, 'experiment_tag': 'exp', 'time_total_s': 0.7350201606750488}
```

Resources to Be Used

First half of the tutorial:

- Overview of AutoML and FLAML
- Task-oriented AutoML with FLAML
  - Notebook: A classification task with AutoML
  - Notebook: A regression task with AuotML using LightGBM as the learner
- ML.Net demo
- Tune user defined functions with FLAML
  - Notebook: Basic tuning procedures and advanced tuning options
  - Notebook: Tune pytorch
- Q & A

Second half of the tutorial:

- Zero-shot AutoML
  - Notebook: Zeroshot AutoML
- Time series forecasting
  - Notebook: AutoML for Time Series Forecast tasks
- Natural language processing
  - Notebook: AutoML for NLP tasks
- Online AutoML
  - Notebook: Online AutoML with Vowpal Wabbit
- Fair AutoML
- Challenges and open problems

- https://github.com/microsoft/FLAML/tree/tutorial/tutorial

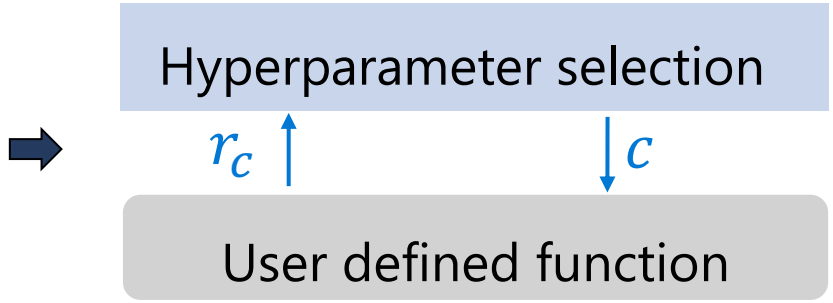
Benefits of Tune

- Save manual efforts (human resource)
- **Effectiveness** and **efficiency**: use small computation resource to find hyperparameter configurations with good performance
- **Flexibility:** support even more diverse use cases than AutoML

More about search space

Inputs:

1. Resource budget
2. Search space
3. User defined function



Outputs: Best config

```
config_search_space = {
    "x": tune.lograndint(lower=1, upper=100000),
    "y": tune.randint(lower=1, upper=100000)
}
```

Cost-related Hyperparameters in Search Space

- *low_cost_partial_config* (optional): A dictionary from a subset of controlled dimensions to the initial low-cost values.
- *cat_hp_cost* (optional): A dictionary from a subset of categorical dimensions to the relative cost of each choice.

```
analysis = tune.run(
    partial(eval_lgbm, X_train, X_test, y_train, y_test),
    config={
        "n_estimators": tune.lograndint(lower=4, upper=32768),
        "max_leaves": tune.lograndint(lower=4, upper=32768),
        "learning_rate": tune.loguniform(lower=1 / 1024, upper=1.0),
        "boosting_type": tune.choice(["gbdt", "goss", "dart"]),
    },
    low_cost_partial_config={"n_estimators": 4, "max_leaves": 4},
    cat_hp_cost={
        "boosting_type": [2, 3, 6],
    },
    metric="loss",
    mode="min",
    num_samples=10,
    )
```

- Search space
- Controlled dimensions where we know the low-cost values
- The relative cost across different categorical choices

Hierarchical Search Space

A hierarchical search space for xgboost

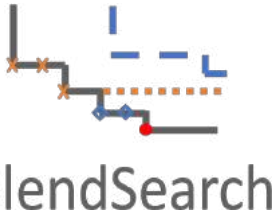
```python
gbtree_hp_space = {
    "booster": "gbtree",
    "n_estimators": tune.lograndint(lower=4, upper=64),
    "max_leaves": tune.lograndint(lower=4, upper=64),
    "learning_rate": tune.loguniform(lower=1 / 1024, upper=1.0),
}
gblinear_hp_space = {
    "booster": "gblinear",
    "lambda": tune.uniform(0, 1),
    "alpha": tune.loguniform(0.0001, 1),
}

full_space = {
    "xgb_config": tune.choice([gbtree_hp_space, gblinear_hp_space]),
}
```

```python
def xgb_obj(X_train, X_test, y_train, y_test, config):
    config = config['xgb_config']
    params = config2params(config)
    dtrain = xgb.DMatrix(X_train, label=y_train)
    booster_type = config.get("booster")

    if booster_type == "gblinear":
        model = xgb.train(params, dtrain,)
    else:
        _n_estimators = params.pop("n_estimators")
        model = xgb.train(params, dtrain, _n_estimators)

    # get validation loss
    from sklearn.metrics import r2_score
    dtest = xgb.DMatrix(X_test)
    y_test_predict = model.predict(dtest)
    test_loss = 1.0 - r2_score(y_test, y_test_predict)
    return {"loss": test_loss}
```

```python
from flaml.data import load_openml_dataset

X_train, X_test, y_train, y_test = load_openml_dataset(
    dataset_id=537, data_dir="./"
)
analysis = tune.run(
    partial(xgb_obj, X_train, X_test, y_train, y_test),
    config=full_space,
    metric="loss",
    mode="min",
    time_budget_s=30,
    num_samples=5,
)
print("analysis", analysis.best_result)
```

HPO algorithm

Local search [AAAI'21]



Local search + Global search [ICLR'21]

CFO is suggested when

- Simple search space
- Known good (low-cost) starting points
- Non-parallel tuning or low parallelization

```
pip install flaml[blendsearch]
```

```
from flaml import CFO
tune.run(...
    search_alg=CFO(low_cost_partial_config=low_cost_partial_config),
)
```

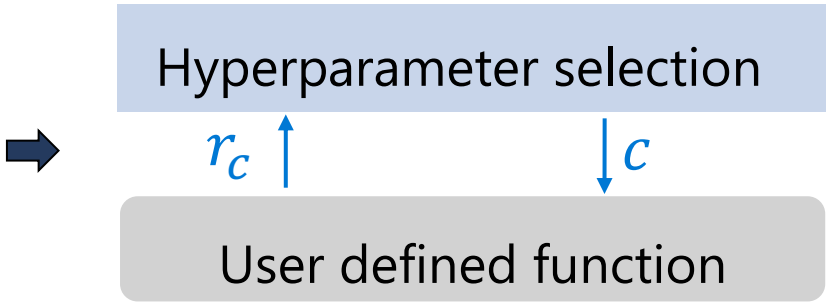
```
# require: pip install flaml[blendsearch]
from flaml import BlendSearch
tune.run(...
    search_alg=BlendSearch(low_cost_partial_config=low_cost_partial_config),
)
```

Tune User Defined Function: More Constraints ?

Inputs:

1. Resource budget
2. Search space
3. User defined function

- More constraints?



Outputs: Best config

More Constraints on the Tuning

- *config_constraints:* constraints on the configurations (a list of 3-tuple)

```
def my_model_size(config):
    return config["n_estimators"] * config["max_leaves"]
```

```
analysis = tune.run(
    partial(xgb_simple_obj, X_train, X_test, y_train, y_test),
    config=xgb_space,
    metric="loss",
    mode="min",
    config_constraints = [(my_model_size, "<=", 40)],
    metric_constraints = [("training_cost", "<=", 1)],
    num_samples=20,
)
```

f: config -> float inequality constraint threshold

More Constraints on the Tuning

- *metric_constraints:* constraints on the metrics (a list of 3-tuple)

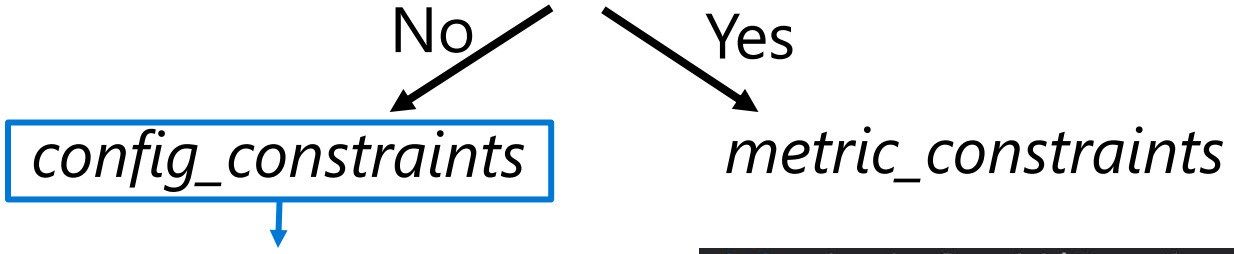
```
analysis = tune.run(
    partial(xgb_simple_obj, X_train, X_test, y_train, y_test),
    config=xgb_space,
    metric="loss",
    mode="min",
    config_constraints = [(my_model_size, "<=", 40)],
    metric_constraints = [('training_cost", "<=", 1)],
    num_samples=20,
)
```

```
def xgb_simple_obj(X_train, X_test, y_train, y_test, config):
    params = config2params(config)
    dtrain = xgb.DMatrix(X_train, label=y_train)
    start_time = time.time()
    _n_estimators = params.pop("n_estimators")
    model = xgb.train(params, dtrain, _n_estimators)
    end_time = time.time()
    # get validation loss
    from sklearn.metrics import r2_score
    dtest = xgb.DMatrix(X_test)
    y_test_predict = model.predict(dtest)
    test_loss = 1.0 - r2_score(y_test, y_test_predict)
    return {"loss": test_loss, "training_cost": end_time-start_time}
```

Needs to be reported in the evaluation function

config_constraints vs *metric_constraints*

Does the calculation of the constraints **relies on the evaluation procedure in the metric function**?



Note: This type of constraint can be checked before evaluation. So if a config does not satisfy *config_constraints,* it will not be evaluated (which saves computation).

```python
def my_model_size(config):
    return config["n_estimators"] * config["max_leaves"]
```

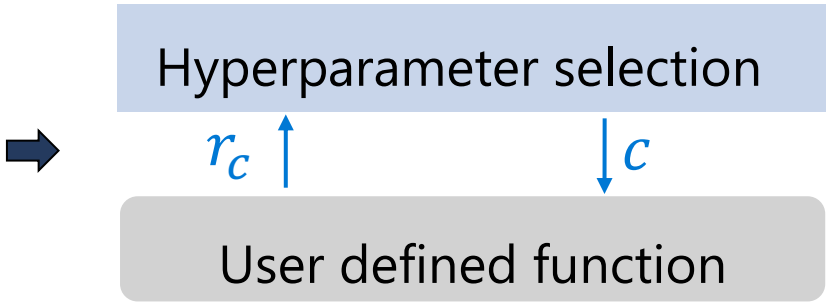
```python
def xgb_simple_obj(X_train, X_test, y_train, y_test, config):
    params = config2params(config)
    dtrain = xgb.DMatrix(X_train, label=y_train)
    start_time = time.time()
    _n_estimators = params.pop("n_estimators")
    model = xgb.train(params, dtrain, _n_estimators)
    end_time = time.time()
    # get validation loss
    from sklearn.metrics import r2_score
    dtest = xgb.DMatrix(X_test)
    y_test_predict = model.predict(dtest)
    test_loss = 1.0 - r2_score(y_test, y_test_predict)
    return {"loss": test_loss, "training_cost": end_time-start_time}
```

Tune User Defined Function: Parallel Tuning

Inputs:

1. Resource budget
2. Search space
3. User defined function

- More constraints?
- Enable parallel tuning?



Outputs: Best config

Parallel Tuning

- *resource_per_trial:* a dict of hardware resources to allocate per trial

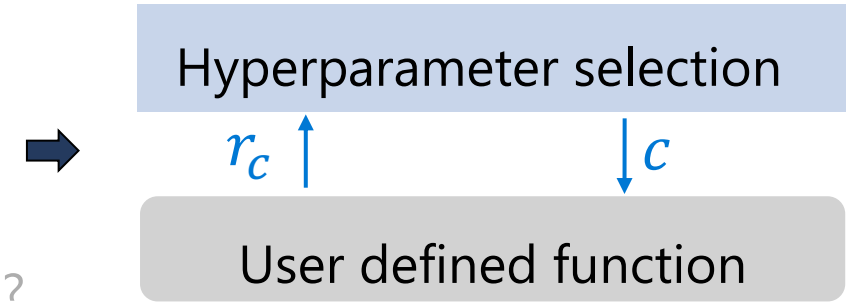
Required
Recommended

```
# require: pip install flaml[ray]
# recommended: pip install flaml[blendsearch]
analysis = tune.run(
    evaluate_config,  # the function to evaluate a config
    config=config_search_space,  # the search space defined
    metric="score",
    mode="min",  # the optimization mode, "min" or "max"
    num_samples=-1,  # the maximal number of configs to try, -1 means infinite
    time_budget_s=10,  # the time budget in seconds
    use_ray=True,
    resources_per_trial={"cpu": 2} # limit resources allocated per trial
)
```

Tune User Defined Function: Warm Start

Inputs:

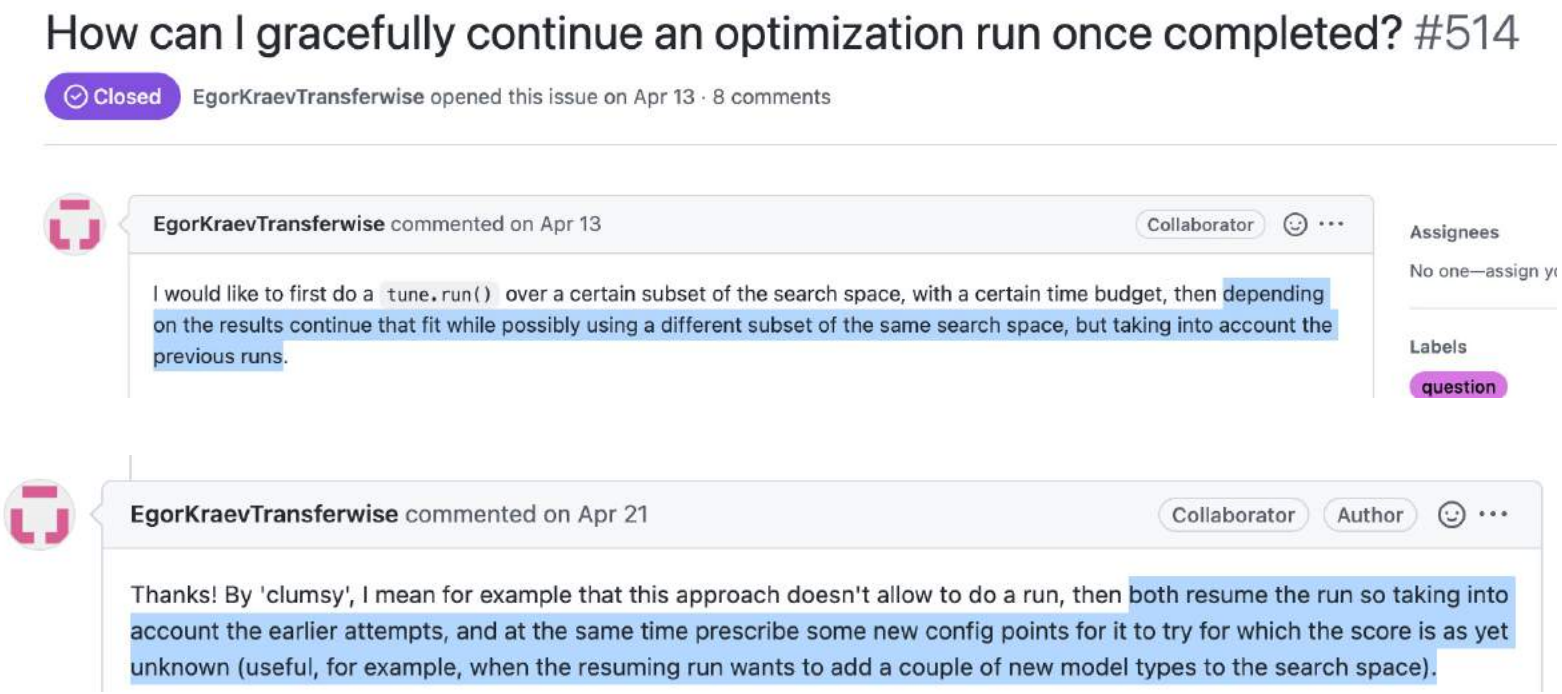
1. Resource budget
2. Search space
3. User defined function
   - More constraints?
   - Enable parallel tuning?
   - Enable warm start?



Outputs: Best config

Warm Start

- *points_to_evaluate:* a list of initial configs to try first
- *evaluated_reward:* a list of reward for the corresponding configs provided in *points_to_evaluate* (must be the same or shorter length than *points_to_evaluate*.)



The need to leverage results from previous runs.

Results from previous runs + some new configs to try first

Warm Start

- *points_to_evaluate:* a list of initial configs to try first
- *evaluated_reward:* a list of reward for the corresponding configs provided in *points_to_evaluate* (must be the same or shorter length than *points_to_evaluate*.)

```
space = {
    "a": tune.uniform(lower=0, upper=0.99),
    "b": tune.uniform(lower=0, upper=3),
}
points_to_evaluate = [
    {"b": .99, "a": 3},
    {"b": .99, "a": 2},
    {"b": .80, "a": 3},
    {"b": .80, "a": 2},
]
evaluated_rewards = [3.99, 2.99]
```

Evaluated configs and results

Additional points to evaluate

```
analysis = tune.run(
    simple_obj,
    config=space,
    mode="max",
    points_to_evaluate=points_to_evaluate,
    evaluated_rewards=evaluated_rewards,
    num_samples=10,
)
```

Warm Start

