Lecture 9: Hyperparameter Optimization

Assoc Prof Tham Chen Khong
Dept of Electrical & Computer Engineering (ECE)
NUS
E-mail: eletck@nus.edu.sg

Hyperparameters

- Hyperparameters are those parameters of a model that are not updated during the learning procedure of a model. It can be considered as the 'configuration' of a model.
- Hyperparameters can be parameters that specifically concern the model itself, but may also concern its objective function that is either to be minimized or maximized.
- For example: a neural network can be configured using a hyperparameter concerning the number of layers of neurons, while a hyperparameter concerning the objective function could be the 'learning rate' or 'step size' with which the function is to be optimized.
- The domain from which several configurations of hyperparameter values are to be sampled is called the *search space, configuration space, sampling domain,* or simply *hyperparameter space*. This search space can become increasingly complex, and can be comprised of continuous, categorical, and conditional hyperparameters.
- To add a layer of complexity: the optimization algorithm of the model itself can be a hyperparameter, e.g. SGD, Adam etc.
- This causes some hyperparameters (often just their values) to be dependent on a value that is selected for another hyperparameter. These are often called *conditional hyperparameters* creating a conditional or *tree-like* hyperparameter space.
- There are several hyperparameter optimization (HPO) frameworks implemented in Python that support methods that are able to handle nested search spaces.

Some Hyperparameter Optimization methods

- Random Search
- Grid Search
- Sequential model-based optimization
- Bayesian Optimization
- Gaussian Processes
- Tree-Parzen Estimators
- Scikit-learn
- Scikit-Optimize
- Hyperopt
- Optuna

Hyperparameter Optimization (HPO)

- The aim of hyperparameter optimization in machine learning is to find the hyperparameters of a given machine learning algorithm that returns the best performance as measured on a validation set.

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

- *f*(*x*) represents an objective score to minimize - such as MSE or error rate - evaluated on the validation set; *x** is the set of hyperparameters that yields the lowest value of the score, and *x* can take on any value in the domain *X*.
- In simple terms, we want to find the model hyperparameters that yield the best score on the validation set metric.

Hyperparameter Optimization (HPO)

- The evaluation of $f(x)$ can be **very computationally expensive** and take a lot of time.
  - Each time we try different hyperparameters, we have to train a model on the training data, make predictions on the validation data, and then calculate the validation metric (i.e. train-predict-evaluate cycle).
- Besides the time spent on estimation, one also has to think about what hyperparameters (and their values) might be useful to search over.
- The time needed to optimize increases exponentially as new hyperparameters or values are added to the search space (*curse of dimensionality*).
- Furthermore, HPO is often performed in conjunction with cross-validation (e.g. *k*-fold CV) to ensure that the optimal set of hyperparameters is robust in terms of generalization to new data, which requires even more iterations.
- Three categories of HPO:
  - Exhaustive Search, e.g. Random or Grid search
  - Sequential model-based optimization
  - Other / mixed approaches (such as evolutionary algorithms)

Random Search: randomly select sets of hyperparameter values

Grid Search

- Grid Search is often the go-to method for HPO. (`GridSearchCV` in scikit-learn)
- The idea is quite simple: Define a set of hyperparameters and their values, train a model for *each* possible combination, and subsequently select the combination with the best (cross-validated) score/performance.
- All hyperparameter combinations can be laid out in a Cartesian grid.
- However, this approach suffers from the *curse of dimensionality*, meaning that for each dimension/hyperparameter that is added to the search space - or for each value that is added to a hyperparameter - the number of evaluations that have to be performed increases exponentially.
- Therefore, this approach can become increasingly computationally expensive, especially for models that have many relevant hyperparameters that influence its performance.

RS & GS shortcoming and SMBO

- Both Random Search and Grid Search have some limitations, the most prominent one being that both methods waste time looking for hyperparameter configurations in search space areas that, in reality, are not that promising.
- This is because no information about the quality of each search iteration is communicated from one search iteration to the next.
- They simply try out combinations of hyperparameter values, not tracking in which direction of the search space the **loss** might be gradually minimized, nor being able to adjust their behaviour based on this information. This is often called an *uninformed search*.
- **Sequential model-based optimization (SMBO)** methods address this shortcoming to a certain extent. They keep score of promising areas in the search space because they build up a history of configuration settings and their scores, adjusting their behaviour at each iteration (an *informed search*).

Sequential Model-Based Optimization (SMBO): Bayesian Optimization

- SMBO is a group of methods that fall under the *Bayesian Optimization* paradigm
  - Main idea: Build a probability model of the objective function and use it to select the most promising hyperparameters to evaluate in the true objective function
- These methods use a **surrogate model** (probabilistic model) and an **acquisition function** that work together to find the best model by iteratively selecting the most promising hyperparameters in the **search space** to approximate the actual **objective function**.

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

x is (set of) hyperparameters, *f*(*x*) is objective function
H is observation history

S is acquisition function (e.g. EI) for selecting *x*
M_t is probabilistic model (e.g. GP, TPE) of objective function *f*(*x*)
- acts as surrogate to *f*(*x*)

i.e. training and validation

from
Bergstra, James S., et al. "Algorithms for hyper-parameter optimization." *Advances in Neural Information Processing Systems*, 2011.

Acquisition function

- **Acquisition function**: in the choice of *x*, balance between exploring new areas in objective space and exploiting areas that are already known to have favourable values
  - a good acquisition function is Expected Improvement (EI), but there are others
- **Expected Improvement (EI)**: non-negative expected improvement over the best previously observed objective value
  - define $y=f(x)$ (in practice, it is a metric, e.g. loss)
  - EI = integral of $(f_{best}-y)$ over $p(y|x,H)$ obtained from probabilistic model M_t
  - The aim is to **maximize the Expected Improvement** with respect to the hyperparameters *x*.