```
[I 2022-05-22 12:54:13,821] A new study created in memory with name: optuna
[flaml.tune.tune: 05-22 12:54:13] {471} INFO - trial 1 config: {'b': 0.8, 'a': 2.0}
[flaml.tune.tune: 05-22 12:54:13] {471} INFO - trial 2 config: {'b': 0.8, 'a': 3.0}
[flaml.tune.tune: 05-22 12:54:13] {471} INFO - trial 3 config: {'a': 0.7636074368340785, 'b': 0.0622558480782045}
[flaml.tune.tune: 05-22 12:54:13] {471} INFO - trial 4 config: {'a': 0.6273117525770127, 'b': 2.246411647615836}
[flaml.tune.tune: 05-22 12:54:13] {471} INFO - trial 5 config: {'a': 0.4935219421795645, 'b': 0.674389936592543}
[flaml.tune.tune: 05-22 12:54:13] {471} INFO - trial 6 config: {'a': 0.196082236112027 74, 'b': 2.2815921365968763}
[flaml.tune.tune: 05-22 12:54:13] {471} INFO - trial 7 config: {'a': 0.1674197281969101, 'b': 0.2650194425220308}
[flaml.tune.tune: 05-22 12:54:13] {471} INFO - trial 8 config: {'a': 0.6785062201841192, 'b': 2.8601800385848097}
[flaml.tune.tune: 05-22 12:54:13] {471} INFO - trial 9 config: {'a': 0.0039087836646353 07, 'b': 1.53657679015733}
[flaml.tune.tune: 05-22 12:54:13] {471} INFO - trial 10 config: {'a': 0.8044947520355924, 'b': 1.8375782004881644}
```

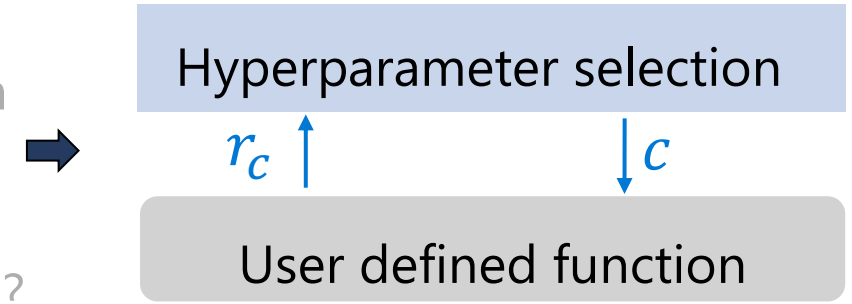
Points to evaluate

Tune User Defined Function: Trial Scheduling

Inputs:

1. Resource budget
2. User defined function
3. Search space

- More constraints?
- Enable parallel tuning?
- Enable warm start?
- Enable trial scheduling



Outputs: Best config

What Is a Scheduler Doing?

A scheduler can help **manage the trials' execution**. It can be used to perform **multi-fidelity evaluation**, or/and **early stopping**.

Trial Scheduling

- *scheduler:* A scheduler for executing the trials.
  - 'flaml': Authentic scheduler in FLAML
  - 'asha': The **A**synchronous **S**uccessive **H**alving **A**lgorithm
  - An instance of the TrialScheduler class from ray.tune
- *resource_attr:* A string to specify the resource dimension used by the scheduler.
- *min_resource:* A float of the minimal resource to use for the resource_attr.
- *max_resource:* A float of the maximal resource to use for the resource_attr.
- *reduction_factor:* A float of the reduction factor used for incremental pruning.

Trial Scheduling

- **scheduler='flaml'**

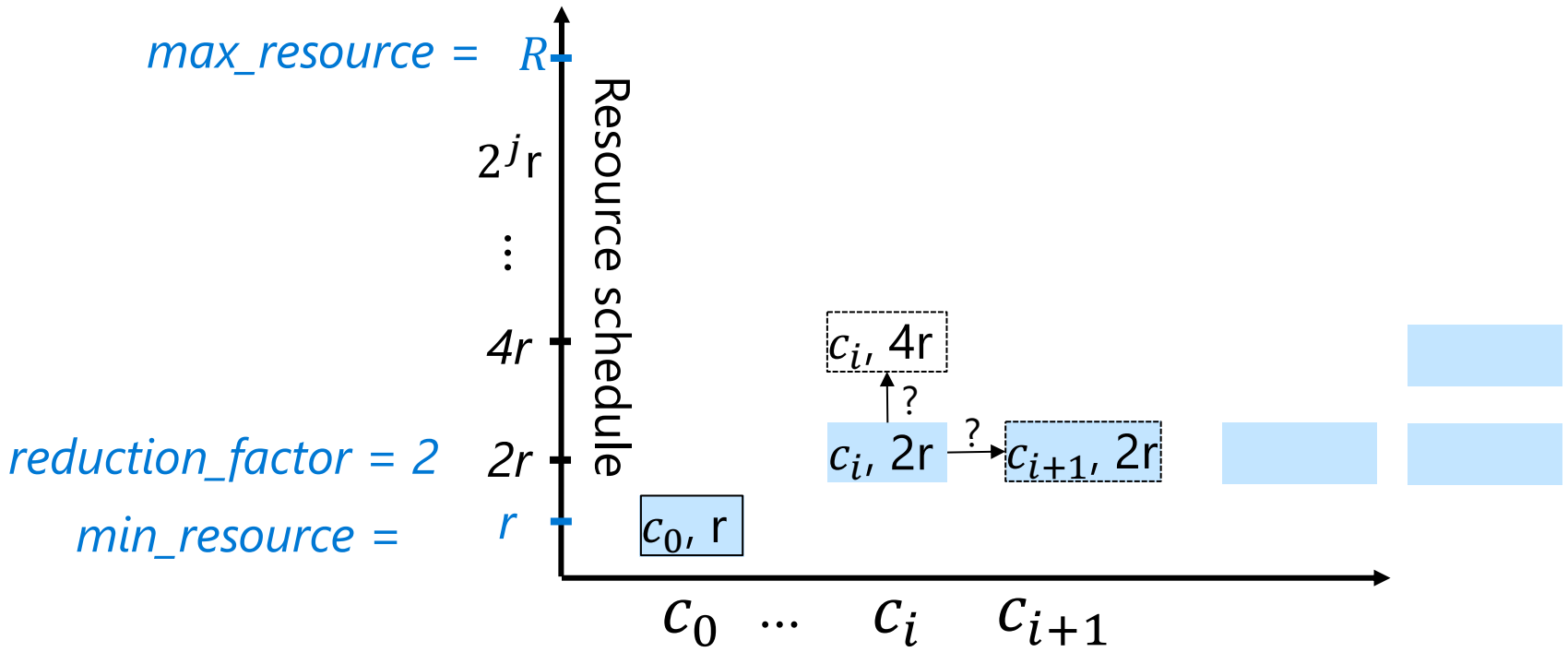
(An authentic scheduler in FLAML)

- **Starts** the search with the minimum resource.

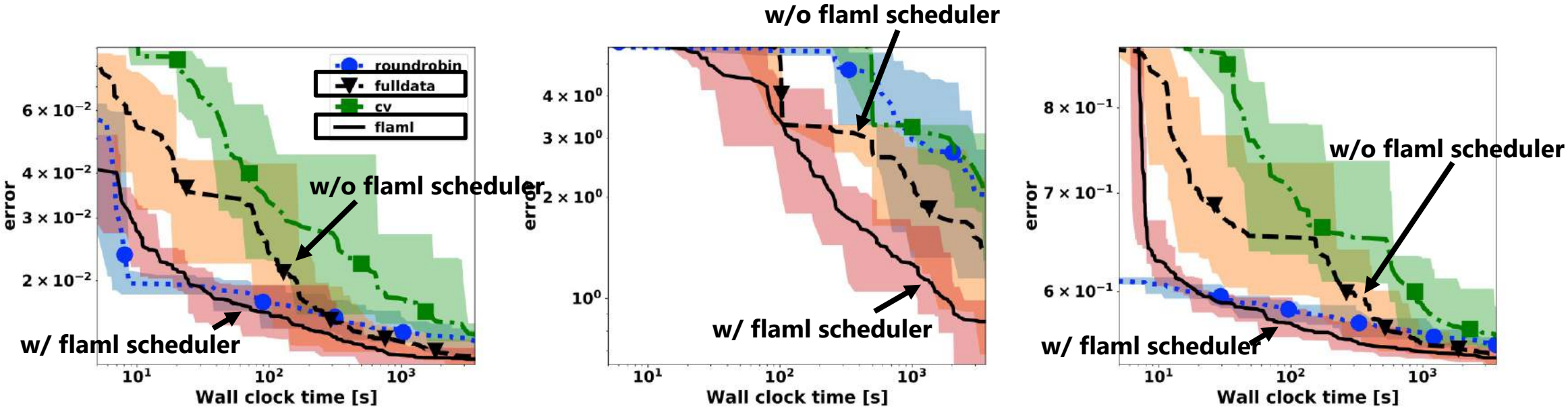
At any time point (before the max resource is reached)

- **Switches** between **HPO with the current resource** and **increasing the resource** for evaluation depending on which leads to faster improvement.

resource_attr = the attribute name for r (e.g., sample size)



Effectiveness of the "flaml" Scheduler



Effectiveness of the authentic scheduler (*scheduler = 'flaml'*) in FLAML [*MLSys'21*]

Trial Scheduling With scheduler='flaml'

```
max_resource = len(y_train)
resource_attr = "sample_size"
min_resource = 1000
analysis = tune.run(
    partial(
        obj_from_resource_attr, resource_attr, X_train, X_test, y_train, y_test
    ),
    config=search_space,
    metric="loss",
    mode="min",
    resource_attr=resource_attr,
    scheduler="flaml",
    max_resource=max_resource,
    min_resource=min_resource,
    reduction_factor=2,
    time_budget_s=10,
    num_samples=-1,
)
```

- Specifying "sample_size" as the resource dimension
- Using the "flaml" scheduler
- Setting lower limit, upper limit and changing factor of the resource

Trial Scheduling With scheduler='flaml'

```
'''Set a evaluation function with resource dimension'''
def obj_from_resource_attr(resource_attr, X_train, X_test,
        y_train, y_test, config):
    from lightgbm import LGBMClassifier
    from sklearn.metrics import accuracy_score

    # in this example sample size is our resource dimension
    resource = int(config[resource_attr])
    sampled_X_train = X_train.iloc[:resource]
    sampled_y_train = y_train[:resource]

    # construct a LGBM model from the config
    # note that you need to first remove the resource_attr field
    # from the config as it is not part of the original search space
    model_config = config.copy()
    del model_config[resource_attr]
    model = LGBMClassifier(**model_config)

    model.fit(sampled_X_train, sampled_y_train)
    y_test_predict = model.predict(X_test)
    test_loss = 1.0 - accuracy_score(y_test, y_test_predict)
    return {"loss": test_loss}
```

- The **suggested config contains the additional resource_attr dimension.**

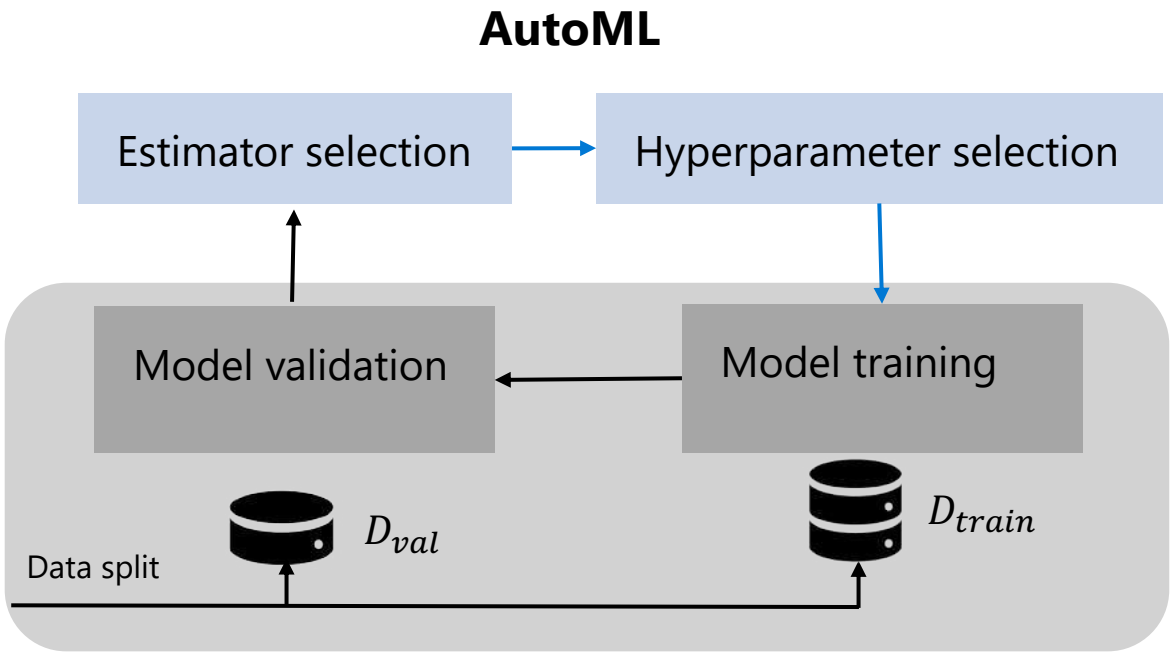
- In the evaluation function, the resource dimension shall be used properly.

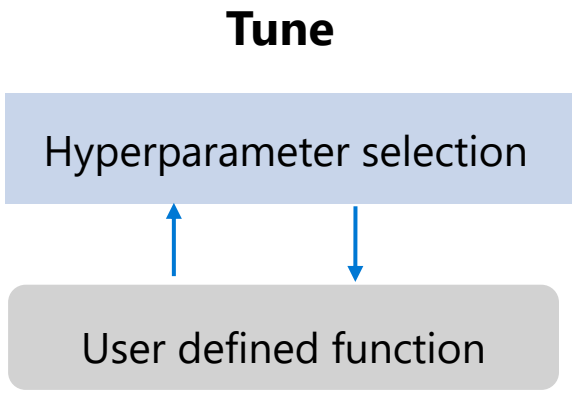
A Use Case of Tune: Validating Strategies

Overall objectives and constraints:
Tuning hyperparameters to find the best strategies (the evaluation of which require expensive simulation), that lead to the highest profitable rate.

Advantages of FLAML.Tune:
- Fits the use case well
- "scheduler" allows for low cost evaluations

" I started playing around with the tuning module - which is really cool by the way and fits really neatly into one of my use cases, so cheers for that! I'm running simulations on market strategies in a given bootstrapped sample.
...
I love the idea of having a resource_attr that increases to validate low cost results on a full dataset. For my case, it's the number of bootstrap iterations that my data should be sampled with each strategy I'm testing. So the low cost is 30 samples, which runs in a few seconds, and the validation/max scenario would be 1000 samples, which takes several minutes. The 30 folds help me exclude obviously bad strategies without having to waste resources on running them 1000 times.
"





Application: Supercharge A/B testing w/automated causal inference

(AutoML: Automating generic regression/classification tasks)

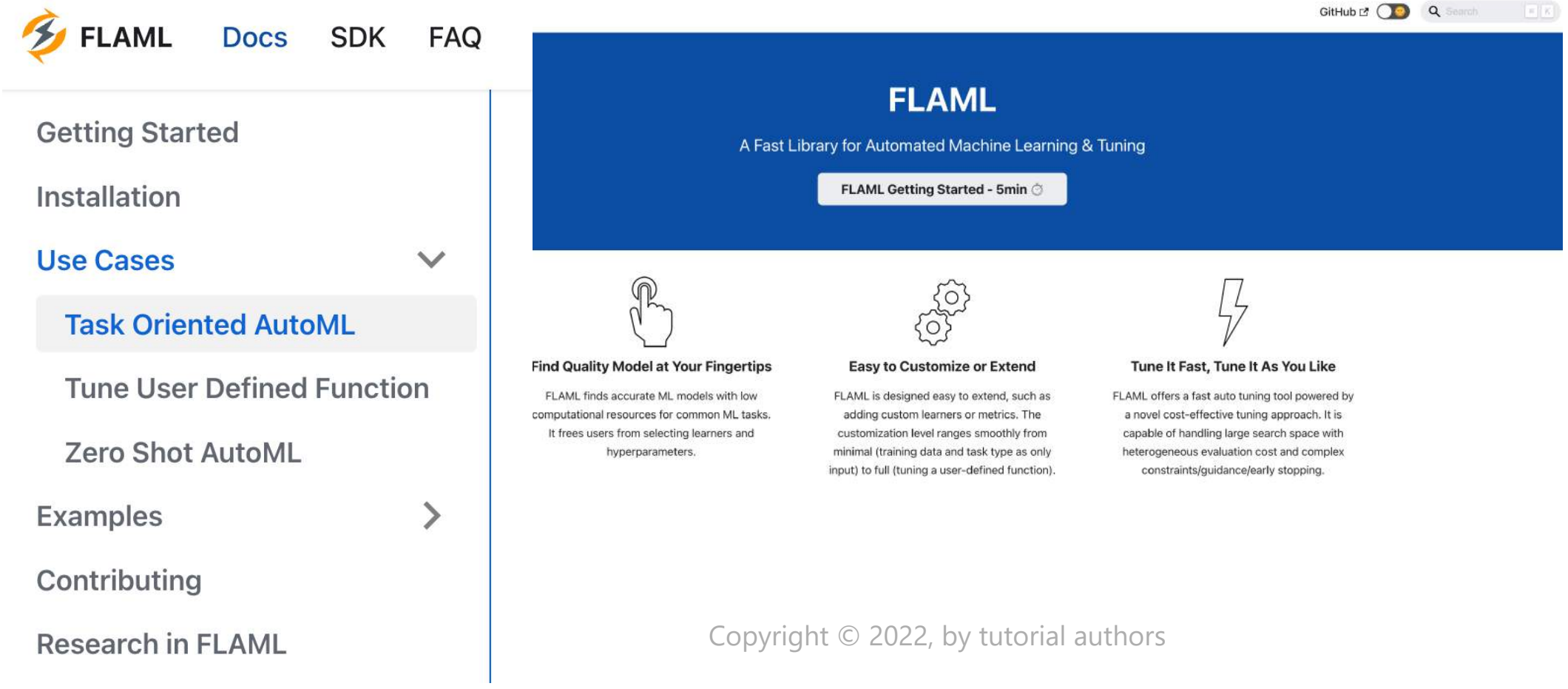
"*First of all**, I use FLAML models FLAML AutoML as the generic regression**. So whenever the models need a regression component, I throw flaml into there. And also if they need the classifier, I use FLAML classifier. The only exception is if my sample is actually random. I use the dummy classifier as a propensity function. But then also **on the second level I use FLAML for model hyperparameter and estimator search**.*"

(Tune: Tuning causal models)

-Head of AI at WISE

Interested in Knowing More ?

- Github: https://github.com/microsoft/FLAML
- Documentation: https://microsoft.github.io/FLAML/



Frequently Asked Questions (FAQ)

https://microsoft.github.io/FLAML/docs/FAQ

- About ***low_cost_partial_config*** in Tune
- How does FLAML handle **imbalanced data** (unequal distribution of target classes in classification task)?
- How to **interpret model performance**? Is it possible for me to **visualize feature importance**, SHAP values, optimization history?

Agenda

First half (9:30 AM – 11:10 AM)

- Overview of AutoML and FLAML (9:30 AM)
- Task-oriented AutoML with FLAML (9:45 AM)
- ML.NET demo (10:30 AM)
- Tune user defined functions with FLAML (10:40 AM)

Break, Q & A

Second half (11:20 AM – 12:30 AM)

- Zero-shot AutoML (11:20 AM)
- Time series forecasting
- Natural language processing (11:45 AM)
- Online AutoML (12:00 PM)
- Fair AutoML
- Challenges and open problems

Agenda

First half (9:30 AM – 11:10 AM)

- Overview of AutoML and FLAML (9:30 AM)
- Task-oriented AutoML with FLAML (9:45 AM)
- ML.NET demo (10:30 AM)
- Tune user defined functions with FLAML (10:40 AM)

Break, Q & A

Second half (11:20 AM – 12:30 AM)

- **Zero-shot AutoML (11:20 AM)**
- **Time series forecasting**
- Natural language processing (11:45 AM)
- Online AutoML (12:00 PM)
- Fair AutoML
- Challenges and open problems

Zero-shot AutoML

```python
from lightgbm import LGBMRegressor
estimator = LGBMRegressor()
print(estimator.get_params())
```

Python

```
{'boosting_type': 'gbdt', 'class_weight': None, 'colsample_bytree': 1.0,
'importance_type': 'split', 'learning_rate': 0.1, 'max_depth': -1,
'min_child_samples': 20, 'min_child_weight': 0.001, 'min_split_gain': 0.0,
'n_estimators': 100, 'n_jobs': -1, 'num_leaves': 31, 'objective': None,
'random_state': None, 'reg_alpha': 0.0, 'reg_lambda': 0.0, 'silent': 'warn',
'subsample': 1.0, 'subsample_for_bin': 200000, 'subsample_freq': 0}
```

What Is Zero-shot AutoML

- Zero-shot AutoML = "no-tuning" AutoML
- Recommend **data-dependent** default configurations at runtime
  - The configuration depends on the dataset (X_train and y_train)

```
from flaml.default import LGBMRegressor
```

```
estimator = LGBMRegressor()  # imported from flaml.default
estimator.fit(X_train, y_train)
print(estimator.get_params())
```

What Is Zero-shot AutoML

- Zero-shot AutoML = "no-tuning" AutoML
- Recommend **data-dependent** default configurations at runtime

```
from flaml.data import load_openml_dataset
X_train, X_test, y_train, y_test = load_openml_dataset(dataset_id=537, data_dir='./')
```

Dataset name*: houses*

```
    estimator = LGBMRegressor()  # imported from flaml.default
    estimator.fit(X_train, y_train)
    print(estimator.get_params())
```

Python

```
{'boosting_type': 'gbdt', 'class_weight': None, 'colsample_bytree': 0.7019911744574896, 'importance_type':
'split', 'learning_rate': 0.022635758411078528, 'max_depth': -1, 'min_child_samples': 2,
'min_child_weight': 0.001, 'min_split_gain': 0.0, 'n_estimators': 4797, 'n_jobs': -1, 'num_leaves': 122,
'objective': None, 'random_state': None, 'reg_alpha': 0.004252223402511765, 'reg_lambda':
0.11288241427227624, 'silent': 'warn', 'subsample': 1.0, 'subsample_for_bin': 200000, 'subsample_freq': 0,
'max_bin': 511, 'verbose': -1}
```

What Is Zero-shot AutoML

- Zero-shot AutoML = "no-tuning" AutoML
- Recommend **data-dependent** default configurations at runtime

```
from flaml.data import load_openml_dataset
X_train, X_test, y_train, y_test = load_openml_dataset(dataset_id=537, data_dir='./')
```