Sequential Model-Based Optimization (SMBO)

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

i.e. training and validation

x is (set of) hyperparameters, *f*(*x*) is objective function
H is observation history

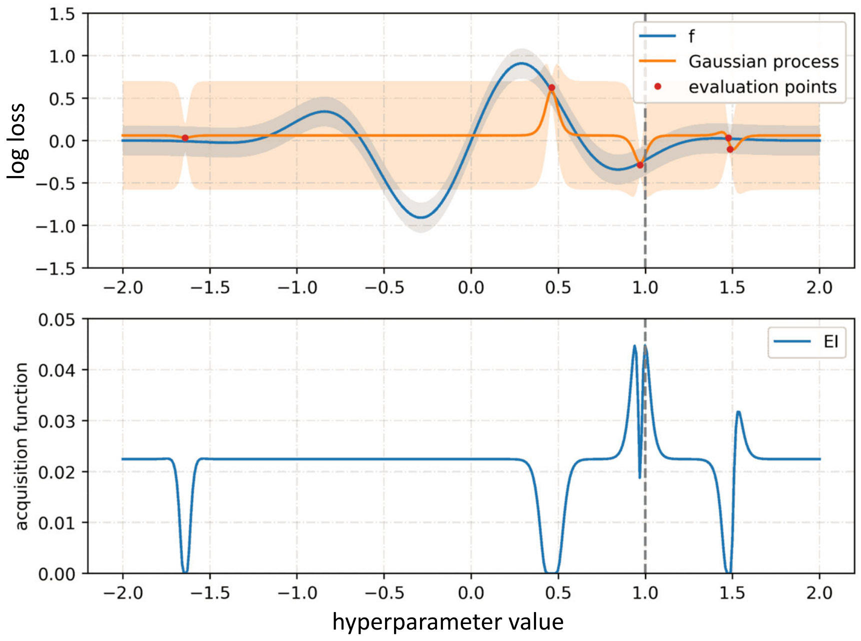
S is acquisition function (e.g. EI) for selecting *x*
M_t is probabilistic model (e.g. GP, TPE) of objective function *f*(*x*) - acts as surrogate to *f*(*x*)

- At each iteration, the selected hyperparameters from the search space are evaluated w.r.t. the objective function.
- The objective function takes in a hyperparameter configuration and outputs a scalar metric which is to be minimized (or maximized).
- The surrogate model/function is a probabilistic model of which there are several types:
  - **Gaussian Processes** and **Tree-Parzen Estimators** are often used

1. Gaussian Processes (GPs) - 1

- Gaussian Processes (GPs) are often the default when considering Bayesian Optimization.
- A GP specifies a **probability distribution over a set of functions**, characterised by its mean function and covariance function.
- They are able to approximate complex functions, and are non-parametric in nature.
- Here, we use a GP to model the loss *y* given hyperparameter configuration *x*, i.e. it estimates *p*(*y*|*x*).
- Use Bayesian inference to improve estimated mean and covariance functions based on observations, i.e. to get posterior distribution, when the actual objective function is evaluated.
- The acquisition function, e.g. Expected Improvement (EI), will support the GP by reducing uncertainty through iteratively selecting training values in areas that are likely to minimize the GP function.
- At each iteration, a value is sampled where the acquisition function is **maximized**.

1. Gaussian Processes (GP) - 2



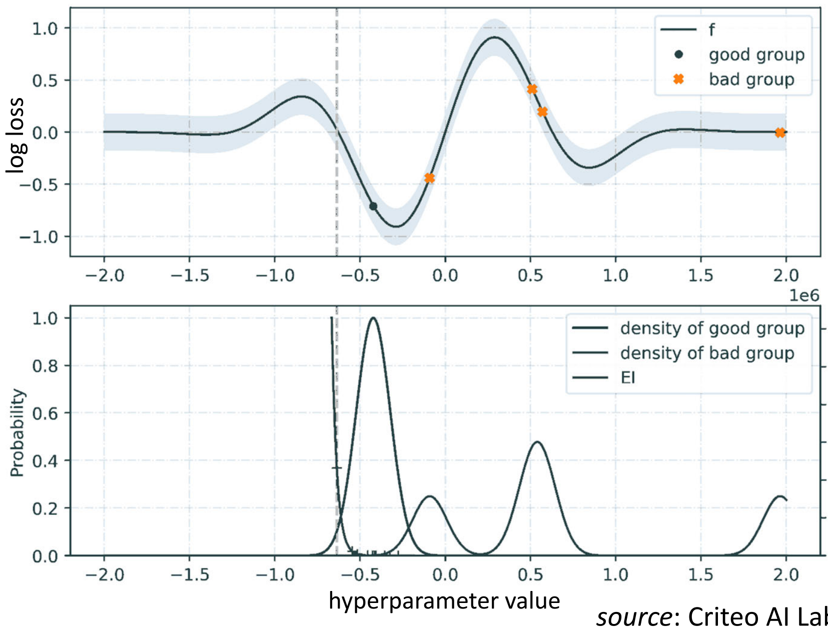
The blue line *f* represents the unknown objective function, which is iteratively approximated by the surrogate model* (GP) represented by the orange line (+ confidence intervals).
***note that this is updated**
As the number of iterations increases, the better the model is able to determine where *f* is minimized. The acquisition function (EI) indicates where in the search space the surrogate model should orientate itself next where the **Expected Improvement is maximized**: i.e. **low surrogate model value with broad confidence interval** (orange gap)
→ efficient search in hyperparameter space

source: Criteo AI Lab

2. Tree-Parzen Estimators (TPE) - 1

- GPs model the loss *y* given hyperparameter configurations in *x* directly, resulting in *p*(*y*|*x*).
- TPE works somewhat differently:
- For a deep NN, the hyperparameters are continuous or categorical, and frequently conditional on one another. **A conditional non-continuous hyperparameter configuration space would be tree-like.**
- To get *p*(*y*|*x*), TPE first determines *p*(*y*) and *p*(*x*|*y*) and uses Bayes rule.
- TPE first uses random samples in *x* to get an initial set of loss scores. Subsequently, it singles out the best loss scores (the 'good' group) based on some threshold *y** for the loss, and appoints the remaining ones to the 'remainder group'.
- By creating these two groups, two (non-parametric) densities can be constructed to model *p*(*x*|*y*). Let *l*(*x*) be the density of the 'good' group and *g*(*x*) be the density of the 'remainder group' a.k.a. 'bad group'.
  l(*x*) is modelled by $p(x|y < y^*)$ and *g*(*x*) is modelled by $p(x|y \geq y^*)$ using *Parzen estimators* (kernel density estimators).
- Expected Improvement (EI) is proportional to ratio of *l*(*x*) over *g*(*x*).
- With each iteration of the TPE process, a sampled hyperparameter configuration is evaluated and the densities are adjusted to approximate the true loss function.

2. Tree-Parzen Estimators (TPE) - 2



source: Criteo AI Lab

As with GPs, in each iteration, the acquisition function (EI) determines which values are chosen next. Intuitively, one would like to choose a hyperparameter configuration from *l*(*x*), since this is the 'good' group that yields the lowest loss. EI does this automatically for us. It is maximized where the ratio of *l*(*x*)/*g*(*x*) is maximized. This means that the hyperparameter configuration that is chosen for the next iteration is the one for which the probability of it belonging to the 'good' group divided by the probability of it belonging to the 'remainder group' is highest (see the bottom part of the plot).
After each iteration, the values that were already used can be reassigned to either the 'good' or 'remainder group (see top part of the plot).

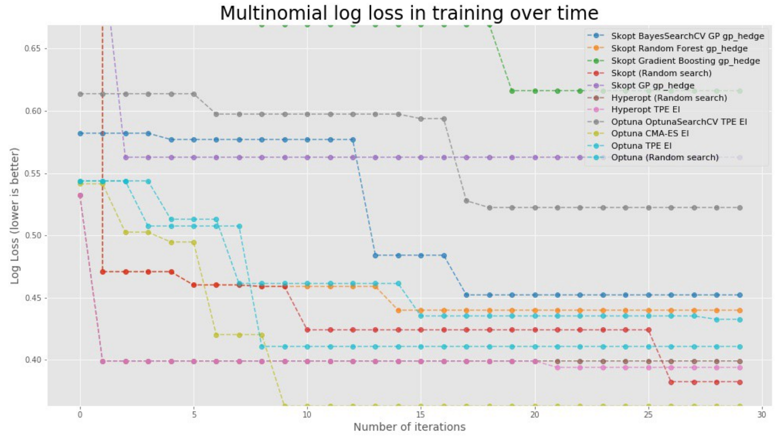
Hyperparameter Optimization Software

- Which method should be used when optimizing hyperparameters in Python?
- Several frameworks (scikit-learn, scikit-optimize, Hyperopt, Optuna) implement both Exhaustive Search methods as well as SMBO.
- TPE with EI in Hyperopt and Optuna generally perform well
  - https://github.com/hyperopt/hyperopt
  - https://optuna.org/

Example with
OttoGroup dataset
and LightGBM model

Notes:
Optuna-CMA-ES: evolutionary
Hyperopt TPE EI
Random does quite well

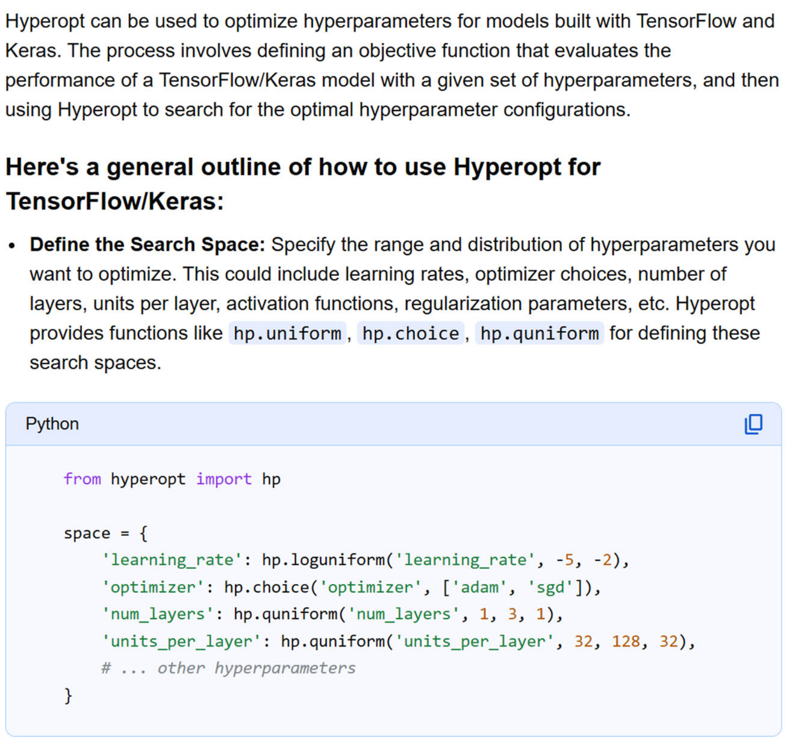
Multinomial log loss in training over time



Summary

- Bayesian model-based optimization methods build a probability model of the objective function to propose smarter choices for the next set of hyperparameters to evaluate.
- SMBO is a formalization of Bayesian optimization which is more efficient at finding the best hyperparameters for a machine learning model than random or grid search.
- Sequential model-based optimization methods differ in how they build the surrogate model, but they all rely on information from previous trials to propose better hyperparameters for the next evaluation.

Hyperopt for TensorFlow/Keras - 1



Hyperopt can be used to optimize hyperparameters for models built with TensorFlow and Keras. The process involves defining an objective function that evaluates the performance of a TensorFlow/Keras model with a given set of hyperparameters, and then using Hyperopt to search for the optimal hyperparameter configurations.

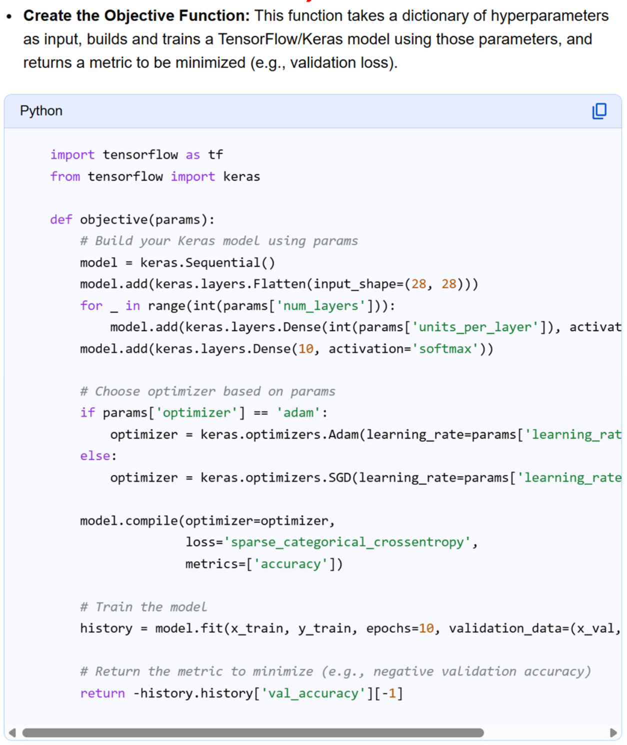
Here's a general outline of how to use Hyperopt for TensorFlow/Keras:

- **Define the Search Space:** Specify the range and distribution of hyperparameters you want to optimize. This could include learning rates, optimizer choices, number of layers, units per layer, activation functions, regularization parameters, etc. Hyperopt provides functions like `hp.uniform`, `hp.choice`, `hp.quniform` for defining these search spaces.