```
from lightgbm import LGBMRegressor
```

```
from flaml.default import LGBMRegressor
```

```
estimator = LGBMRegressor()
estimator.fit(X_train, y_train)
estimator.score(X_test, y_test)
```

Static default, $r^2 = 0.8296$

Data-dependent default, $r^2 = 0.8537$

What Is Zero-shot AutoML

- Zero-shot AutoML = "no-tuning" AutoML
- Recommend data-dependent default configurations at runtime
- Requires mining good hyperparameter configurations across different datasets **offline**

Benefit of Zero-shot AutoML

Training one model only

The decision of hyperparameter configuration is instant

User code remains the same

It requires less input (tuning budget) from the user

The offline preparation can be customized for a domain and leverage the historical tuning data

Concerns of Zero-shot AutoML

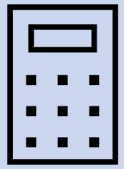
Performance

Robust meta-learning

Combine with HPO

Transparency

Transparent portfolio-based approach



New tasks/estimators/metrics

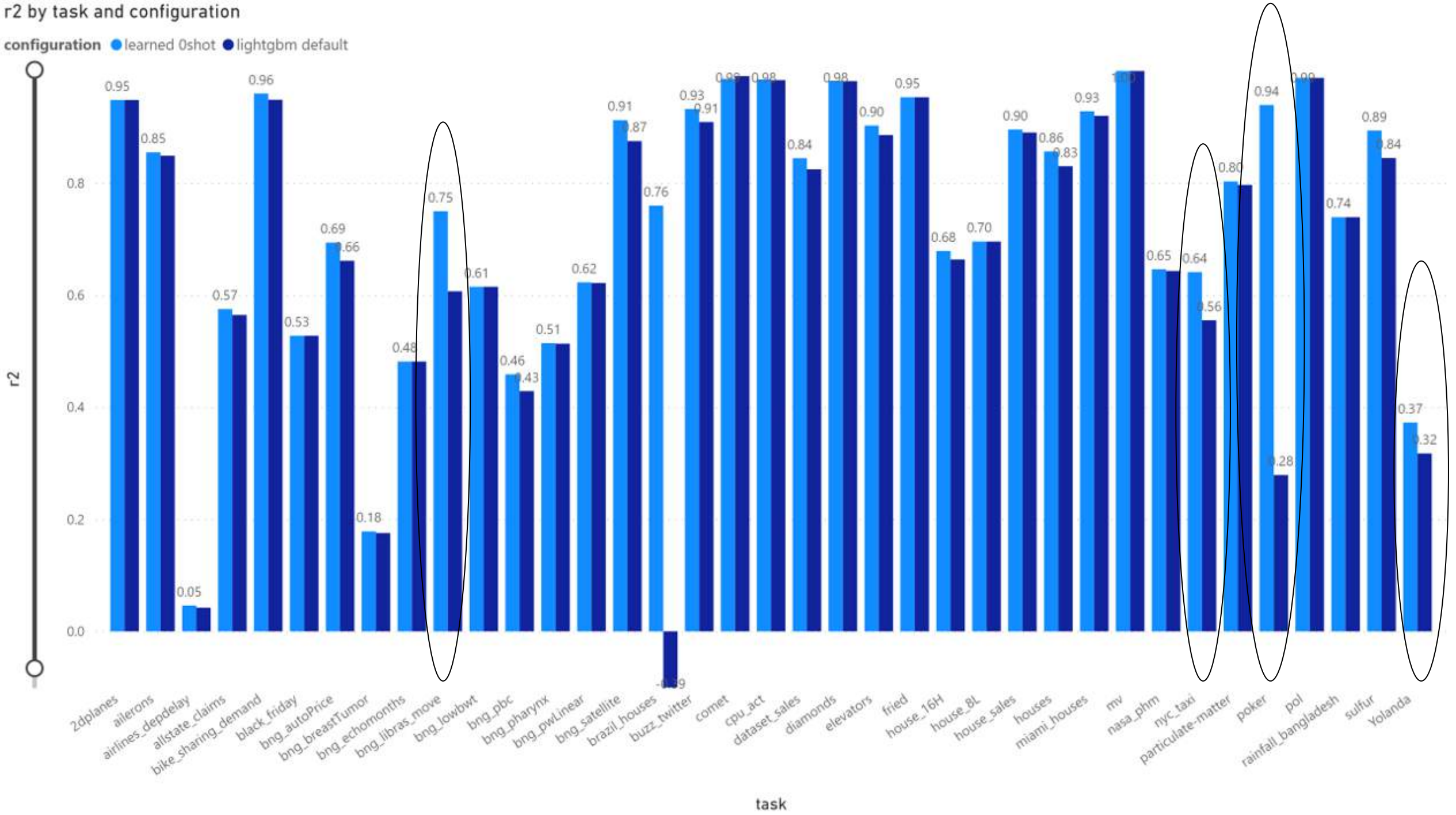
Customizable meta-learning

Data-dependent Defaults vs. Data-agnostic Defaults

Mature libraries already have carefully crafted static defaults

- LightGBM
- XGBoost
- Random Forest

Can data-dependent defaults outperform static defaults consistently?


r2 by task and configuration
configuration learned 0shot lightgbm default
r2
task
0.0
0.2
0.4
0.6
0.8
2dplanes
ailerons
airlines_depdelay
allstate_claims
bike_sharing_demand
black_friday
bng_autoPrice
bng_breastTumor
bng_echomonths
bng_libras_move
bng_lowbwt
bng_pbc
bng_pharynx
bng_pwLinear
bng_satellite
brazil_houses
buzz_twitter
comet
cpu_act
dataset_sales
diamonds
elevators
fried
house_16H
house_8L
house_sales
houses
miami_houses
mv
nasa_phm
nyc_taxi
particulate-matter
poker
pol
rainfall_bangladesh
sulfur
Yolanda

Learned Zero-shot vs. Static Default

The margin is large in many tasks

- bng_libras_move (+24%)
- nyc_taxi (+15%)
- Yolanda (+17%)
- >1% on >67% of the tasks

The static default performs catastrophically in some tasks

- brazil_houses (-0.39 vs 0.76)
- poker (0.28 vs. 0.94)

Learned zero-shot is slightly worse in 1 task

- comet (0.9857 vs. 0.9907, -0.5%)

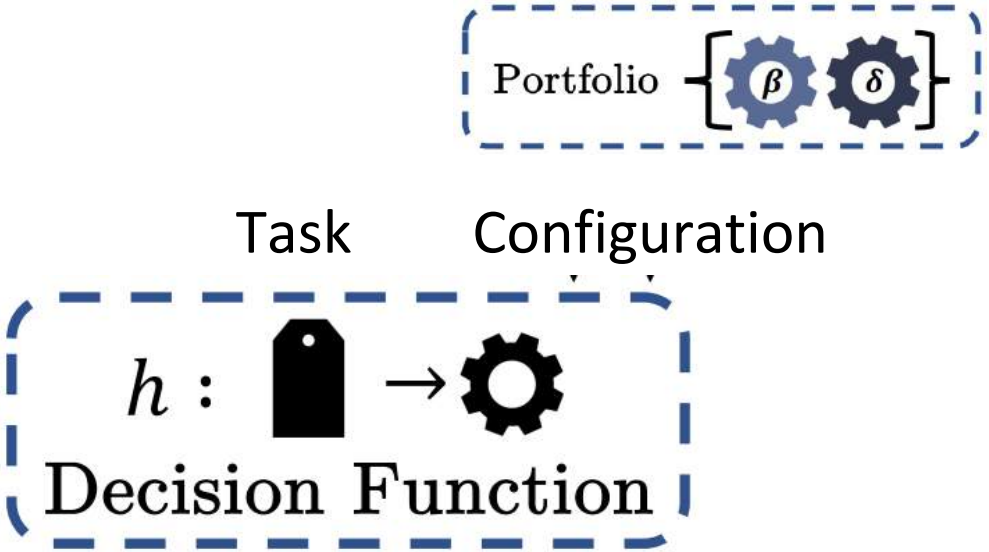
Use Zero-shot AutoML: Import a “Flamlized" Learner

- LGBMClassifier, LGBMRegressor (inheriting LGBMClassifier, LGBMRegressor from lightgbm)
- XGBClassifier, XGBRegressor (inheriting XGBClassifier, XGBRegressor from xgboost)
- RandomForestClassifier, RandomForestRegressor (inheriting from scikit-learn)
- ExtraTreesClassifier, ExtraTreesRegressor (inheriting from scikit-learn)

```
from flaml.default import LGBMRegressor
```

Magic Behind the Scene

- flaml.default.LGBMRegressor inherits lightgbm.LGBMRegressor
- Decide the hyperparameter configurations based on
  - Training data (size and other metafeatures)
  - Offline AutoML results
- The decision is made instantly



Check the Configuration Before Training

```python
from flaml.default import LGBMRegressor

estimator = LGBMRegressor()
hyperparams, _, _, _ = estimator.suggest_hyperparams(X_train, y_train)
print(hyperparams)
```

Python

```
{'n_estimators': 4797, 'num_leaves': 122, 'min_child_samples': 2,
'learning_rate': 0.022635758411078528, 'colsample_bytree':
0.7019911744574896, 'reg_alpha': 0.0042522234025117 65, 'reg_lambda':
0.11288241427227624, 'max_bin': 511, 'verbose': -1}
```

Combine Zero-shot AutoML and HPO

- Further improve the accuracy by tuning

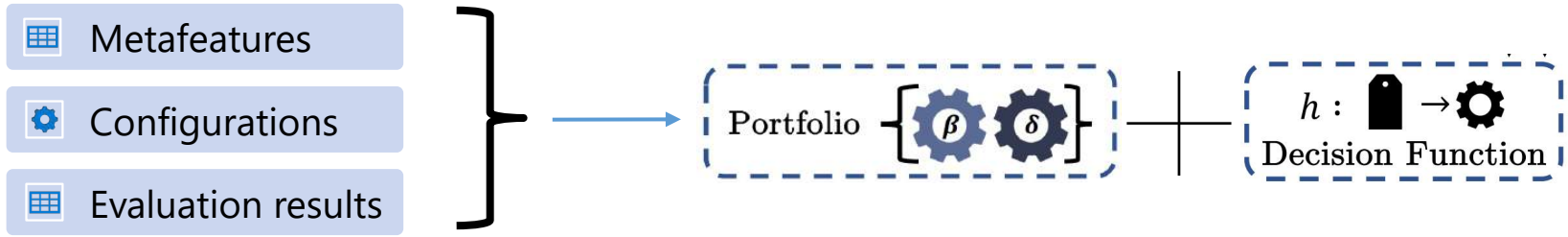
```
from flaml import AutoML

automl = AutoML()
settings = {
    "task": "regression",
    "starting_points": "data",
    "estimator_list": ["lgbm"],
    "time_budget": 600,
}
automl.fit(X_train, y_train, **settings)
```

Use Your Own Meta-learned Defaults

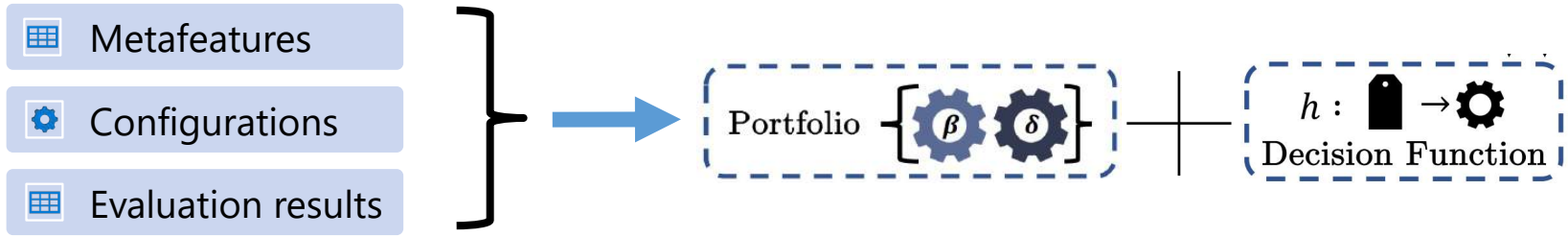
Who may consider customized defaults?

- AutoML providers for a particular domain
- Data scientists or engineers who need to repeatedly train models for similar tasks with varying training data
- Researchers or developers who would like to leverage meta learning for new tasks/estimators/metrics

Meta Learning in FLAML



Learn Data-dependent Defaults




```
my/
        all/metafeatures.csv
        lgbm/
                2dplanes.json
                …
                results.csv
        rf/…
```

python –m flaml.default.portfolio --output my --input my --metafeatures my/all/metafeatures.csv --task binary --estimator lgbm rf

```
my/
        lgbm/binary.json
        rf/binary.json
        all/binary.json
```

Use Your Own Meta-learned Defaults

- Use "flamlized" learner

```
estimator = flaml.default.LGBMRegressor(default_location="location_for_defaults")
```

- Combine zero-shot and HPO

```
automl = AutoML()
automl_settings = {
    "task": "classification",
    "log_file_name": "test/iris.log",
    "starting_points": "data:location_for_defaults",
    "estimator_list": ["lgbm", "xgb_limitdepth", "rf"]
    "max_iter": 0,
}
automl.fit(X_train, y_train, **automl_settings)
```

location_for_defaults/
    all/multiclass.json
    lgbm/multiclass.json
    xgb_limitdepth/multiclass.json
    rf/multiclass.json

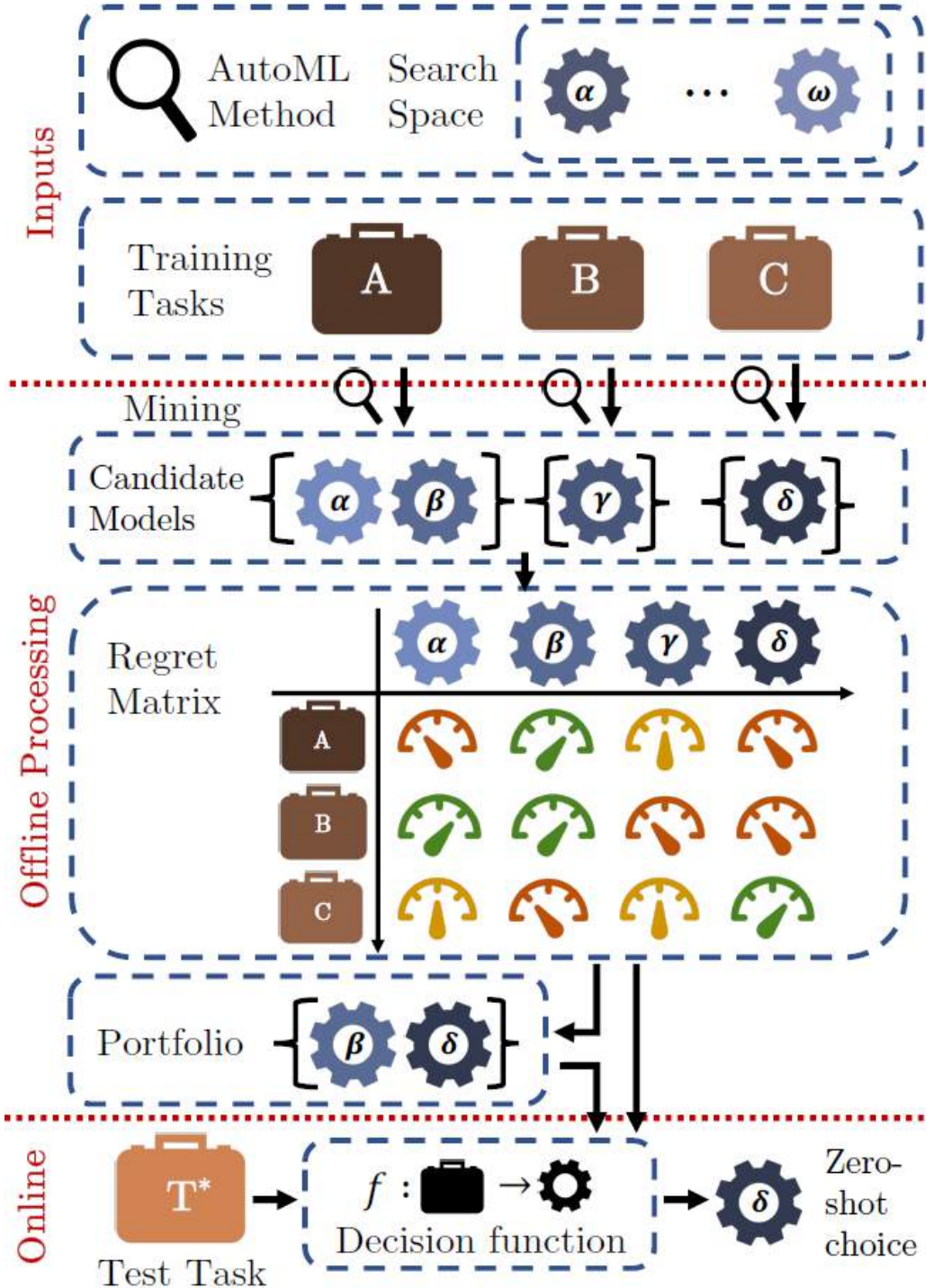
"Flamlize" a Learner

```
import sklearn.ensemble as ensemble
from flaml.default import flamlize_estimator
ExtraTreesClassifier = flamlize_estimator(
    ensemble.ExtraTreesClassifier, "extra_tree", "classification"
)
```

- Share it with others
  - or the *future* yourself
- Update the learned defaults continuously

Reference

- Mining Robust Default Configurations for Resource-constrained AutoML. Moe Kayali, Chi Wang. KDD-AutoML 2022.



Time Series Forecasting