Python

```python
from hyperopt import hp

space = {
    'learning_rate': hp.loguniform('learning_rate', -5, -2),
    'optimizer': hp.choice('optimizer', ['adam', 'sgd']),
    'num_layers': hp.quniform('num_layers', 1, 3, 1),
    'units_per_layer': hp.quniform('units_per_layer', 32, 128, 32),
    # ... other hyperparameters
}
```

Hyperopt for TensorFlow/Keras - 2

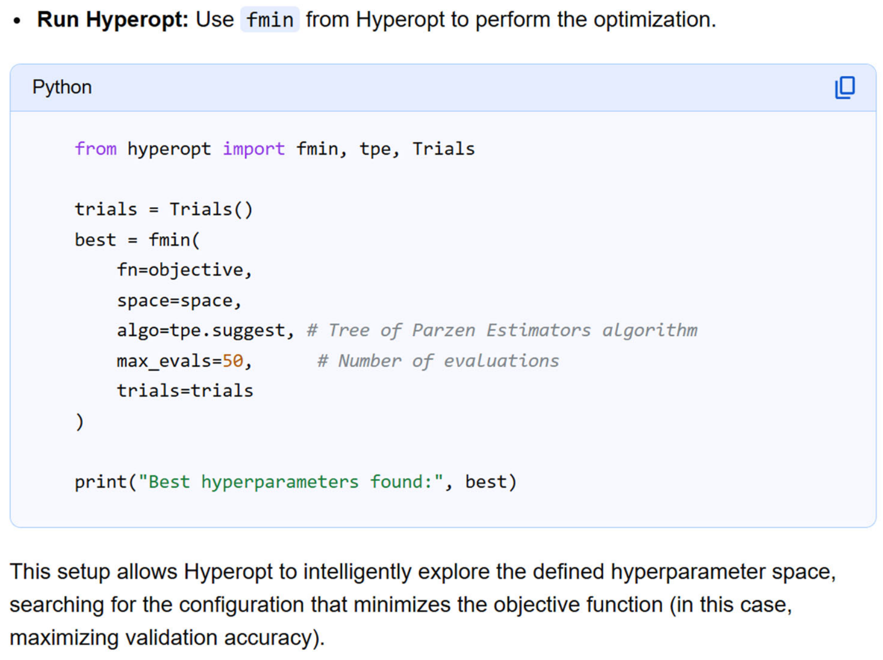


- **Create the Objective Function:** This function takes a dictionary of hyperparameters as input, builds and trains a TensorFlow/Keras model using those parameters, and returns a metric to be minimized (e.g., validation loss).

Python

```python
import tensorflow as tf
from tensorflow import keras

def objective(params):
    # Build your Keras model using params
    model = keras.Sequential()
    model.add(keras.layers.Flatten(input_shape=(28, 28)))
    for _ in range(int(params['num_layers'])):
        model.add(keras.layers.Dense(int(params['units_per_layer']), activat
    model.add(keras.layers.Dense(10, activation='softmax'))

    # Choose optimizer based on params
    if params['optimizer'] == 'adam':
        optimizer = keras.optimizers.Adam(learning_rate=params['learning_rat
    else:
        optimizer = keras.optimizers.SGD(learning_rate=params['learning_rate

    model.compile(optimizer=optimizer,
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])

    # Train the model
    history = model.fit(x_train, y_train, epochs=10, validation_data=(x_val,

    # Return the metric to minimize (e.g., negative validation accuracy)
    return -history.history['val_accuracy'][-1]
```

Hyperopt for TensorFlow/Keras - 3



- **Run Hyperopt:** Use `fmin` from Hyperopt to perform the optimization.

Python

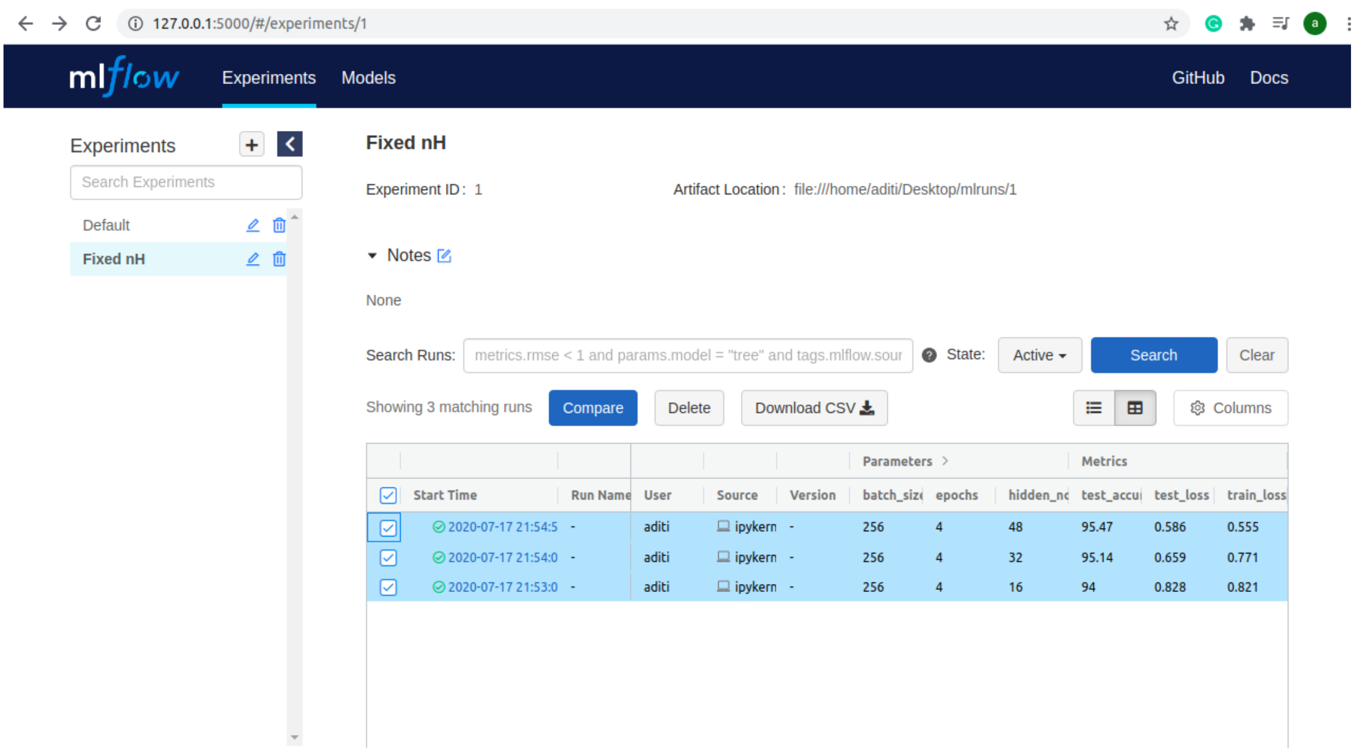
```python
from hyperopt import fmin, tpe, Trials

trials = Trials()
best = fmin(
    fn=objective,
    space=space,
    algo=tpe.suggest, # Tree of Parzen Estimators algorithm
    max_evals=50,      # Number of evaluations
    trials=trials
)

print("Best hyperparameters found:", best)
```

This setup allows Hyperopt to intelligently explore the defined hyperparameter space, searching for the configuration that minimizes the objective function (in this case, maximizing validation accuracy).

Example: MLflow(Tracking)+hyperparam opt/tuning - 1

- MLflow Tracking is organized around the concept of runs (mlruns). Each run records some information like what is explained as below.
- Create an experiment id using mlflow.create_experiment() creates a new experiment and returns its ID. Runs can be launched under the experiment by passing the experiment ID to mlflow.start_run() which returns current active run if there is any, else it will create new active run and return its object.
- Log the value of the metric train loss, test loss, and test accuracy using mlflow.log_metric() which logs into a single key-value metric.
- At each run, log parameters like start time, batch size, epochs, learning rate, momentum, hidden nodes and test loss along with its source code using mlflow.log_param() which logs them into a single key-value param in the currently active run.
- Select the different models for comparison by clicking on the compare button.

- For further details, see https://medium.com/swlh/hyperparameter-tuning-with-mlflow-tracking-b67ec4de18c9



Fixed nH

Experiment ID: 1 Artifact Location: file:///home/aditi/Desktop/mlruns/1

Notes

None

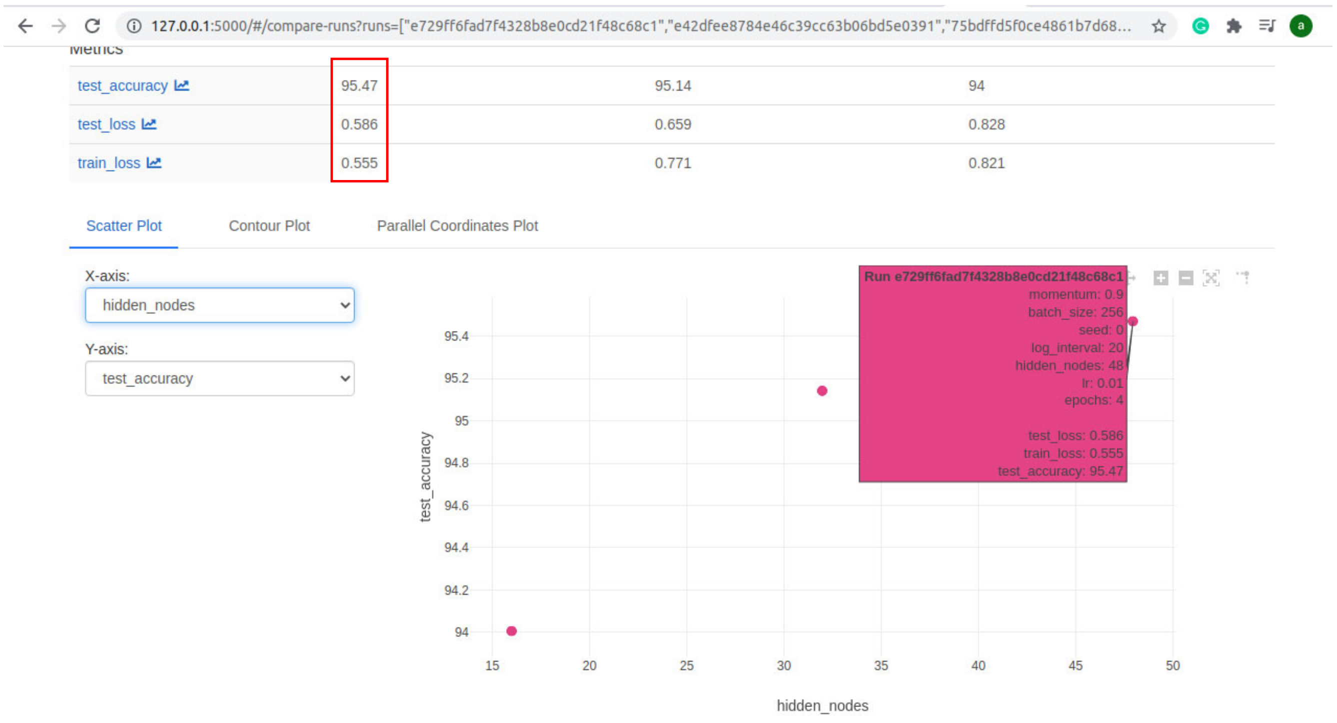
Search Runs: metrics.rmse < 1 and params.model = "tree" and tags.mlflow.sour State: Active

Showing 3 matching runs

						Parameters			Metrics		
✓	Start Time	Run Name	User	Source	Version	batch_size	epochs	hidden_nc	test_accu	test_loss	train_loss
✓	2020-07-17 21:54:5	-	aditi	ipykern	-	256	4	48	95.47	0.586	0.555
✓	2020-07-17 21:54:0	-	aditi	ipykern	-	256	4	32	95.14	0.659	0.771
✓	2020-07-17 21:53:0	-	aditi	ipykern	-	256	4	16	94	0.828	0.821

Example: MLflow(Tracking)+hyperparam opt/tuning - 2

- Scatter plot: select the x-axis and y-axis parameters to compare the selected model and check the performance of the model.
- Check the performance of the individual model and check different metrics plots (for example, to track how the model's loss function is converging).
- Record images (for example, confusion matrix for each run), models (for example, a Keras model), or even different data files as artifacts using mlflow.log_artifact().
- Log all runs for each hyperparameter setting, and each of those runs includes the hyperparameter setting and the evaluation metric. Comparing these runs in the MLflow UI helps with visualizing the effect of tuning each hyperparameter.
- Once a better configuration of hyperparameter is discovered, load that model by specifying the run id and use for inference.
- Conclusion:
- Using MLflow, we can easily track and manage the different configured trained models and compare them easily to find the best set of hyperparameters.