```
pip install "flaml[ts_forecast]"
```

Time Series Forecasting Tasks

Forecast label can be numerical (default) or categorical	• Numerical: task="ts_forecast", or task="ts_forecast_regression" • Categorical: task="ts_forecast_classification"
Data ordered by a datetime column with equal intervals	• Daily: 3-30-2022, 3-31-2022, 4-1-2022, 4-2-2022, … • Weekly: 3-28-2022, 4-4-2022, 4-11-2022, … • Monthly: 3-1-2022, 4-1-2022, 5-1-2022, …
Forecast horizon	• How many future time points to predict

Time Series Forecasting Dataset

SKU	Date	Volume	Month	Price	Temperature
SKU_01	01-01-2022	4	Jan	11	25
SKU_02	01-01-2022	8	Jan	12	25
SKU_01	02-01-2022	20	Feb	10.5	27
SKU_02	02-01-2022	32	Feb	11.2	27
…	…	…	…	…	…

Time series ID (categorical)

Timestamp (datetime)

Forecast label (numerical or categorical)

Time Series Forecasting Dataset

SKU	Date	Volume	Month	Price	Temperature
SKU_01	01-01-2022	4	Jan	11	25
SKU_02	01-01-2022	8	Jan	12	25
SKU_01	02-01-2022	20	Feb	10.5	27
SKU_02	02-01-2022	32	Feb	11.2	27
…	…	…	…	…	…

Time series ID (categorical)

Timestamp (datetime)

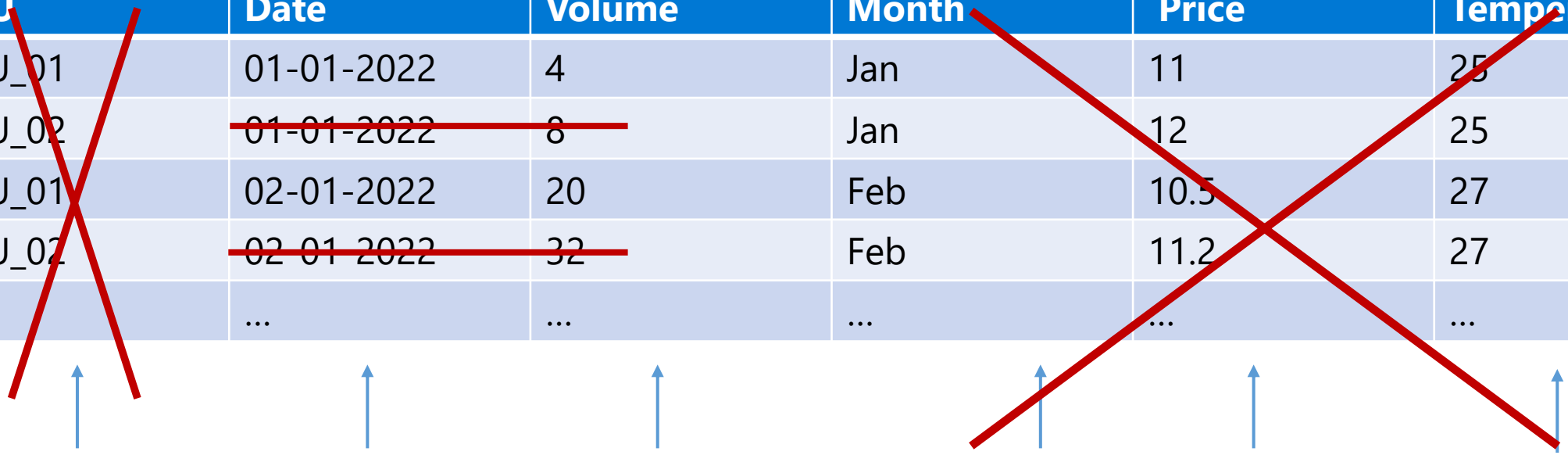
Forecast label (numerical or categorical)

Known features (numerical or categorical)

Unknown features (numerical or categorical)

Time Series Forecasting Dataset

SKU	Date	Volume	Month	Price	Temperature
SKU_01	01-01-2022	4	Jan	11	25
SKU_02	01-01-2022	8	Jan	12	25
SKU_01	02-01-2022	20	Feb	10.5	27
SKU_02	02-01-2022	32	Feb	11.2	27
...	...	...	...	...	...



Time series ID (categorical)

Timestamp (datetime)

Forecast label (numerical or categorical)

Known features (numerical or categorical)

Unknown features (numerical or categorical)

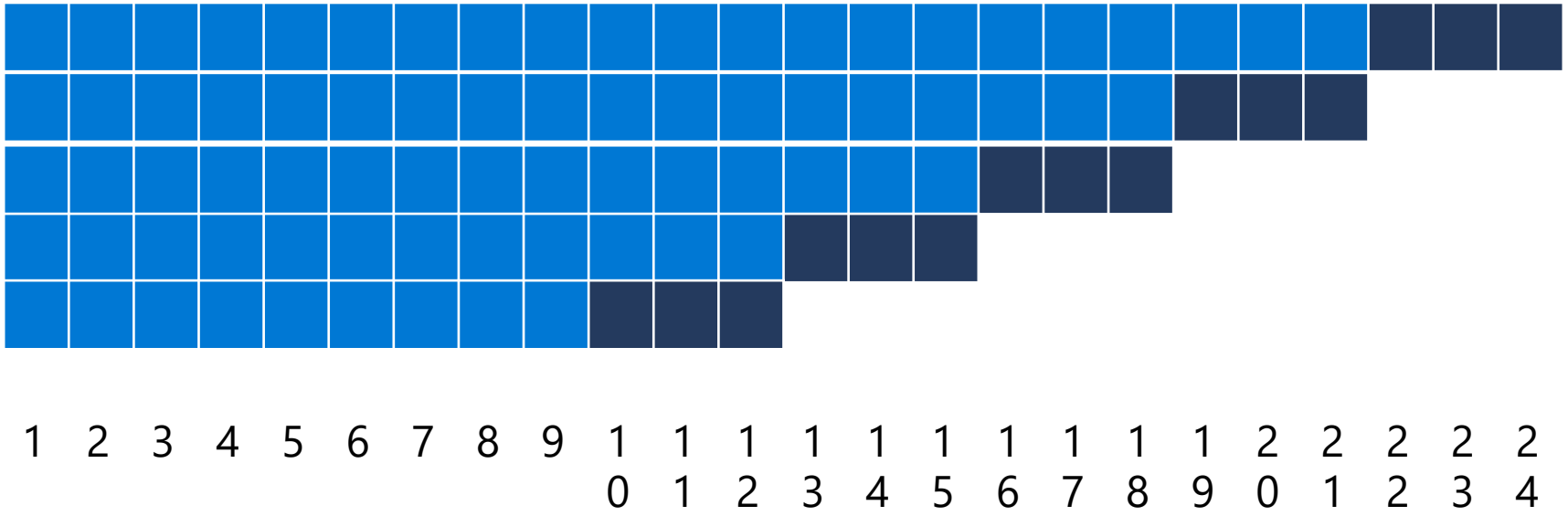
Estimators

Statistical models	Regressors/Classifiers	Neural networks
• Prophet • ARIMA • SARIMAX	• LightGBM • XGBoost • RandomForest • ExtraTrees	• TemporalFusionTransformer (pytorch-forecast)
Id columns ✗ Unknown features ✗	Unknown features ✗	Training cost is high

Data Splitter

- Holdout

- Cross validation
  - Fold 1
  - Fold 2
  - Fold 3
  - Fold 4
  - Fold 5



Example: Prepare Dataset

```python
import statsmodels.api as sm

data = sm.datasets.co2.load_pandas().data
data = data['co2'].resample('MS').mean()
data = data.bfill().ffill()
data = data.to_frame().reset_index()

num_samples = data.shape[0]
time_horizon = 12
split_idx = num_samples - time_horizon
train_df = data[:split_idx]
X_test = data[split_idx:]['index'].to_frame()
y_test = data[split_idx:]['co2']
```

train_df

	index	co2
0	1958-03-01	316.100000
1	1958-04-01	317.200000
2	1958-05-01	317.433333
3	1958-06-01	315.625000
4	1958-07-01	315.625000
...	...	...
509	2000-08-01	367.950000
510	2000-09-01	366.540000
511	2000-10-01	366.725000
512	2000-11-01	368.125000
513	2000-12-01	369.440000

514 rows × 2 columns

X_test

	index
514	2001-01-01
515	2001-02-01
516	2001-03-01
517	2001-04-01
518	2001-05-01
519	2001-06-01
520	2001-07-01
521	2001-08-01
522	2001-09-01
523	2001-10-01
524	2001-11-01
525	2001-12-01

y_test

```
514    370.175
515    371.325
516    372.060
517    372.775
518    373.800
519    373.060
520    371.300
521    369.425
522    367.880
523    368.050
524    369.375
525    371.020
Name: co2, dtype: float64
```

Example: Run AutoML.fit()

```python
from flaml import AutoML

automl = AutoML()
settings = {
    "time_budget": 240,
    "metric": 'mape',
    "task": 'ts_forecast',
    "log_file_name": 'CO2_forecast.log',
    "eval_method": "holdout",
    "seed": 7654321,
}

automl.fit(dataframe=train_df,
           label='co2',
           period=12,
           **settings)
```

List of ML learners in AutoML Run: ['lgbm', 'rf', 'xgboost', 'extra_tree', 'xgb_limitdepth', 'prophet', 'arima', 'sarimax']

```
iteration 0, current learner lgbm
Estimated sufficient time budget=2854s. Estimated
 at 1.0s,  estimator lgbm's best error=0.0621,
iteration 1, current learner lgbm
 at 1.0s,  estimator lgbm's best error=0.0621,
iteration 2, current learner lgbm
 at 1.0s,  estimator lgbm's best error=0.0277,
```

...

```
iteration 553, current learner lgbm
 at 239.2s,        estimator lgbm's best error=0.0022,
iteration 554, current learner extra_tree
 at 239.3s,        estimator extra_tree's best error=0.0016
iteration 555, current learner rf
 at 239.3s,        estimator rf's best error=0.0016,
iteration 556, current learner arima
 at 240.0s,        estimator arima's best error=0.0033,
retrain sarimax for 0.7s
retrained model: <statsmodels.tsa.statespace.sarimax.SARIMA
fit succeeded
Time taken to find the best model: 188.97322726249695
```

Example: Check Result

```python
    ''' retrieve best config and best learner'''
    print('Best ML leaner:', automl.best_estimator)
    print('Best hyperparmeter config:', automl.best_config)
    print(f'Best mape on validation data: {automl.best_loss}')
    print(f'Training duration of best run: {automl.best_config_train_time}s')
```

Python

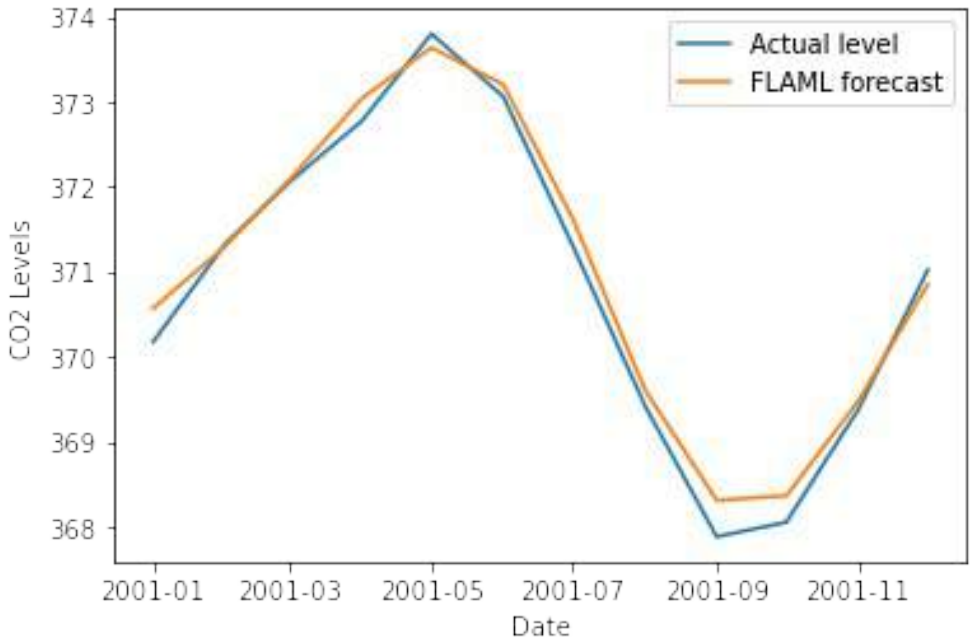
```
Best ML leaner: sarimax
Best hyperparmeter config: {'p': 8.0, 'd': 0.0, 'q': 8.0, 'P': 6.0, 'D': 3.0, 'Q': 1.0, 's': 6}
Best mape on validation data: 0.00043466573064228554
Training duration of best run: 0.6672513484954834s
```

Example: Prediction

```
flaml_y_pred = automl.predict(X_test)
print(f"Predicted labels\n{flaml_y_pred}")
```

```
Predicted labels
2001-01-01    370.568362
2001-02-01    371.297747
2001-03-01    372.087653
2001-04-01    373.040897
2001-05-01    373.638221
2001-06-01    373.202665
2001-07-01    371.621574
2001-08-01    369.611740
2001-09-01    368.307775
2001-10-01    368.360786
2001-11-01    369.476460
2001-12-01    370.849193
Freq: MS, Name: predicted_mean, dtype: float64
```

```
import matplotlib.pyplot as plt
plt.plot(X_test, y_test, label='Actual level')
plt.plot(X_test, flaml_y_pred, label='FLAML forecast')
plt.xlabel('Date')
plt.ylabel('CO2 Levels')
plt.legend()
```



Agenda

First half (9:30 AM – 11:10 AM)

- Overview of AutoML and FLAML (9:30 AM)
- Task-oriented AutoML with FLAML (9:45 AM)
- ML.NET demo (10:30 AM)
- Tune user defined functions with FLAML (10:40 AM)

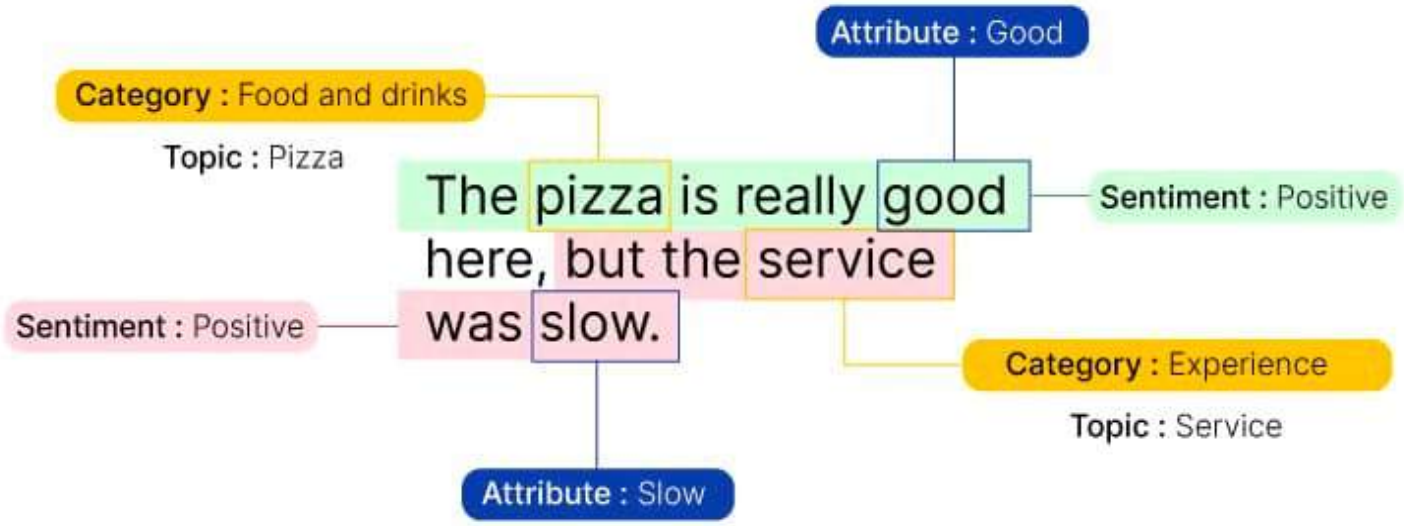
Break, Q & A

Second half (11:20 AM – 12:30 AM)

- Zero-shot AutoML (11:20 AM)
- Time series forecasting
- **Natural language processing (11:45 AM)**
- Online AutoML (12:00 PM)
- Fair AutoML
- Challenges and open problems

Natural Language Processing

```
pip install "flaml[nlp]"
```



Natural Language Processing Tasks by FLAML

Natural language understanding

Sequence classification	• Sentiment classification / Hate speech detection / Document categorization
Sequence regression	• Review rating prediction, book price prediction
Token classification	• Named entity recognition / POS tagging

Natural language generation

Multiple choice classification	• Multiple choice classification
Seq2seq	• Text summarization

FLAML NLP Backbone: Fine-Tuning Language Models

BERT Explained: State of the art language model for NLP

BERT (Bidirectional Encoder Representations from Transformers) is a recent paper published by researchers at Google AI Language. It has caused a stir in the Machine Learning community by presenting state-of-the-art results in a wide variety of NLP tasks, including Question Answering (SQuAD v1.1), Natural Language Inference (MNLI), and others.

Meet GPT-3. It Has Learned to Code (and Blog and Argue).

The latest natural-language system generates tweets, pens poetry, summarizes emails, answers trivia questions, translates languages and even writes its own computer programs.

- Library for fine-tuning transformer models
- Supports major NLP tasks
- Open access to most state-of-the-art language models: huggingface.co/models

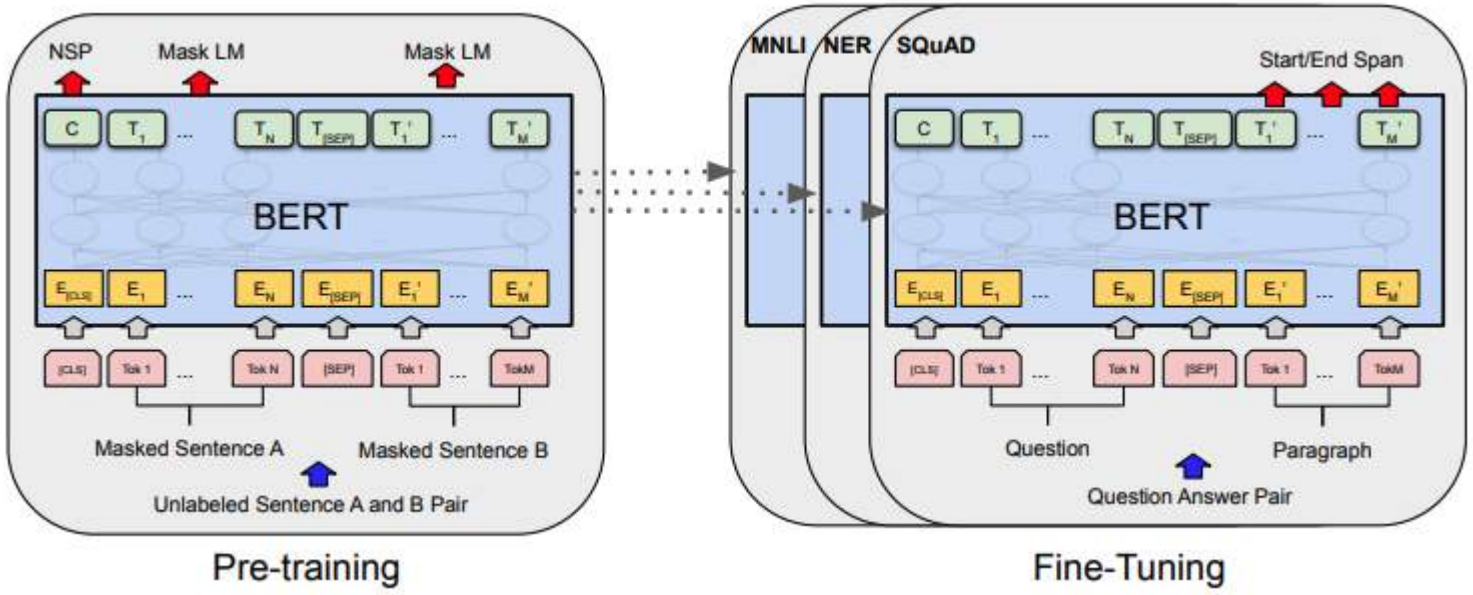
FLAML NLP Backbone: Fine-Tuning Language Models

Stage 1: Pretraining on unlabeled data

Natural language inference

Named entity recognition

QA



Stage 2: Fine tuning on downstream tasks

NLP: A Basic Use Case

FineTuning NLP Models with FLAML Library

1. Introduction

FLAML is a Python library (https://github.com/microsoft/FLAML) designed to automatically produce accurate machine learning models with low computational cost. It is fast and economical. The simple and lightweight design makes it easy to use and extend, such as adding new learners. FLAML can

- serve as an economical AutoML engine,
- be used as a fast hyperparameter tuning tool, or
- be embedded in self-tuning software that requires low latency & resource in repetitive tuning tasks.

2. Sentiment Classification Example

Load data and preprocess

The Stanford Sentiment treebank (SST-2) dataset is a dataset for sentiment classification. First, let's load this dataset into pandas dataframes:

In [1]:
```
from datasets import load_dataset

train_dataset = load_dataset("glue", "sst2", split="train").to_pandas()
dev_dataset = load_dataset("glue", "sst2", split="validation").to_pandas()
test_dataset = load_dataset("glue", "sst2", split="test").to_pandas()
```

```
Reusing dataset glue (/home/xliu127/.cache/huggingface/datasets/glue/sst2/1.0.0/dacbe3125aa31d7f70367a07a8a9e72a5a0bf
eb5fc42e75c9db75b96da6053ad)
Reusing dataset glue (/home/xliu127/.cache/huggingface/datasets/glue/sst2/1.0.0/dacbe3125aa31d7f70367a07a8a9e72a5a0bf
eb5fc42e75c9db75b96da6053ad)
Reusing dataset glue (/home/xliu127/.cache/huggingface/datasets/glue/sst2/1.0.0/dacbe3125aa31d7f70367a07a8a9e72a5a0bf
eb5fc42e75c9db75b96da6053ad)
```

NLP: A Basic Use Case

- **Get the data**
- AutoML using FLAML
- Result

```
from datasets import load_dataset

train_dataset = load_dataset("glue", "sst2", split="train").to_pandas()
dev_dataset = load_dataset("glue", "sst2", split="validation").to_pandas()
test_dataset = load_dataset("glue", "sst2", split="test").to_pandas()
```