127.0.0.1:5000/#/compare-runs?runs=["e729ff6fad7f4328b8e0cd21f48c68c1","e42dfee8784e46c39cc63b06bd5e0391","75bdffd5f0ce4861b7d68...

Metrics

test_accuracy	95.47	95.14	94
test_loss	0.586	0.659	0.828
train_loss	0.555	0.771	0.821

X-axis: hidden_nodes

Y-axis: test_accuracy

Run e729ff6fad7f4328b8e0cd21f48c68c1
momentum: 0.9
batch_size: 256
seed: 0
log_interval: 20
hidden_nodes: 48
lr: 0.01
epochs: 4

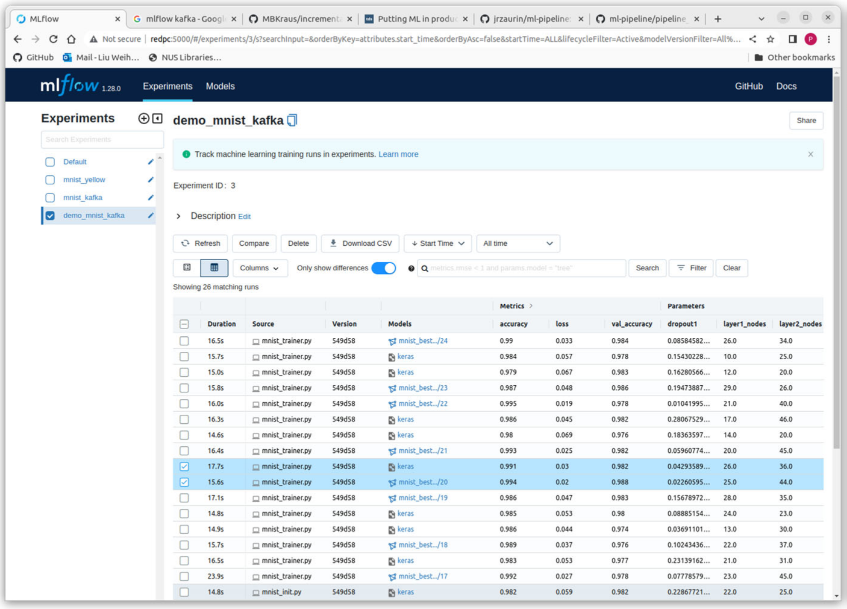
test_loss: 0.586
train_loss: 0.555
test_accuracy: 95.47

Example: Prediction and Training with Kafka+MLflow+Hyperopt

https://github.com/hyperopt/hyperopt

- IoT sensors send information using Kafka to *predictor* which uses initial ML model (trained with historical data) to obtain predictions
- *Trainer* produces new ML model using appended newly arriving data
  - incrementally updating the model with new data can improve the model, while also reducing model drift
  - do hyperparameter optimization, e.g. using Hyperopt (TPE), to determine good configurations and obtain new improved ML model
  - convey new improved ML model to *predictor*
- Repeat the process

Example: Prediction and Training with Kafka+MLflow+Hyperopt



redpc:5000/#/experiments/3/s?searchInput=&orderByKey=attributes.start_time&orderByAsc=false&startTime=ALL&lifecycleFilter=Active&modelVersionFilter=All%...

Experiments: Default, mnist_yellow, mnist_kafka, [✓ demo_mnist_kafka]

demo_mnist_kafka

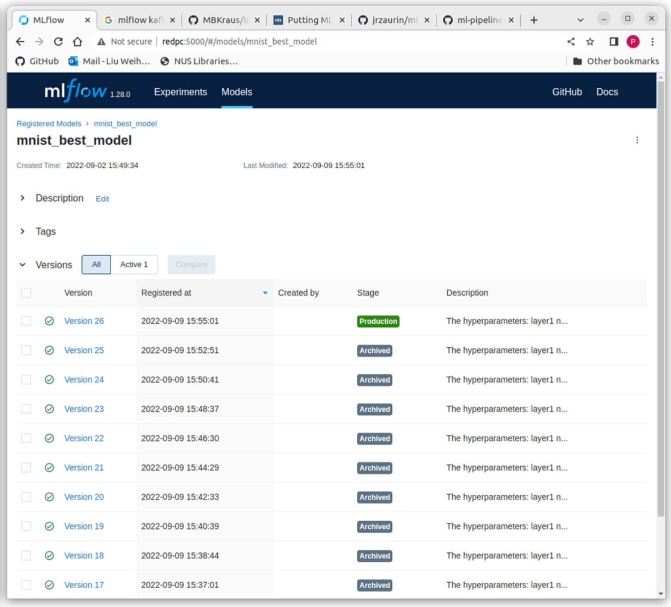
Track machine learning training runs in experiments. Learn more

Experiment ID: 3

Description Edit

Showing 26 matching runs

				Metrics			Parameters		
Duration	Source	Version	Models	accuracy	loss	val_accuracy	dropout1	layer1_nodes	layer2_nodes
16.5s	mnist_trainer.py	549d58	mnist_best.../24	0.99	0.033	0.984	0.08584582...	26.0	34.0
15.7s	mnist_trainer.py	549d58	keras	0.984	0.057	0.978	0.15430228...	10.0	25.0
15.0s	mnist_trainer.py	549d58	keras	0.979	0.067	0.983	0.16280566...	12.0	20.0
15.8s	mnist_trainer.py	549d58	mnist_best.../23	0.987	0.048	0.986	0.19473887...	29.0	26.0
16.0s	mnist_trainer.py	549d58	mnist_best.../22	0.995	0.019	0.978	0.01041995...	21.0	40.0
16.3s	mnist_trainer.py	549d58	keras	0.986	0.045	0.982	0.28067529...	17.0	46.0
14.6s	mnist_trainer.py	549d58	keras	0.98	0.069	0.976	0.18363597...	14.0	20.0
16.4s	mnist_trainer.py	549d58	mnist_best.../21	0.993	0.025	0.982	0.05960774...	20.0	45.0
17.7s	mnist_trainer.py	549d58	keras	0.991	0.03	0.982	0.04293589...	26.0	36.0
15.6s	mnist_trainer.py	549d58	mnist_best.../20	0.994	0.02	0.988	0.02260595...	25.0	44.0
17.1s	mnist_trainer.py	549d58	mnist_best.../19	0.986	0.047	0.983	0.15678972...	28.0	35.0
14.8s	mnist_trainer.py	549d58	keras	0.985	0.053	0.98	0.08885154...	24.0	23.0
14.9s	mnist_trainer.py	549d58	keras	0.986	0.044	0.974	0.03691101...	13.0	30.0
15.7s	mnist_trainer.py	549d58	mnist_best.../18	0.989	0.037	0.976	0.10243436...	22.0	37.0
16.5s	mnist_trainer.py	549d58	keras	0.983	0.053	0.977	0.23139162...	21.0	31.0
23.9s	mnist_trainer.py	549d58	mnist_best.../17	0.992	0.027	0.978	0.07778579...	23.0	45.0
14.8s	mnist_init.py	549d58	keras	0.982	0.059	0.982	0.22867721...	22.0	25.0