```
custom_sent_keys = ["sentence"]          # specify the column names of the i
label_key = "label"                                  # specify the column

X_train, y_train = train_dataset[custom_sent_keys], train_dataset[label_key]
X_val, y_val = dev_dataset[custom_sent_keys], dev_dataset[label_key]
X_test = test_dataset[custom_sent_keys]
```

NLP: A Basic Use Case

- Get the data
- **AutoML using FLAML**
- Result

```
from flaml import AutoML
automl = AutoML()

import ray
if not ray.is_initialized():
    ray.init()
```

```
TIME_BUDGET=1800
automl_settings = {
    "time_budget": TIME_BUDGET,                    # setting the
    "task": "seq-classification",          # setting the task as s
    "fit_kwargs_by_estimator": {
        "transformer": {
            "output_dir": "data/output/",    # setting the outpu
            "model_path": "google/electra-small-discriminator",
        }
    },
    "gpu_per_trial": 1,                    # set to 0 if no GPU is
    "log_file_name": "seqclass.log",       # set the file to save
    "log_type": "all",                     # the log type for tria
    "use_ray": {"local_dir": "data/output/"},
    "n_concurrent_trials": 4,
    "keep_search_state": True,             # keeping the search st
}
```

```
automl.fit(X_train=X_train, y_train=y_train,
    X_val=X_val, y_val=y_val, **automl_settings)
```

NLP: A Basic Use Case

```
In [5]: '''The main flaml automl API'''
automl.fit(X_train=X_train, y_train=y_train, X_val=X_val, y_val=y_val, **automl_settings)
0.0000,num_train__2022-08-13_17-19-11/checkpoint-6315/vocab.txt
(train pid=50443) loading file /data/xliu127/projects/hyperopt/FLAML/notebook/data/output/train_2022-08-13_16-52-17/train_b7e08512_49_FLAML_sample_size=67349,global_max_steps=9223372036854775807,learner=transformer,learning_rate=0.0000,num_train__2022-08-13_17-19-11/checkpoint-6315/tokenizer.json
(train pid=50443) loading file None
(train pid=50443) loading file /data/xliu127/projects/hyperopt/FLAML/notebook/data/output/train_2022-08-13_16-52-17/train_b7e08512_49_FLAML_sample_size=67349,global_max_steps=9223372036854775807,learner=transformer,learning_rate=0.0000,num_train__2022-08-13_17-19-11/checkpoint-6315/special_tokens_map.json
(train pid=50443) loading file /data/xliu127/projects/hyperopt/FLAML/notebook/data/output/train_2022-08-13_16-52-17/train_b7e08512_49_FLAML_sample_size=67349,global_max_steps=9223372036854775807,learner=transformer,learning_rate=0.0000,num_train__2022-08-13_17-19-11/checkpoint-6315/tokenizer_config.json
2022-08-13 17:28:47,144 INFO tune.py:747 -- Total run time: 2189.40 seconds (1803.73 seconds for the tuning loop).
[flaml.automl: 08-13 17:28:52] {3322} INFO - selected model: None
/data/installation/anaconda3/envs/tmp/lib/python3.8/site-packages/transformers/optimization.py:306: FutureWarning:
This implementation of AdamW is deprecated and will be removed in a future version. Use the PyTorch implementation
torch.optim.AdamW instead, or set `no_deprecation_warning=True` to disable this warning
  warnings.warn(
{'loss': 0.417, 'learning_rate': 3.55863617139559e-05, 'epoch': 0.24}
{'loss': 0.2872, 'learning_rate': 3.252648538429503e-05, 'epoch': 0.48}

In [6]: print("The best loss by FLAML: {}".format(automl.best_loss))
The best loss by FLAML: 0.07798165137614677
```

Expected Results

Here are expected results for ELECTRA on various tasks (test set for chunking, dev set for the other tasks). Note that variance in fine-tuning can be quite large, so for some tasks you may see big fluctuations in scores when fine-tuning from the same checkpoint multiple times. The below scores show median performance over a large number of random seeds. ELECTRA-Small/Base/Large are our released models. ELECTRA-Small-OWT is the OpenWebText-trained model from above (it performs a bit worse than ELECTRA-Small due to being trained for less time and on a smaller dataset).

	CoLA	SST	MRPC	STS	QQP	MNLI	QNLI	RTE	SQuAD 1.1	SQuAD 2.0
Metrics	MCC	Acc	Acc	Spearman	Acc	Acc	Acc	Acc	EM	EM
ELECTRA-Large	69.1	96.9	90.8	92.6	92.4	90.9	95.0	88.0	89.7	88.1
ELECTRA-Base	67.7	[illegible]	[illegible]	91.2	91.5	88.8	93.2	82.7	86.8	80.5
ELECTRA-Small	57.0	91.2	88.0	[illegible]	89.0	81.3	88.4	66.7	75.8	70.1
ELECTRA-Small-OWT	56.8	88.3	87.4	86.8	88.3	78.9	87.9	68.5	--	--

FLAML accuracy = 1-0.077 = 0.922

Electra accuracy = 0.912

Fine-Tuning Hyperparameters: FLAML vs Transformers

FLAML's AutoML.fit vs Transformers' hyperparameter_search API:

- Low code implementation
- More customization, e.g., custom metric
- Better performance

hyperparameter_search <source>

```
( hp_space: typing.Union[typing.Callable[[ForwardRef('optuna.Trial')], typing.Dict[str, float]],
NoneType] = None, compute_objective: typing.Union[typing.Callable[[typing.Dict[str, float]], float],
NoneType] = None, n_trials: int = 20, direction: str = 'minimize', backend:
typing.Union[ForwardRef('str'), transformers.trainer_utils.HPSearchBackend, NoneType] = None, hp_name:
typing.Union[typing.Callable[[ForwardRef('optuna.Trial')], str], NoneType] = None, **kwargs ) →
trainer_utils.BestRun
```

Hyperparameter Fine-Tuning with Transformers

```
def preprocess_function(examples):
    inputs = examples[text_column]
    targets = examples[summary_column]
    inputs = [prefix + inp for inp in inputs]
    model_inputs = tokenizer(inputs, padding="max_length", max_length=max_source_length, truncation=True)

    # Setup the tokenizer for targets
    with tokenizer.as_target_tokenizer():
        labels = tokenizer(targets, padding="max_length", max_length=128, truncation=True)

    labels["input_ids"] = [
        [(l if l != tokenizer.pad_token_id else -100) for l in label] for label in labels["input_ids"]
    ]

    model_inputs["labels"] = labels["input_ids"]
    return model_inputs

train_dataset = train_dataset.map(
    preprocess_function,
    batched=True,
)
eval_dataset = valid_dataset.map(
    preprocess_function,
    batched=True
)

from transformers import AutoConfig, AutoModelForSeq2SeqLM

config = AutoConfig.from_pretrained(
    model_path,
)
```

```
def model_init():
    model = AutoModelForSeq2SeqLM.from_pretrained(
        model_path,
        config=config,
        ignore_mismatched_sizes=True
    )
    model.resize_token_embeddings(len(tokenizer))
    if (
        hasattr(model.config, "max_position_embeddings")
        and config.max_position_embeddings < max_source_length
    ):
        model.resize_position_embeddings(max_source_length)
    return model

from transformers import DataCollatorForSeq2Seq

data_collator = DataCollatorForSeq2Seq(
    tokenizer,
    model=model_init(),
    label_pad_token_id=-100,
    pad_to_multiple_of=8,
)

def postprocess_text(preds, labels):
    import nltk
    preds = [pred.strip() for pred in preds]
    labels = [label.strip() for label in labels]

    # rougeLSum expects newline after each sentence
    preds = ["\n".join(nltk.sent_tokenize(pred)) for pred in preds]
    labels = ["\n".join(nltk.sent_tokenize(label)) for label in labels]

    return preds, labels

def compute_metrics(eval_preds):
    import numpy as np
    preds, labels = eval_preds
    if isinstance(preds, tuple):
        preds = preds[0]
    decoded_preds = tokenizer.batch_decode(preds, skip_special_tokens=True)
    labels = np.where(labels != -100, labels, tokenizer.pad_token_id)
    decoded_labels = tokenizer.batch_decode(labels, skip_special_tokens=True)

    decoded_preds, decoded_labels = postprocess_text(decoded_preds, decoded_labels)

    result = metric.compute(predictions=decoded_preds, references=decoded_labels, use_stemmer=True)
    result = {key: value.mid.fmeasure * 100 for key, value in result.items()}

    prediction_lens = [np.count_nonzero(pred != tokenizer.pad_token_id) for pred in preds]
    result["gen_len"] = np.mean(prediction_lens)
    result = {k: round(v, 4) for k, v in result.items()}
    return result
```

```
training_args = TrainingArguments(
    "test", eval_steps=500, disable_tqdm=True)

trainer = Seq2SeqTrainer(
    model_init=model_init,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=eval_dataset,
    tokenizer=tokenizer,
    data_collator=data_collator,
    compute_metrics=compute_metrics,
)

best_run = trainer.hyperparameter_search(
    direction="maximize",
    backend="ray",
    n_trials=-1, # number of trials
    time_budget_s=TIME_BUDGET,
    resources_per_trial={"gpu": 1},
    local_dir="data/output",
    keep_checkpoints_num=1,
)
```

AutoML with Transformers for text summarization: 107 lines

Hyperparameter Fine-Tuning with Transformers

```python
def tokenize_and_align_labels(examples):
    tokenized_inputs = tokenizer(
        examples[text_column_name],
        padding="max_length",
        truncation=True,
        max_length=max_seq_length,
        # We use this argument because the texts in our dataset are lists
        is_split_into_words=True,
    )
    labels = []
    for i, label in enumerate(examples[label_column_name]):
        word_ids = tokenized_inputs.word_ids(batch_index=i)
        previous_word_idx = None
        label_ids = []
        for word_idx in word_ids:
            # Special tokens have a word id that is None. We set the label
            # ignored in the loss function.
            if word_idx is None:
                label_ids.append(-100)
            # We set the label for the first token of each word.
            elif word_idx != previous_word_idx:
                label_ids.append(label_to_id[label[word_idx]])
            # For the other tokens in a word, we set the label to either t
            # the label_all_tokens flag.
            else:
                label_ids.append(-100)
            previous_word_idx = word_idx

        labels.append(label_ids)
    tokenized_inputs["labels"] = labels
    return tokenized_inputs
```

```python
train_dataset = train_dataset.map(
    tokenize_and_align_labels,
    batched=True,
        )
eval_dataset = valid_dataset.map(
    tokenize_and_align_labels,
    batched=True
)

from transformers import AutoConfig, AutoModelForTokenClassification

num_labels = len(label_list)
config = AutoConfig.from_pretrained(
    model_path,
    num_labels=num_labels
)

def model_init():
    model = AutoModelForTokenClassification.from_pretrained(
        model_path,
        config=config,
    )
    return model

from transformers import DataCollatorForTokenClassification

data_collator = DataCollatorForTokenClassification(
    tokenizer,
    pad_to_multiple_of=8
)

def compute_metrics(p):
    import numpy as np
    predictions, labels = p
    predictions = np.argmax(predictions, axis=2)

    # Remove ignored index (special tokens)
    true_predictions = [
        [label_list[p] for (p, l) in zip(prediction, label) if l != -100]
        for prediction, label in zip(predictions, labels)
    ]
    true_labels = [
        [label_list[l] for (p, l) in zip(prediction, label) if l != -100]
        for prediction, label in zip(predictions, labels)
    ]

    results = metric.compute(predictions=true_predictions, references=true_labels)
    return {
        "precision": results["overall_precision"],
        "recall": results["overall_recall"],
        "f1": results["overall_f1"],
        "accuracy": results["overall_accuracy"],
    }
```

```python
training_args = TrainingArguments(
    "test", eval_steps=500, disable_tqdm=True)

trainer = Trainer(
    model_init=model_init,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=eval_dataset,
    tokenizer=tokenizer,
    data_collator=data_collator,
    compute_metrics=compute_metrics,
)

TIME_BUDGET = 3600

best_run = trainer.hyperparameter_search(
    direction="maximize",
    backend="ray",
    n_trials=-1, # number of trials
    time_budget_s=TIME_BUDGET,
    resources_per_trial={"gpu": 1},
    local_dir="data/output",
    keep_checkpoints_num=1,
)
```

AutoML with Transformers
for NER: 133 lines

Hyperparameter Fine-Tuning with Transformers

```python
def preprocess_function(examples):
    inputs = examples[text_column]
    targets = examples[summary_column]
    inputs = [prefix + inp for inp in inputs]
    model_inputs = tokenizer(inputs, padding="max_length", max_length=max_source_length, truncation=True)

    # Setup the tokenizer for targets
    with tokenizer.as_target_tokenizer():
        labels = tokenizer(targets, padding="max_length", max_length=128, truncation=True)

    labels["input_ids"] = [
        [(l if l != tokenizer.pad_token_id else -100) for l in label] for label in labels["input_ids"]
    ]

    model_inputs["labels"] = labels["input_ids"]
    return model_inputs

train_dataset = train_dataset.map(
    preprocess_function,
    batched=True,
)
eval_dataset = valid_dataset.map(
    preprocess_function,
    batched=True
)

from transformers import AutoConfig, AutoModelForSeq2SeqLM

config = AutoConfig.from_pretrained(
    model_path,
)
```

Pre-processing/tokenization

```python
def model_init():
    model = AutoModelForSeq2SeqLM.from_pretrained(
        model_path,
        config=config,
        ignore_mismatched_sizes=True
    )
    model.resize_token_embeddings(len(tokenizer))
    if (
        hasattr(model.config, "max_position_embeddings")
        and config.max_position_embeddings < max_source_length
    ):
        model.resize_position_embeddings(max_source_length)
    return model
```

model

```python
from transformers import DataCollatorForSeq2Seq

data_collator = DataCollatorForSeq2Seq(
    tokenizer,
    model=model_init(),
    label_pad_token_id=-100,
    pad_to_multiple_of=8,
)
```

data collator

```python
def postprocess_text(preds, labels):
    import nltk
    preds = [pred.strip() for pred in preds]
    labels = [label.strip() for label in labels]

    # rougeLSum expects newline after each sentence
    preds = ["\n".join(nltk.sent_tokenize(pred)) for pred in preds]
    labels = ["\n".join(nltk.sent_tokenize(label)) for label in labels]

    return preds, labels
```

Post processing

```python
def compute_metrics(eval_preds):
    import numpy as np
    preds, labels = eval_preds
    if isinstance(preds, tuple):
        preds = preds[0]
    decoded_preds = tokenizer.batch_decode(preds, skip_special_tokens=True)
    labels = np.where(labels != -100, labels, tokenizer.pad_token_id)
    decoded_labels = tokenizer.batch_decode(labels, skip_special_tokens=True)

    decoded_preds, decoded_labels = postprocess_text(decoded_preds, decoded_labels)

    result = metric.compute(predictions=decoded_preds, references=decoded_labels, use_stemmer=True)
    result = {key: value.mid.fmeasure * 100 for key, value in result.items()}

    prediction_lens = [np.count_nonzero(pred != tokenizer.pad_token_id) for pred in preds]
    result["gen_len"] = np.mean(prediction_lens)
    result = {k: round(v, 4) for k, v in result.items()}
    return result
```

computing metrics

```python
training_args = TrainingArguments(
    "test", eval_steps=500, disable_tqdm=True)

trainer = Seq2SeqTrainer(
    model_init=model_init,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=eval_dataset,
    tokenizer=tokenizer,
    data_collator=data_collator,
    compute_metrics=compute_metrics,
)

best_run = trainer.hyperparameter_search(
    direction="maximize",
    backend="ray",
    n_trials=-1, # number of trials
    time_budget_s=TIME_BUDGET,
    resources_per_trial={"gpu": 1},
    local_dir="data/output",
    keep_checkpoints_num=1,
)
```

Trainer

User code for implementing summarization with Transformers

Hyperparameter Fine-Tuning with Transformers

```python
def tokenize_and_align_labels(examples):
    tokenized_inputs = tokenizer(
        examples[text_column_name],
        padding="max_length",
        truncation=True,
        max_length=max_seq_length,
        # We use this argument because the texts in our dataset are lists
        is_split_into_words=True,
    )
    labels = []
    for i, label in enumerate(examples[label_column_name]):
        word_ids = tokenized_inputs.word_ids(batch_index=i)
        previous_word_idx = None
        label_ids = []
        for word_idx in word_ids:
            # Special tokens have a word id that is None. We set the label
            # ignored in the loss function.
            if word_idx is None:
                label_ids.append(-100)
            # We set the label for the first token of each word.
            elif word_idx != previous_word_idx:
                label_ids.append(label_to_id[label[word_idx]])
            # For the other tokens in a word, we set the label to either t
            # the label_all_tokens flag.
            else:
                label_ids.append(-100)
            previous_word_idx = word_idx

        labels.append(label_ids)
    tokenized_inputs["labels"] = labels
    return tokenized_inputs
```

Pre-processing/tokenization

```python
train_dataset = train_dataset.map(
    tokenize_and_align_labels,
    batched=True,
        )
eval_dataset = valid_dataset.map(
    tokenize_and_align_labels,
    batched=True
)

from transformers import AutoConfig, AutoModelForTokenClassification

num_labels = len(label_list)
config = AutoConfig.from_pretrained(
    model_path,
    num_labels=num_labels
)

def model_init():
    model = AutoModelForTokenClassification.from_pretrained(
        model_path,
        config=config,
    )
    return model

from transformers import DataCollatorForTokenClassification

data_collator = DataCollatorForTokenClassification(
    tokenizer,
    pad_to_multiple_of=8
)
```

model

Data collator

```python
def compute_metrics(p):
    import numpy as np
    predictions, labels = p
    predictions = np.argmax(predictions, axis=2)

    # Remove ignored index (special tokens)
    true_predictions = [
        [label_list[p] for (p, l) in zip(prediction, label) if l != -100]
        for prediction, label in zip(predictions, labels)
    ]
    true_labels = [
        [label_list[l] for (p, l) in zip(prediction, label) if l != -100]
        for prediction, label in zip(predictions, labels)
    ]

    results = metric.compute(predictions=true_predictions, references=true_labels)
    return {
        "precision": results["overall_precision"],
        "recall": results["overall_recall"],
        "f1": results["overall_f1"],
        "accuracy": results["overall_accuracy"],
    }
```

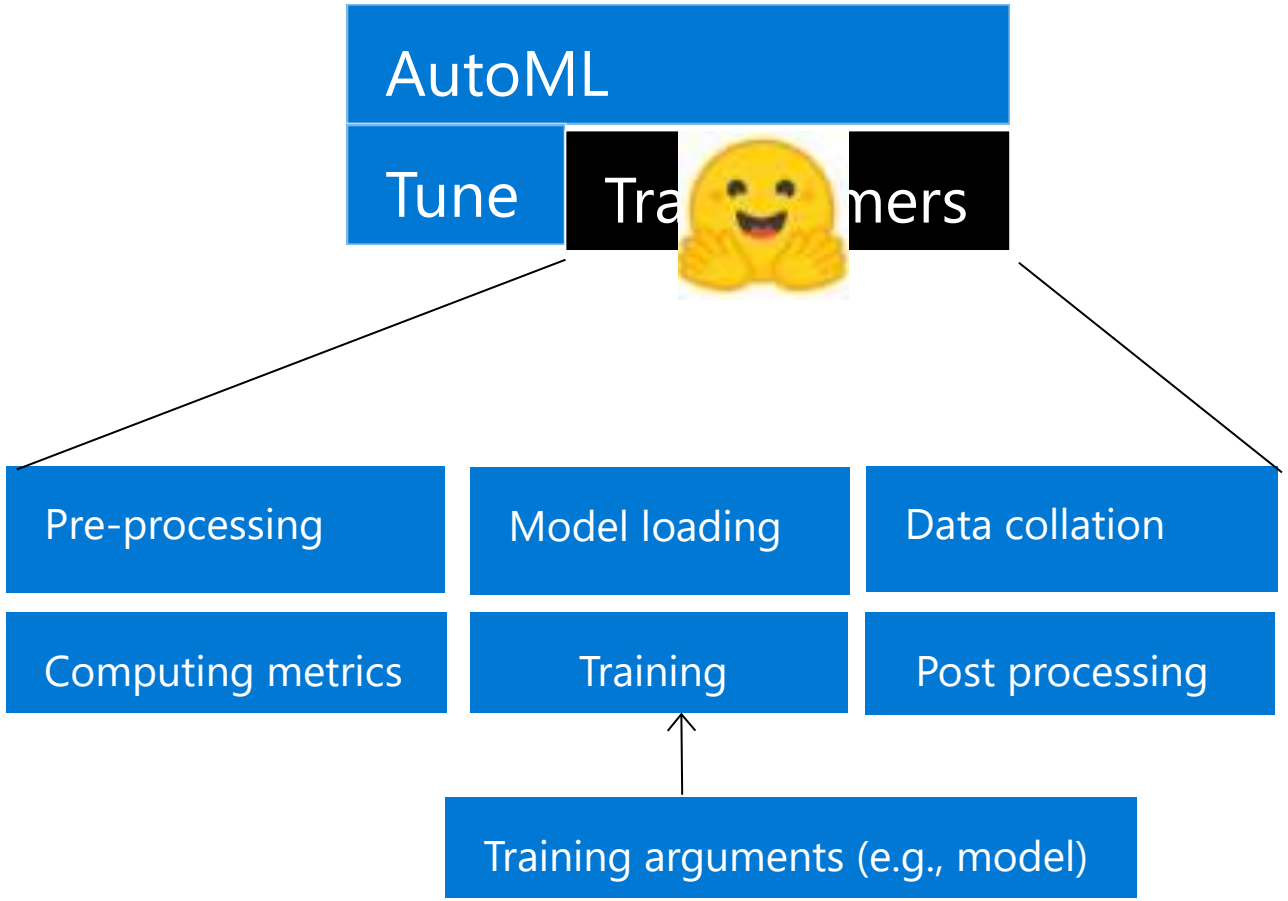
computing metrics