redpc:5000/#/models/mnist_best_model

Registered Models › mnist_best_model

mnist_best_model

Created Time: 2022-09-02 15:49:34 Last Modified: 2022-09-09 15:55:01

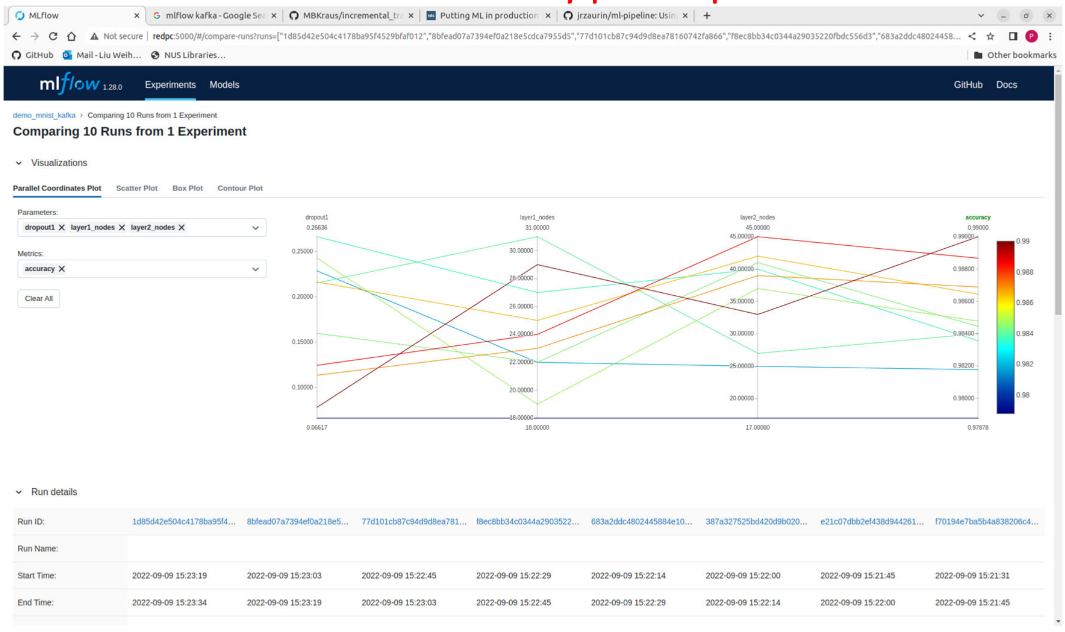
Description Edit

Tags

Versions All Active 1 Compare

Version	Registered at	Created by	Stage	Description
Version 26	2022-09-09 15:55:01		Production	The hyperparameters: layer1 n...
Version 25	2022-09-09 15:52:51		Archived	The hyperparameters: layer1 n...
Version 24	2022-09-09 15:50:41		Archived	The hyperparameters: layer1 n...
Version 23	2022-09-09 15:48:37		Archived	The hyperparameters: layer1 n...
Version 22	2022-09-09 15:46:30		Archived	The hyperparameters: layer1 n...
Version 21	2022-09-09 15:44:29		Archived	The hyperparameters: layer1 n...
Version 20	2022-09-09 15:42:33		Archived	The hyperparameters: layer1 n...
Version 19	2022-09-09 15:40:39		Archived	The hyperparameters: layer1 n...
Version 18	2022-09-09 15:38:44		Archived	The hyperparameters: layer1 n...
Version 17	2022-09-09 15:37:01		Archived	The hyperparameters: layer1 n...

Example: Prediction and Training with Kafka+MLflow+Hyperopt



redpc:5000/#/compare-runs?runs=["1d85d42e504c4178ba95f4529bfaf012","8bfead07a7394ef0a218e5cdca7955d5","77d101cb87c94d9d8ea78160742fa866","f8ec8bb34c0344a29035220fbdc556d3","683a2ddc48024458...