```python
training_args = TrainingArguments(
    "test", eval_steps=500, disable_tqdm=True)

trainer = Trainer(
    model_init=model_init,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=eval_dataset,
    tokenizer=tokenizer,
    data_collator=data_collator,
    compute_metrics=compute_metrics,
)

TIME_BUDGET = 3600

best_run = trainer.hyperparameter_search(
    direction="maximize",
    backend="ray",
    n_trials=-1, # number of trials
    time_budget_s=TIME_BUDGET,
    resources_per_trial={"gpu": 1},
    local_dir="data/output",
    keep_checkpoints_num=1,
)
```

Trainer

User code for implementing NER with Transformers

Low-code Hyperparameter Fine-Tuning with FLAML



Low-code Hyperparameter Fine-Tuning with FLAML

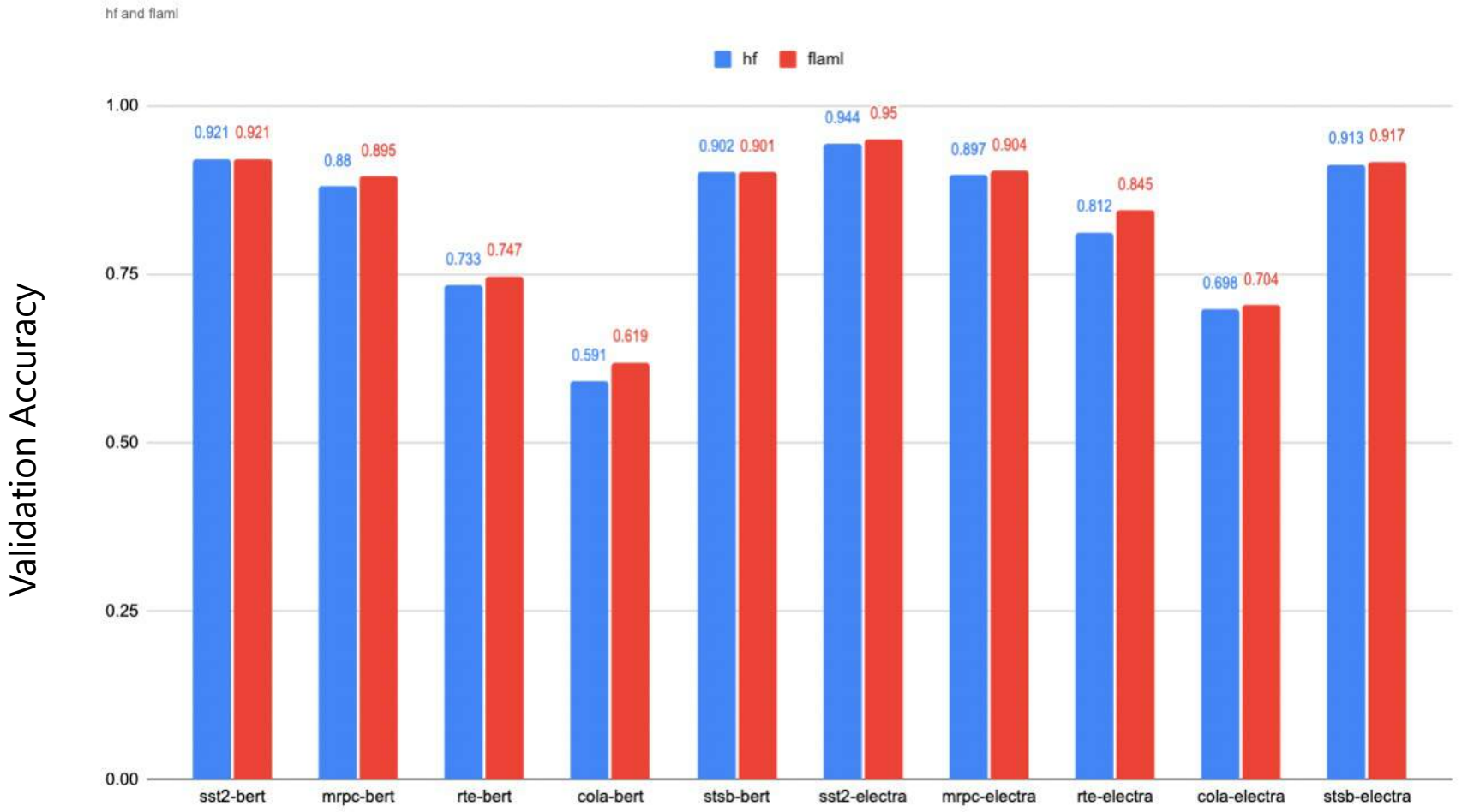
```
from flaml import AutoML
automl = AutoML()

automl_settings = {
    "time_budget": 500,
    "task": "summarization",
    "gpu_per_trial": 1,
    "n_concurrent_trials": 1,
    "metric": "rouge1"
}
automl.fit(X_train=X_train, y_train=y_train,
X_val=X_val, y_val=y_val, **automl_settings)
```

AutoML with FLAML for text summarization: 15 lines

```
from flaml import AutoML
automl = AutoML()

automl_settings = {
    "time_budget": 500,
    "task": "token-classification",
    "gpu_per_trial": 1,
    "n_concurrent_trials": 1,
    "metric": "seqeval:overall_f1"
}
automl.fit(X_train=X_train, y_train=y_train,
X_val=X_val, y_val=y_val, **automl_settings)
```

AutoML with FLAML for NER: 15 lines

Performance: FLAML vs Transformers



FLAML > Transformers

x8

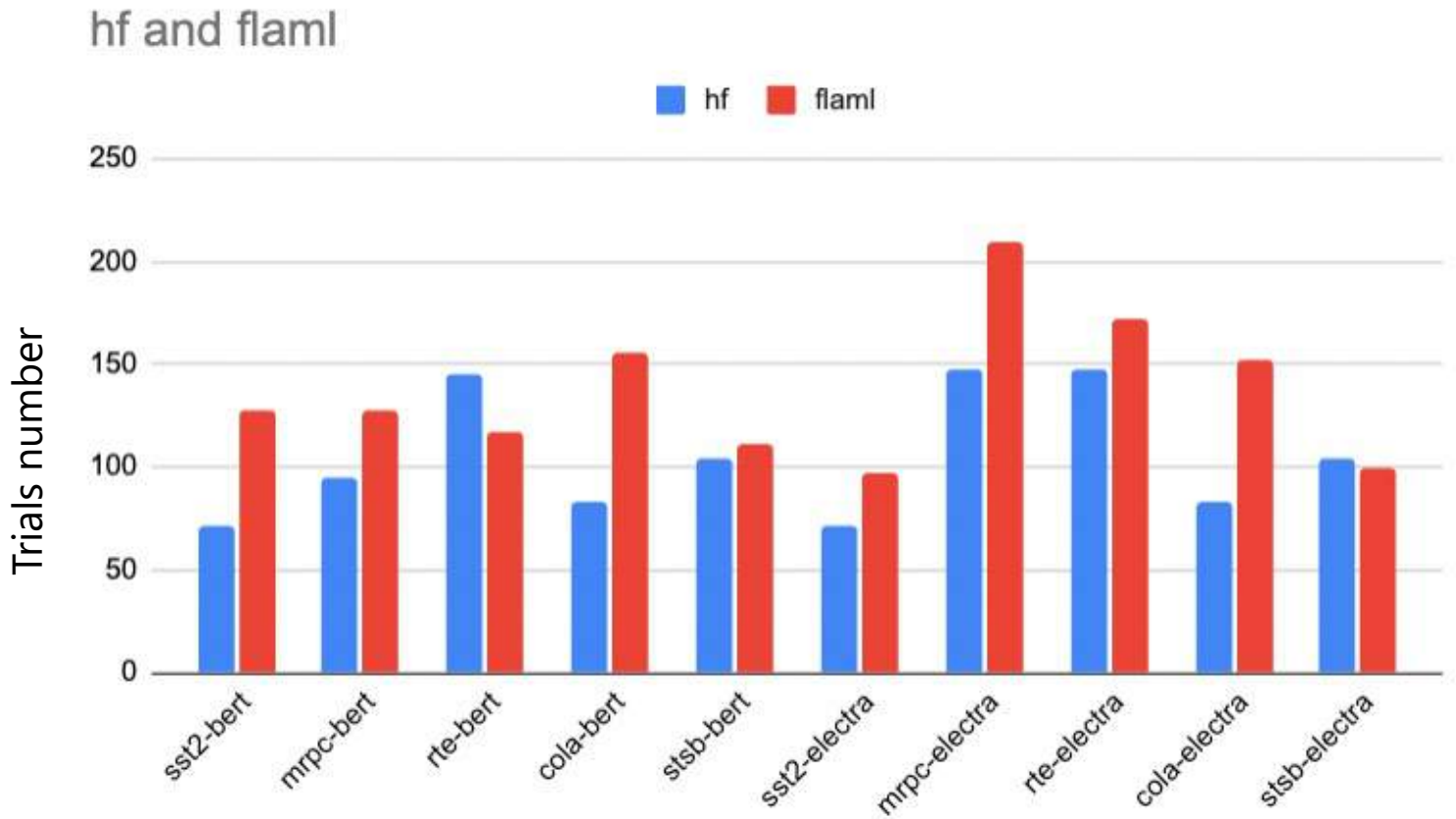
FLAML < Transformers

x1

FLAML = Transformers

x1

Number of Trials: FLAML vs Transformers



FLAML > Transformers

x8

FLAML < Transformers

x2

Open Problems on Fine-Tuning Hyperparameters

- **Model selection**
- Warm starting AutoML/ zero shot AutoML
- **Troubleshooting AutoML failure**
- Optimal search space
- …

Model Selection for Fine-Tuning LM Hyperparameters

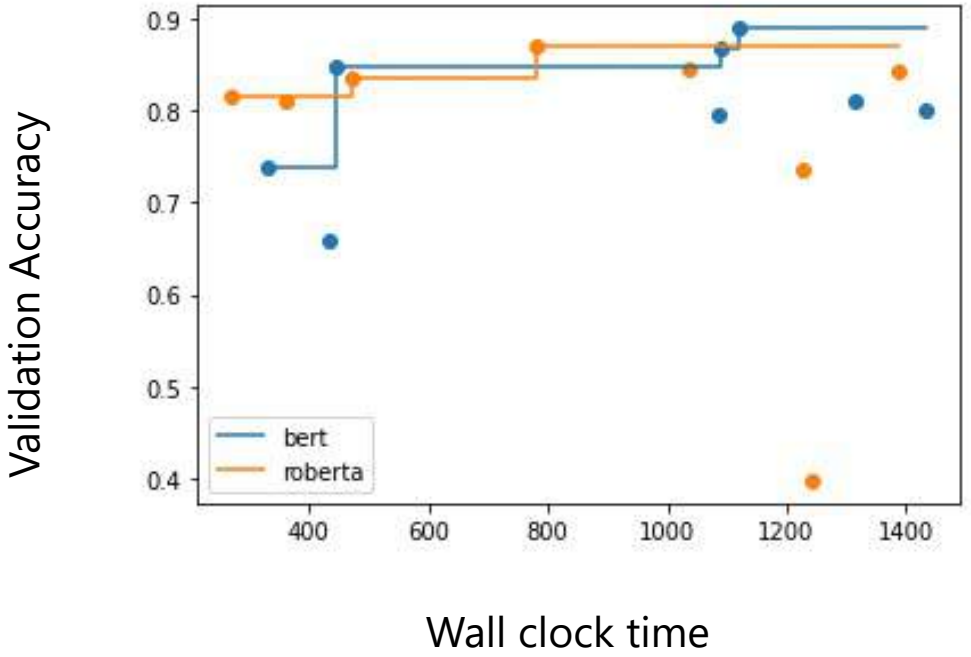
	4	Yi Tay	PaLM 540B		90.4	91.9	94.4/96.0	99.0	88.7/63.6	94.2/93.3	94.1	77.4	95.9	72.9	95.5/90.4
+	5	Zirui Wang	T5 + UDG, Single Model (Google Brain)		90.4	91.4	95.8/97.6	98.0	88.3/63.0	94.2/93.5	93.0	77.9	96.6	69.1	92.7/91.9
+	6	DeBERTa Team - Microsoft	DeBERTa / TuringNLRv4		90.3	90.4	95.7/97.6	98.4	88.2/63.7	94.5/94.1	93.2	77.5	95.9	66.7	93.3/93.8
	7	SuperGLUE Human Baselines	SuperGLUE Human Baselines		89.8	89.0	95.8/98.9	100.0	81.8/51.9	91.7/91.3	93.6	80.0	100.0	76.6	99.3/99.7
+	8	T5 Team - Google	T5		89.3	91.2	93.9/96.8	94.8	88.1/63.3	94.1/93.4	92.5	76.9	93.8	65.6	92.7/91.9
	9	SPoT Team - Google	Frozen T5 1.1 + SPoT		89.2	91.1	95.8/97.6	95.6	87.9/61.9	93.3/92.4	92.9	75.8	93.8	66.9	83.1/82.6
+	10	Huawei Noah's Ark Lab	NEZHA-Plus		86.7	87.8	94.4/96.0	93.6	84.6/55.1	90.1/89.6	89.1	74.6	93.2	58.0	87.1/74.4
+	11	Alibaba PAI&ICBU	PAI Albert		86.1	88.1	92.4/96.4	91.8	84.6/54.7	89.0/88.3	88.8	74.1	93.2	75.6	98.3/99.2
+	12	Infosys : DAWN : AI Research	RoBERTa-iCETS		86.0	88.5	93.2/95.2	91.2	86.4/58.2	89.9/89.3	89.9	72.9	89.0	61.8	88.8/81.5
+	13	Tencent Jarvis Lab	RoBERTa (ensemble)		85.9	88.2	92.5/95.6	90.8	84.4/53.4	91.5/91.0	87.9	74.1	91.8	57.6	89.3/75.6
	14	Zhuiyi Technology	RoBERTa-mtl-adv		85.7	87.1	92.4/95.6	91.2	85.1/54.3	91.7/91.3	88.1	72.1	91.8	58.5	91.0/78.1
	15	Facebook AI	RoBERTa		84.6	87.1	90.5/95.2	90.6	84.4/52.5	90.6/90.0	88.2	69.9	89.0	57.9	91.0/78.1
+	16	Anuar Sharafudinov	AILabs Team, Transformers		82.6	88.1	91.6/94.8	86.8	85.1/54.7	82.8/79.8	88.9	74.1	78.8	100.0	100.0/100.0
+	17	CASIA	INSTALL(ALBERT)-few-shot		76.6	78.4	85.9/92.0	85.6	75.9/35.1	84.3/83.5	74.9	60.9	84.9	-0.4	100.0/50.0
	18	Rakesh Radhakrishnan Menon	ADAPET (ALBERT) - few-shot		76.0	80.0	82.3/92.0	85.4	76.2/35.7	86.1/85.5	75.0	53.5	85.6	-0.4	100.0/50.0
+	19	Timo Schick	iPET (ALBERT) - Few-Shot (32 Examples)		75.4	81.2	79.9/88.8	90.8	74.1/31.7	85.9/85.4	70.8	49.3	88.4	36.2	97.8/57.9
	20	Adrian de Wynter	Bort (Alexa AI)		74.1	83.7	81.9/86.4	89.6	83.7/54.1	49.8/49.0	81.2	70.1	65.8	48.0	96.1/61.5
	21	IBM Research AI	BERT-mtl		73.5	84.8	89.6/94.0	73.8	73.2/30.5	74.6/74.0	84.1	66.2	61.0	29.6	97.8/57.3
	22	Ben Mann	GPT-3 few-shot - OpenAI		71.8	76.4	52.0/75.6	92.0	75.4/30.5	91.1/90.2	69.0	49.4	80.1	21.1	90.4/55.3
	23	SuperGLUE Baselines	BERT++		71.5	79.0	84.8/90.4	73.8	70.0/24.1	72.0/71.3	79.0	69.6	64.4	38.0	99.4/51.4
			BERT		69.0	77.4	75.7/83.6	70.6	70.0/24.1	72.0/71.3	71.7	69.6	64.4	23.0	97.8/51.7

RoBERTa: 84.6

BERT: 69.0

Model Selection for Fine-Tuning LM Hyperparameters

- Non trivial problem. The "best" model does not exist!
  - Spooky-author-identification: BERT outperforms RoBERTa (and Electra, Muppet, etc.)



Troubleshooting AutoML Failure

	WNLI	RTE	MRPC	CoLA	STS-B	SST	QNLI	MNLI
Electra-base, validation								
grid	56.3	**84.1**	92.3/89.2	67.2	**91.5/91.4**	**95.1**	**93.5**	88.6
RS	56.8	82.2	**93.0/90.4**	68.8	90.1/90.2	94.7	93.0	88.9
RS+ASHA	57.2	80.3	93.0/90.3	67.9	91.4/91.3	94.9	93.1	88.6
BO+ASHA	**58.2**	82.6	**93.1/90.4**	**69.4**	91.5/91.3	94.7	93.1	**89.2**
Electra-base, test								
grid	**65.1**	**76.8**	**91.1/87.9**	58.5	**89.7/89.2**	**95.7**	**93.5**	88.3
RS	64.4	75.6	90.7/87.5	63.0	88.0/87.6	95.1	93.0	**88.7**
RS+ASHA	62.6	74.1	90.6/87.3	61.2	89.5/89.1	94.9	92.9	88.5
BO+ASHA	61.6	75.1	90.7/87.4	**64.1**	89.7/89.1	94.8	93.0	**88.7**

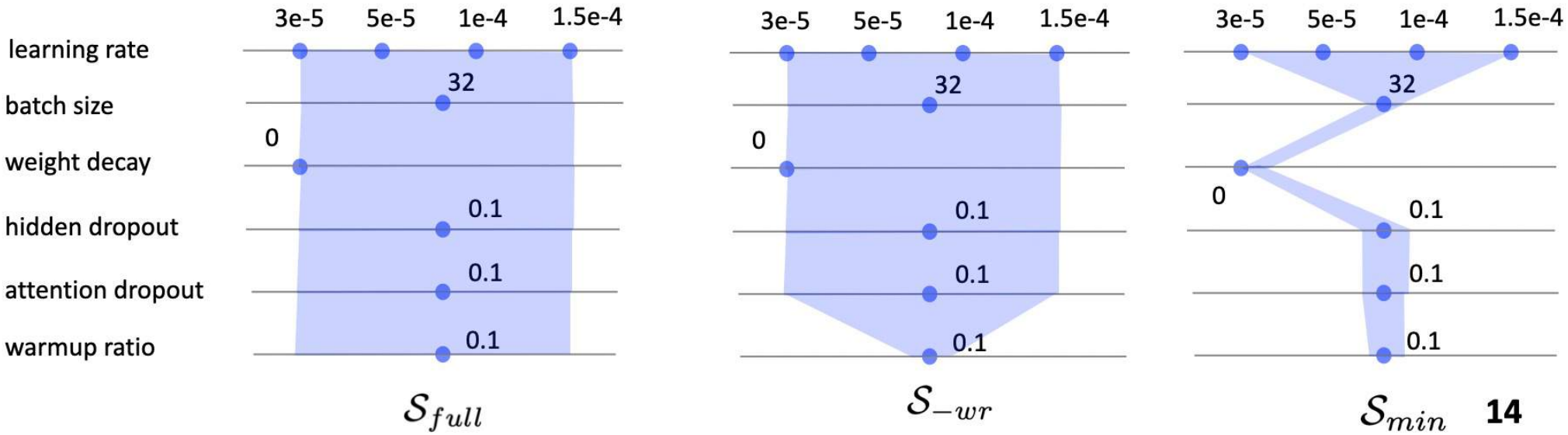
outperform: HPO validation > grid validation, HPO test > grid test

underperform: HPO validation < grid validation

overfit: HPO validation > grid validation, HPO test < grid test

RS, BO, and ASHA underperform recommended HPs [ACL 2021]

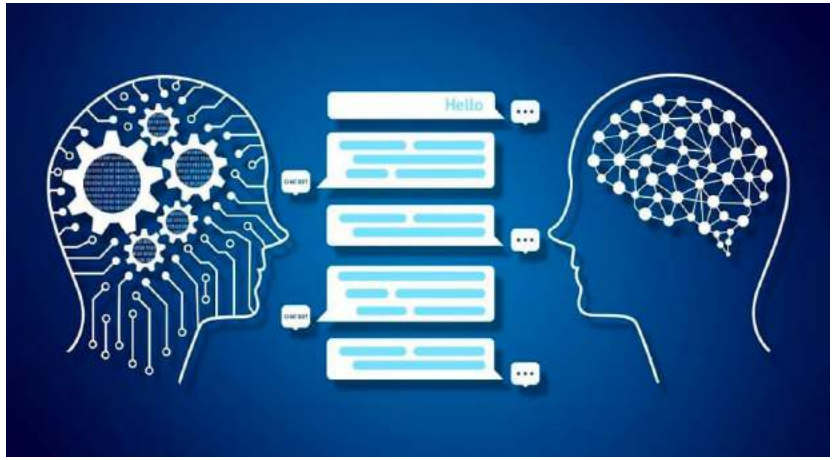
Troubleshooting AutoML Failure



Reducing the search space can effectively reduce overfitting in HPO [ACL 2021]

FLAML NLP: Summary

- An AutoML layer on top of Transformers
- Low-code implementation
- Outperforms Transformers' tuning performance
- Join us to provide a better tool for AutoML for NLP



Agenda

First half (9:30 AM – 11:10 AM)

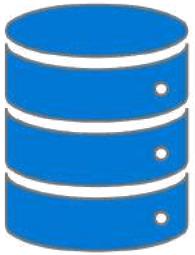
- Overview of AutoML and FLAML (9:30 AM)
- Task-oriented AutoML with FLAML (9:45 AM)
- ML.NET demo (10:30 AM)
- Tune user defined functions with FLAML (10:40 AM)

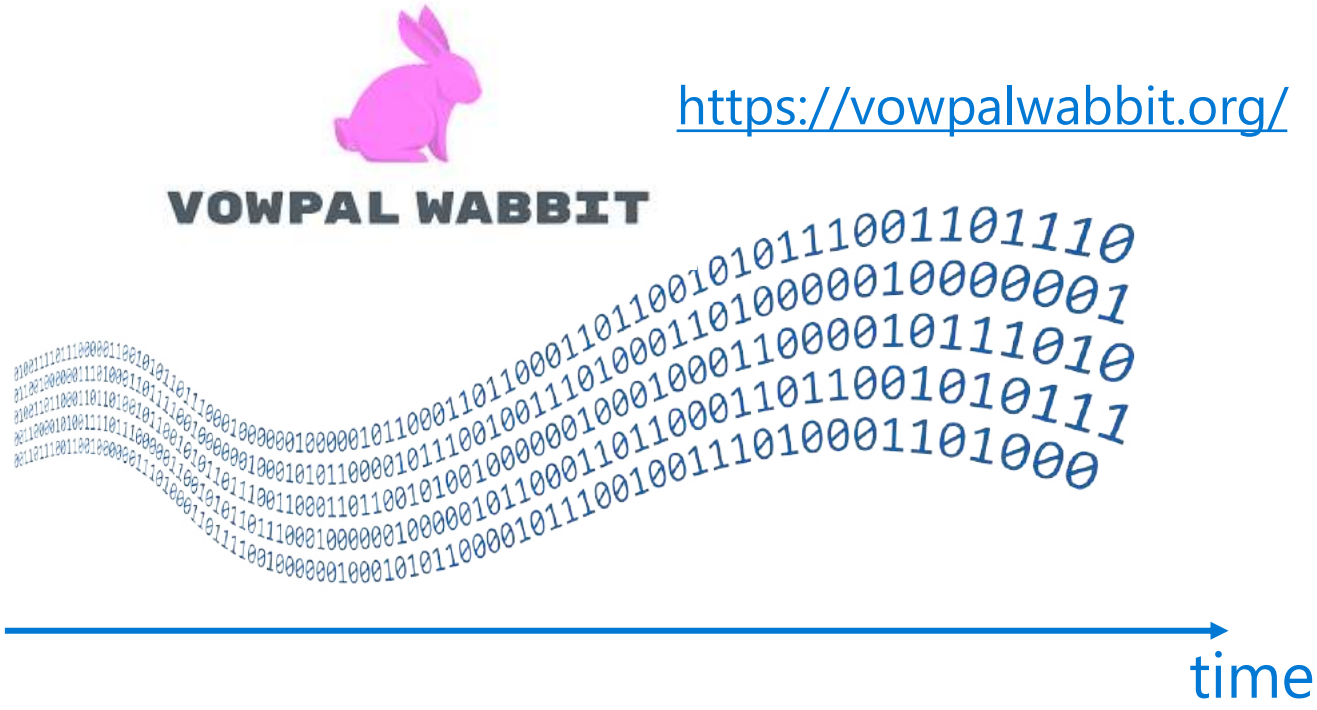
Break, Q & A

Second half (11:20 AM – 12:30 AM)

- Zero-shot AutoML (11:20 AM)
- Time series forecasting
- Natural language processing (11:45 AM)
- **Online AutoML (12:00 PM)**
- **Fair AutoML**
- **Challenges and open problems**

Online AutoML

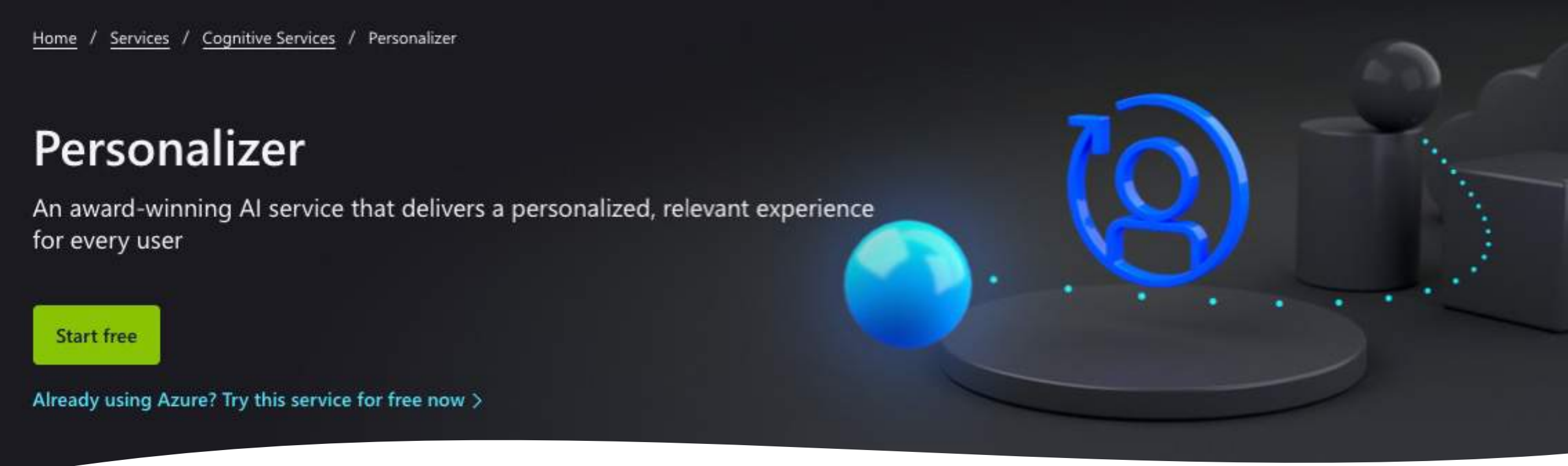


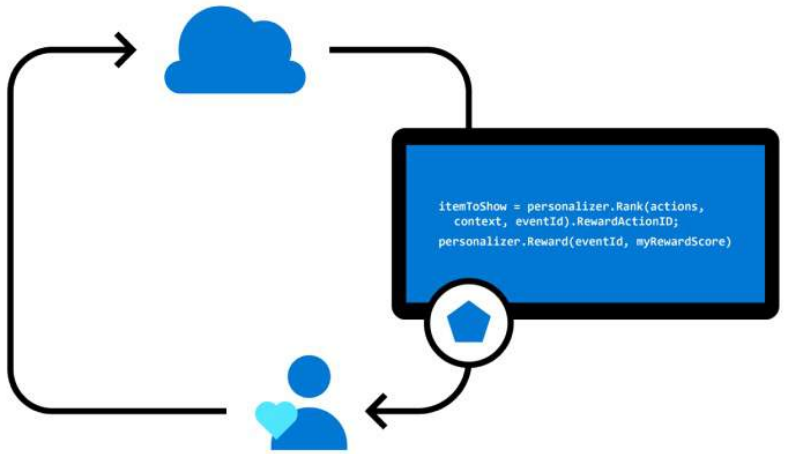


https://vowpalwabbit.org/

Properties of online machine learning

- Data is available in a sequential order
- Training/learning is performed online
- Prediction/decision needs to be made online

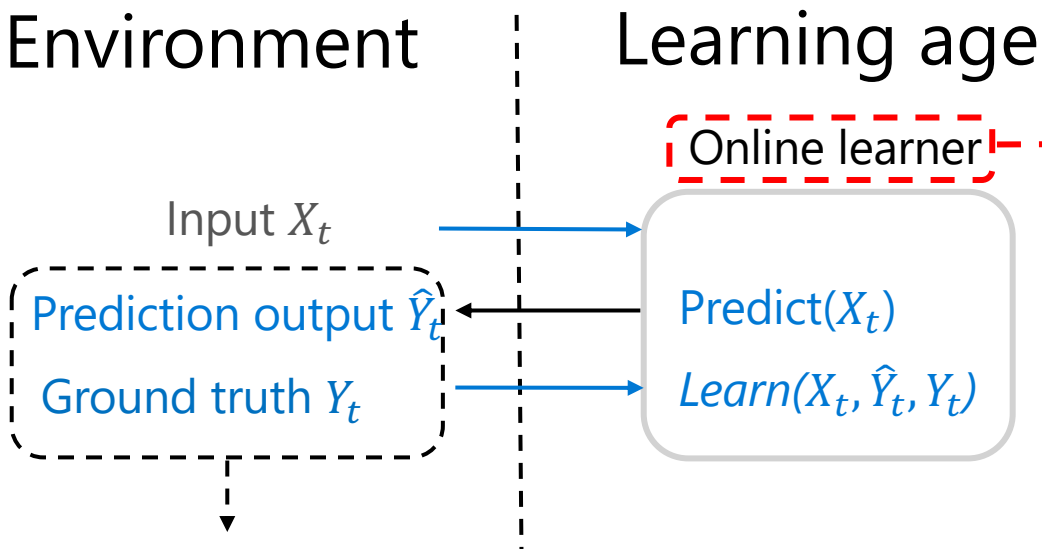




- Improves user experience with real-time learning

https://azure.microsoft.com/en-us/services/cognitive-services/personalizer/

Online ML -> Online AutoML



Need AutoML

Hyperparameters:

- featurization choices
- learning rate
- regularization
- …

Online evaluation (e.g., progressive validation loss, cumulative regret)

Online Namespace Interactions Tuning

- Features grouped into namespaces in VW:

```
vw example: 8.170000076293945 |a 0:10.000000 1:7.000000|b 2:3.000000 3:4.000000|c 4:nan 5:6.330000|d 6:0.136000
7:7.330000|e 8:7.010000 9:6.980000|f 10:0.003000 11:7.000000|g 12:9.700000 13:12.300000|h 14:1021.700012
15:0.000000|i 16:58.000000
```

- Sometimes adding feature interactions is helpful.
- How to automatically decide which namespace interactions to use? (A double exponentially large search space)

Online AutoML ?

- Unique challenges of online AutoML (which make conventional AutoML methods not directly applicable)

Sharp computation constraints (only a constant factor more computation is allowed at any time point)

Vastly different scales of data volume

Learning algorithms are being evaluated constantly

Online AutoML

Research question

How can we best design an efficient online automated machine learning algorithm?

Key challenge

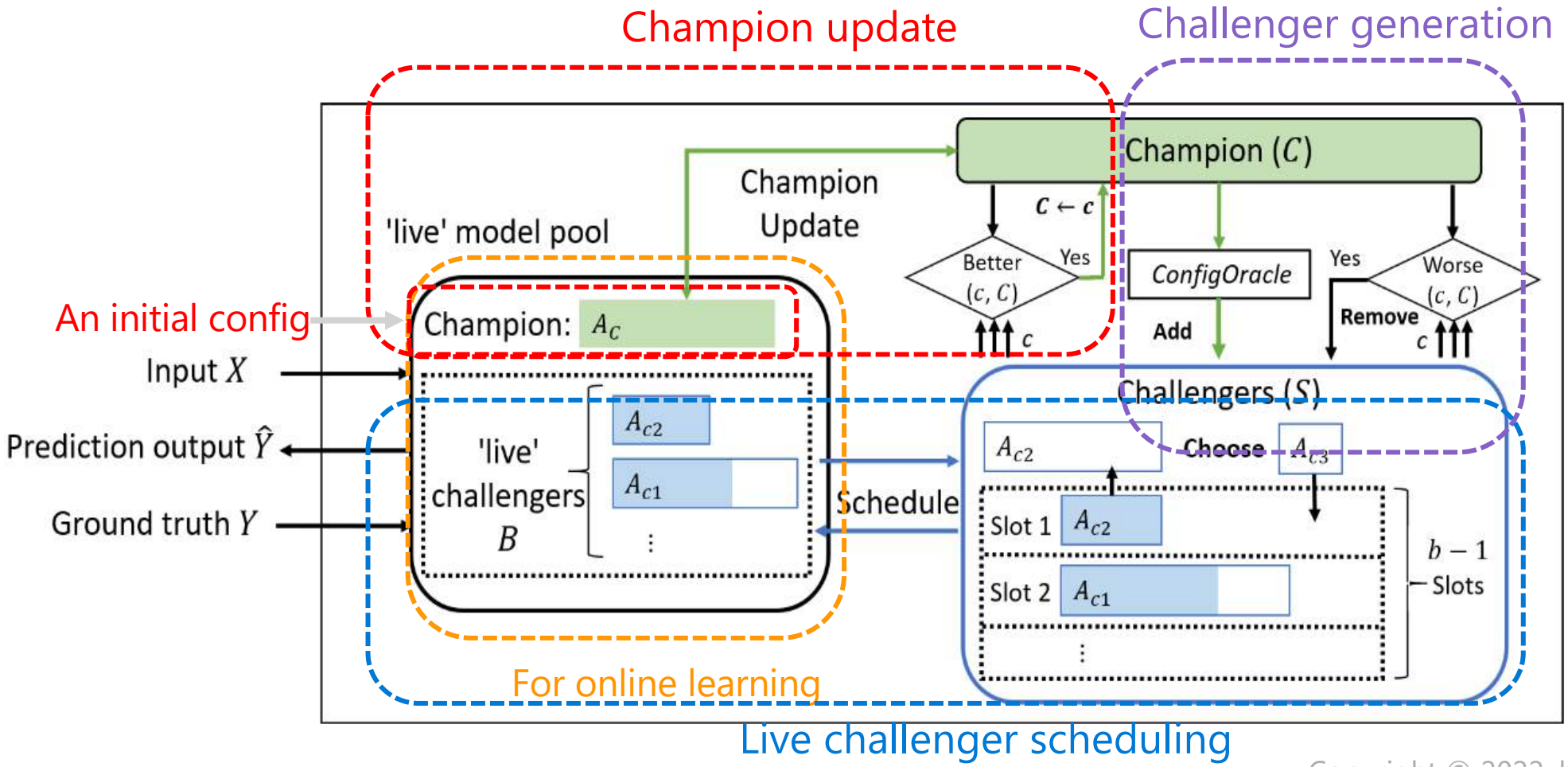
Finding a balance between

- Searching over a large number of plausible choices
- Concentrating the limited computation budget on a few promising choices such that we do not pay a high `learning price' (regret)

Champion-Challengers (ChaCha) for Online AutoML [ICML'21]

Champion: proven best config at the concerned time point

Challengers: the rest of the candidate configs under consideration



Resources to Be Used

First half of the tutorial:

- Overview of AutoML and FLAML
- Task-oriented AutoML with FLAML
  - Notebook: A classification task with AutoML
  - Notebook: A regression task with AuotML using LightGBM as the learner
- ML.Net demo
- Tune user defined functions with FLAML
  - Notebook: Basic tuning procedures and advanced tuning options
  - Notebook: Tune pytorch
- Q & A

Second half of the tutorial:

- Zero-shot AutoML
  - Notebook: Zeroshot AutoML
- Time series forecasting
  - Notebook: AutoML for Time Series Forecast tasks
- Natural language processing
  - Notebook: AutoML for NLP tasks
- Online AutoML
  - Notebook: Online AutoML with Vowpal Wabbit
- Fair AutoML
- Challenges and open problems

https://github.com/microsoft/FLAML/tree/tutorial/tutorial

Demo: AutoVW

+

AutoVW:

```
''' import AutoVW class from flaml package '''
from flaml import AutoVW

'''create an AutoVW instance for tuning namespace interactions'''
# configure both hyperparamters to tune, e.g., 'interactions', and fixed arguments about the online learner,
# e.g., 'quiet' in the search space argument.
autovw_ni = AutoVW(max_live_model_num=5, search_space={'interactions': AutoVW.AUTOMATIC, 'quiet': ''})

# online learning with AutoVW
loss_list_autovw_ni = online_learning_loop(max_iter_num, vw_examples, autovw_ni)
print('Final progressive validation loss of autovw:', sum(loss_list_autovw_ni)/len(loss_list_autovw_ni))
```

Vanilla VW:

```
from vowpalwabbit import pyvw
''' create a vanilla vw instance '''
vanilla_vw = pyvw.vw('--quiet')

# online learning with vanilla VW
loss_list_vanilla = online_learning_loop(max_iter_num, vw_examples, vanilla_vw)
print('Final progressive validation loss of vanilla vw:', sum(loss_list_vanilla)/len(loss_list_vanilla))
```

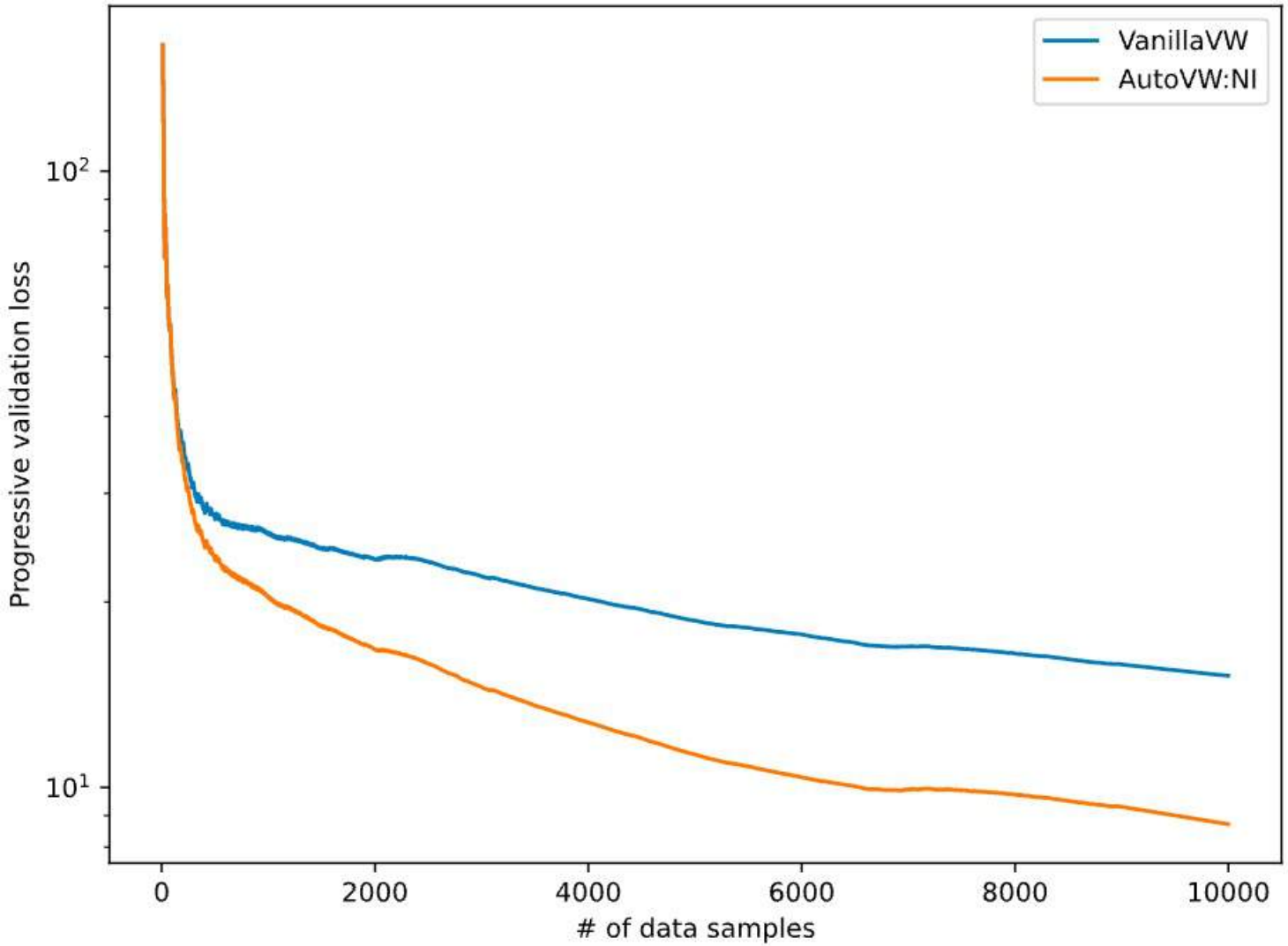
Demo: AutoVW

+


```
Seed namespaces (singletons and interactions): ['g', 'a', 'h', 'b', 'c', 'i', 'd', 'e', 'f']
Created challengers from champion ||
New challenger size 37, ['|ah|', '|eg|', '|gi|', '|ag|', '|de|', '|ei|', '|eh|', '|fg|', '|cf|', '|hi|', '|bf|',
'|cd|', '|ai|', '|ef|', '|cg|', '|ch|', '|ad|', '|bc|', '|gh|', '|bh|', '|ci|', '|fh|', '|bg|', '|be|', '|bd|',
'|fi|', '|bi|', '|df|', '|ac|', '|ae|', '|dg|', '|af|', '|di|', '|ce|', '|dh|', '|ab|', '||']
Online learning for 10000 steps...
Seed namespaces (singletons and interactions): ['ce', 'g', 'a', 'h', 'b', 'c', 'i', 'd', 'e', 'f']
Created challengers from champion |ce|
New challenger size 43, ['|be_ce|', '|bce_ce|', '|ce_ei|', '|ce_ceg|', '|ce_fh|', '|ce_gh|', '|ce_cef|', '|cd_ce|',
'|ce_cg|', '|cde_ce|', '|ce_cf|', '|bd_ce|', '|ae_ce|', '|ce_gi|', '|ce_ci|', '|ab_ce|', '|ce_fg|', '|ce_di|',
'|bi_ce|', '|ce_de|', '|ce_eg|', '|ce_dg|', '|ce_hi|', '|ai_ce|', '|ag_ce|', '|ac_ce|', '|bh_ce|', '|ce_ch|',
'|ce|', '|ace_ce|', '|ah_ce|', '|af_ce|', '|bc_ce|', '|ce_dh|', '|ce_ef|', '|ad_ce|', '|ce_df|', '|ce_cei|',
'|ce_eh|', '|bg_ce|', '|ce_ceh|', '|bf_ce|', '|ce_fi|']
Final progressive validation loss of autovw: 8.7188174219445 29
```

Demo: AutoVW

FLAML + VOWPAL WABBIT



ML Fairness and Fair AutoML

- Fairness in ML

Regulated domains

- **Credit** (Equal Credit Opportunity Act)
- **Education** (Civil Rights Act of 1964; Education Amendments of 1972)
- **Employment** (Civil Rights Act of 1964)
- **Housing** (Fair Housing Act)
- **'Public Accommodation'** (Civil Rights Act of 1964)

Legally recognized 'protected classes'

Race (Civil Rights Act of 1964); **Color** (Civil Rights Act of 1964); **Sex** (Equal Pay Act of 1963; Civil Rights Act of 1964); **Religion** (Civil Rights Act of 1964); **National origin** (Civil Rights Act of 1964); **Citizenship** (Immigration Reform and Control Act); **Age** (Age Discrimination in Employment Act of 1967); **Pregnancy** (Pregnancy Discrimination Act); **Familial status** (Civil Rights Act of 1968); **Disability status** (Rehabilitation Act of 1973; Americans with Disabilities Act of 1990); **Veteran status** (Vietnam Era Veterans' Readjustment Assistance Act of 1974; Uniformed Services Employment and Reemployment Rights Act); **Genetic information** (Genetic Information Nondiscrimination Act)

Existing and ongoing efforts:
✓Fairness definitions in ML
✓Unfairness mitigation methods

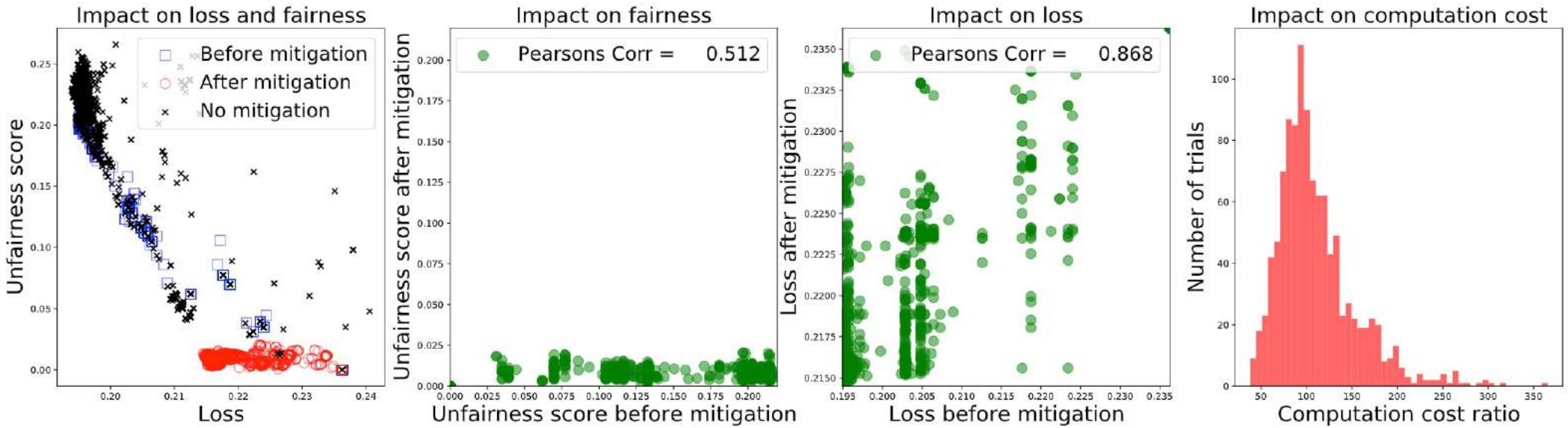
Source: https://fairmlbook.org/tutorial1.html

ML Fairness and Fair AutoML

- Fair AutoML: To find models not only with good accuracy but also **fair.**
  - Is it possible to find a model that is fair without applying additional unfairness mitigation?
  - If unfairness mitigation is necessary, how to incorporate it into an AutoML pipeline?
  - If we can find fair models (w/ or w/o unfairness mitigation), how the “utility” (e.g., accuracy, cost) will be affected?

Fair AutoML

- Developed abstractions for fairness assessment and unfairness mitigation techniques in AutoML.
- Investigated the need and impact of unfairness mitigation on AutoML.



[preprint] Fair AutoML. Qingyun Wu, Chi Wang, arXiv preprint arXiv:2111.06495 (2022).

Fair AutoML

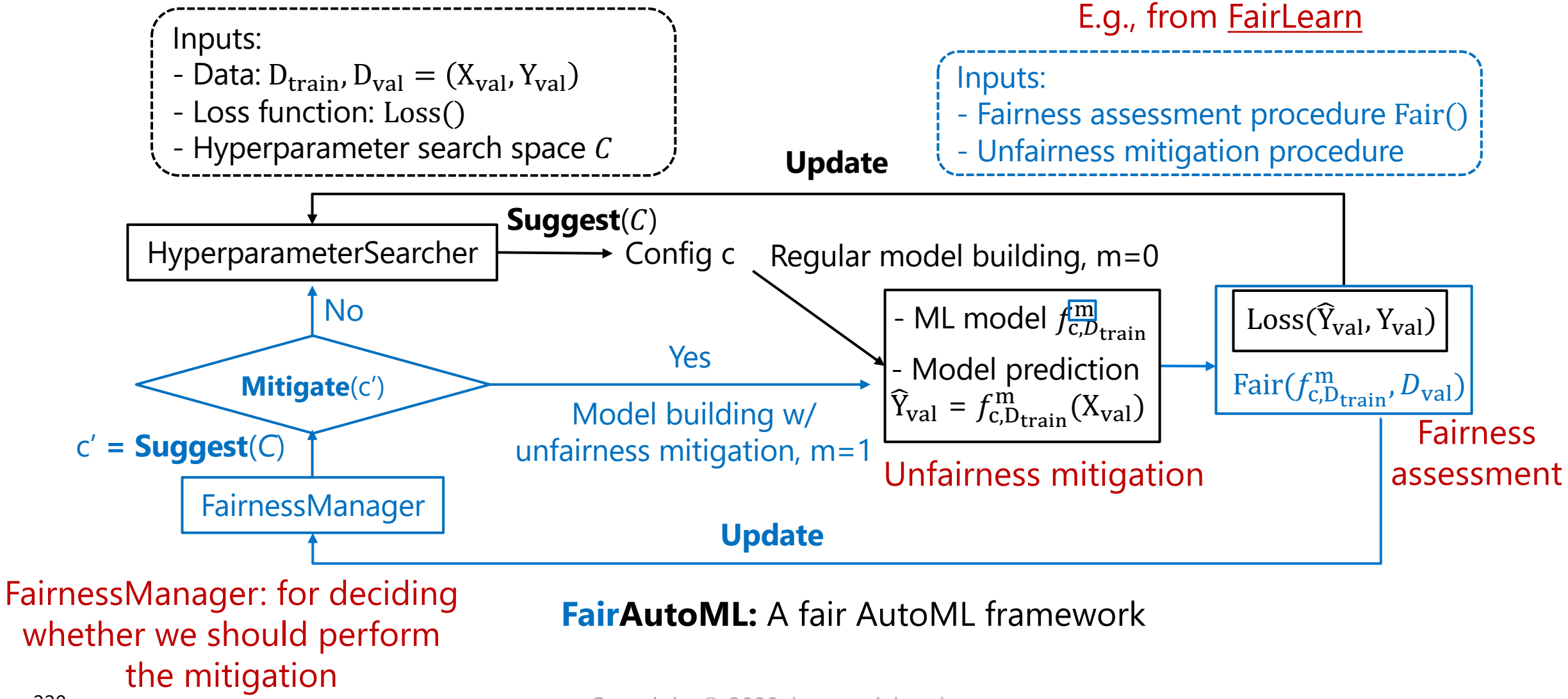
- The fair AutoML problem

$$\min_{c \in \mathcal{C}, m \in \{0,1\}} \texttt{Loss}(f^m_{c, D_{\text{train}}}(\mathbf{X}_{\text{val}}), \mathbf{Y}_{\text{val}}), \quad s.t. \ \texttt{Fair}(f^m_{c, D_{\text{train}}}, D_{\text{val}})$$

Unfairness mitigation

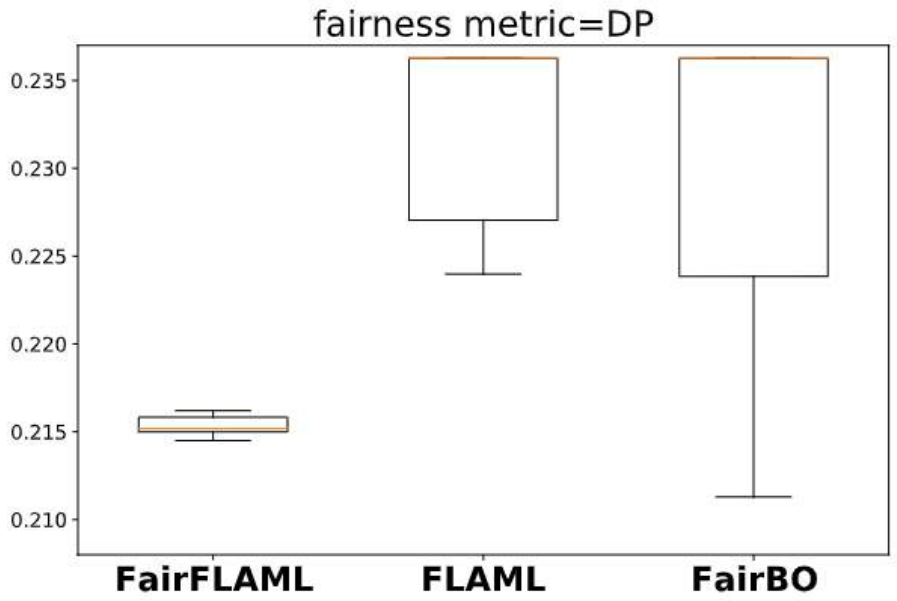
Fairness assessment

Fair AutoML



FairFLAML

- A self-adaptive strategy (via FairnessManager) to balance HPO and unfairness mitigation with a goal of finding a model with the best fair loss efficiently and effectively.



Fair loss on the adult dataset (sensitive attribute = “sex”)

Future Work

Open Problems and Research Opportunities on AutoML

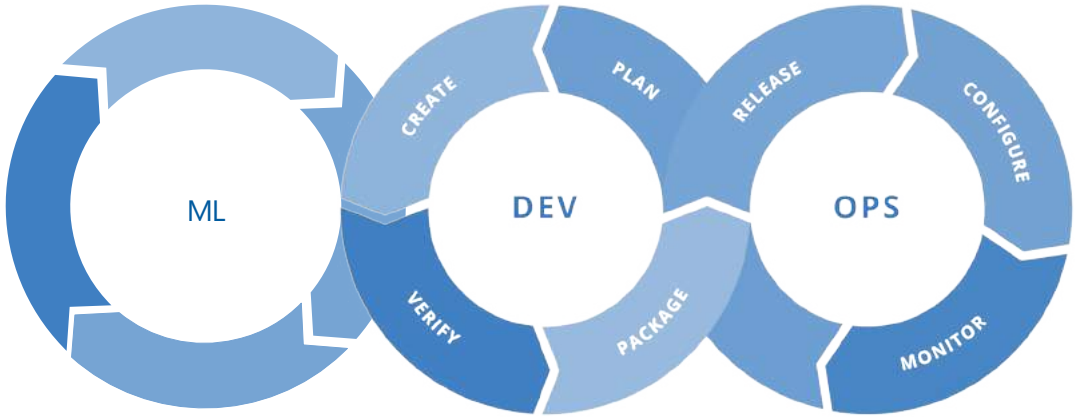
- Deficiencies of AutoML
  - **Causes system failures due to compute intensive workloads**
  - **Lacks customizability**
  - Lacks comprehensive end-to-end support
  - Lacks transparency and interpretability

- Timeout
- Elasticity
- Multi-output tasks
- Early stopping of trials

- Search space suggestion
- Overfitting
- Multiple optimization metrics
- Customize meta learning

Open Problems and Research Opportunities on AutoML

- Deficiencies of AutoML
  - Causes system failures due to compute intensive workloads
  - Lacks customizability
  - **Lacks comprehensive end-to-end support**
  - **Lacks transparency and interpretability**



- Deployment constraints
- Online learning
- Guardrail

- How to decide time budget
- Trustworthiness and Ethics
- How many data samples are needed
- Reproducibility

Use FLAML to Improve/Inspire Your Research

✓ ML model tuning

- Tuning reinforcement learning models
- Tuning graph neural networks

…

✓ Tuning certain choices in your task towards a particular objective

- Synthetic dataset generation by tuning data generation related choices
- Finding the most profitable investment strategies
- RL policy tuning

…

✓ Compare different methods/models more comprehensively with sufficient tuning of each method/model

Call for Contribution

This project welcomes and encourages all forms of contribution, including but not limited to:

- ✓ Pushing patches.
- ✓ Code review of pull requests.
- ✓ Documentation, examples and test cases.
- ✓ Community participation in issues, discussions, and gitter.
- ✓ Tutorials, blog posts, talks that promote the project.
- ✓ Sharing application scenarios and/or related research.

Call For Contribution: Opportunities

✓Share your use cases and needs

- Connect via gitter: https://gitter.im/FLAMLer/community
- Create issues

✓ Development

- Improve existing features: zero-shot AutoML, time series forecasting, online AutoML
- Develop new features: computer vision tasks, multi-modal model, fair AutoML, quality monitoring and drift detection, visualization and explanation

✓ Integration with other libraries

Call for Contribution

Gitter:
https://gitter.im/FLAMLer/community

Contributing guide:
https://microsoft.github.io/FLAML/docs/Contribute

Roadmap:
https://github.com/microsoft/FLAML/wiki/Roadmap-for-Upcoming-Features

Q&A

aka.ms/FLAML

References

- *[MLSys'21]* FLAML: A Fast and Lightweight AutoML Library. Chi Wang, Qingyun Wu, Markus Weimer, Erkang Zhu.
- *[ICML'21]* ChaCha for Online AutoML. Qingyun Wu, Chi Wang, John Langford, Paul Mineiro, Marco Rossi.
- *[ICLR'21]* Economical Hyperparameter Optimization With Blended Search Strategy. Chi Wang, Qingyun Wu, Silu Huang, Amin Saied.
- *[AAAI'21]* Frugal Optimization for Cost-related Hyperparameters. Qingyun Wu, Chi Wang, Silu Huang.
- *[ACL'21]* An Empirical Study on Hyperparameter Optimization for Fine-Tuning Pre-trained Language Models. Susan Liu, Chi Wang.
- *[KDD-AutoML'22]* Mining Robust Default Configurations for Resource-constrained AutoML. Moe Kayali, Chi Wang.
- *[preprint] Fair AutoML. Qingyun Wu, Chi Wang. arXiv preprint arXiv:2111.06495 (2022).*

Thanks to
all contributors
& collaborators!