demo_mnist_kafka › Comparing 10 Runs from 1 Experiment

Comparing 10 Runs from 1 Experiment

Visualizations

Parallel Coordinates Plot | Scatter Plot | Box Plot | Contour Plot

Parameters: dropout1, layer1_nodes, layer2_nodes

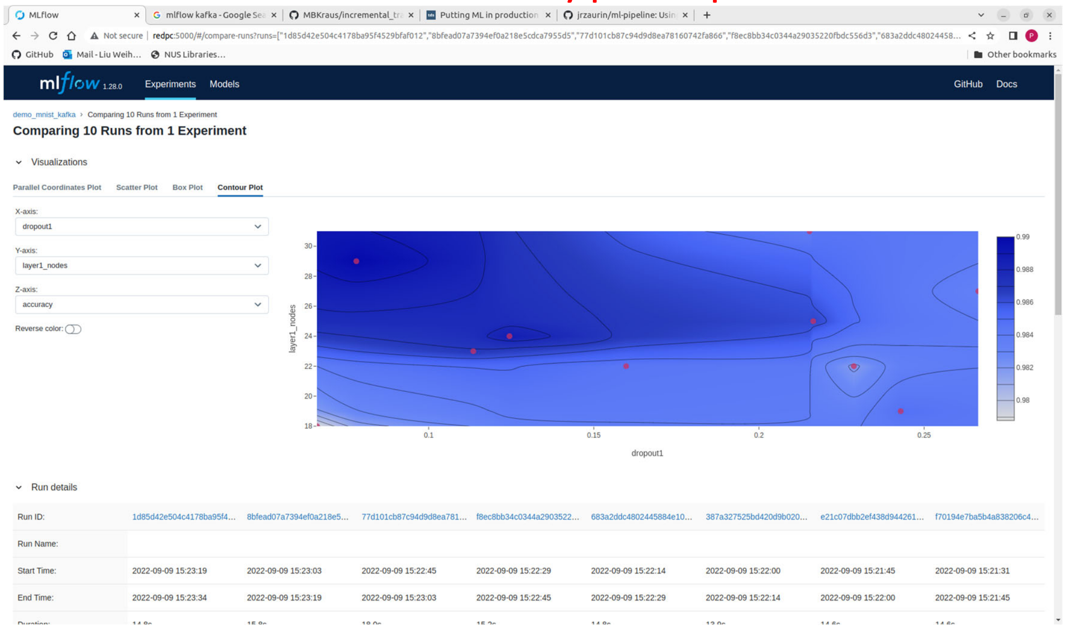
Metrics: accuracy

Clear All

Run details

Run ID:	1d85d42e504c4178ba95f4...	8bfead07a7394ef0a218e5...	77d101cb87c94d9d8ea781...	f8ec8bb34c0344a2903522...	683a2ddc4802445884e10...	387a327525bd420d9b020...	e21c07dbb2ef438d944261...	f70194e7ba5b4a838206c4...
Run Name:								
Start Time:	2022-09-09 15:23:19	2022-09-09 15:23:03	2022-09-09 15:22:45	2022-09-09 15:22:29	2022-09-09 15:22:14	2022-09-09 15:22:00	2022-09-09 15:21:45	2022-09-09 15:21:31
End Time:	2022-09-09 15:23:34	2022-09-09 15:23:19	2022-09-09 15:23:03	2022-09-09 15:22:45	2022-09-09 15:22:29	2022-09-09 15:22:14	2022-09-09 15:22:00	2022-09-09 15:21:45

Example: Prediction and Training with Kafka+MLflow+Hyperopt



redpc:5000/#/compare-runs?runs=["1d85d42e504c4178ba95f4529bfaf012","8bfead07a7394ef0a218e5cdca7955d5","77d101cb87c94d9d8ea78160742fa866","f8ec8bb34c0344a29035220fbdc556d3","683a2ddc48024458...

demo_mnist_kafka › Comparing 10 Runs from 1 Experiment

Comparing 10 Runs from 1 Experiment

X-axis: dropout1

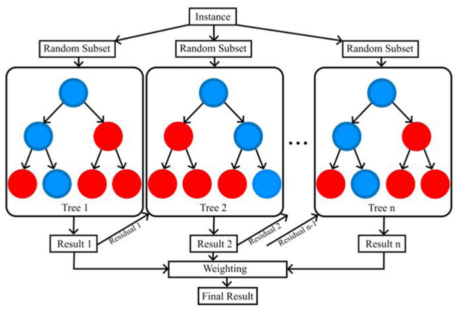
Y-axis: layer1_nodes

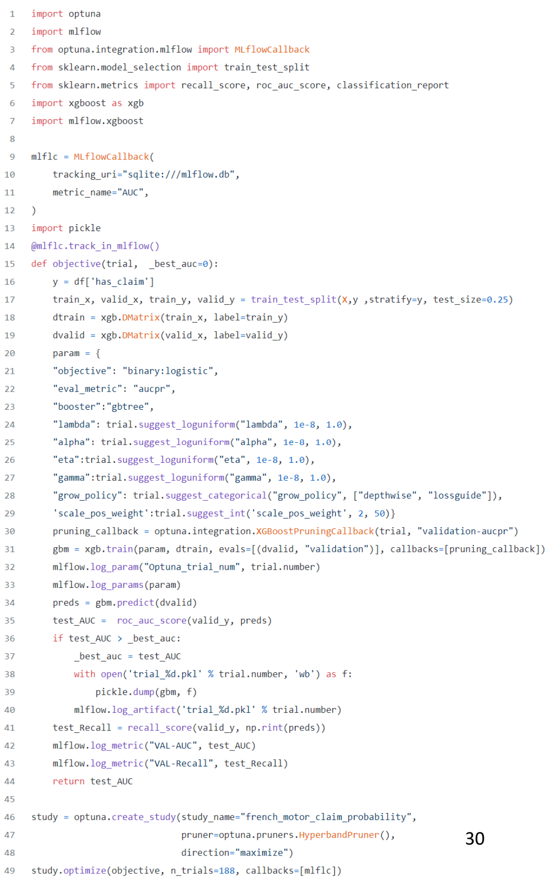
Z-axis: accuracy

Run ID:	1d85d42e504c4178ba95f4...	8bfead07a7394ef0a218e5...	77d101cb87c94d9d8ea781...	f8ec8bb34c0344a2903522...	683a2ddc4802445884e10...	387a327525bd420d9b020...	e21c07dbb2ef438d944261...	f70194e7ba5b4a838206c4...
Run Name:								
Start Time:	2022-09-09 15:23:19	2022-09-09 15:23:03	2022-09-09 15:22:45	2022-09-09 15:22:29	2022-09-09 15:22:14	2022-09-09 15:22:00	2022-09-09 15:21:45	2022-09-09 15:21:31
End Time:	2022-09-09 15:23:34	2022-09-09 15:23:19	2022-09-09 15:23:03	2022-09-09 15:22:45	2022-09-09 15:22:29	2022-09-09 15:22:14	2022-09-09 15:22:00	2022-09-09 15:21:45

MLflow and Optuna (HPO)

- For XGBoost model





```python
import optuna
import mlflow
from optuna.integration.mlflow import MLflowCallback
from sklearn.model_selection import train_test_split
from sklearn.metrics import recall_score, roc_auc_score, classification_report
import xgboost as xgb
import mlflow.xgboost

mlflc = MLflowCallback(
    tracking_uri="sqlite:///mlflow.db",
    metric_name="AUC",
)
import pickle
@mlflc.track_in_mlflow()
def objective(trial,  _best_auc=0):
    y = df['has_claim']
    train_x, valid_x, train_y, valid_y = train_test_split(X,y ,stratify=y, test_size=0.25)
    dtrain = xgb.DMatrix(train_x, label=train_y)
    dvalid = xgb.DMatrix(valid_x, label=valid_y)
    param = {
    "objective": "binary:logistic",
    "eval_metric": "aucpr",
    "booster":"gbtree",
    "lambda": trial.suggest_loguniform("lambda", 1e-8, 1.0),
    "alpha": trial.suggest_loguniform("alpha", 1e-8, 1.0),
    "eta":trial.suggest_loguniform("eta", 1e-8, 1.0),
    "gamma":trial.suggest_loguniform("gamma", 1e-8, 1.0),
    "grow_policy": trial.suggest_categorical("grow_policy", ["depthwise", "lossguide"]),
    'scale_pos_weight':trial.suggest_int('scale_pos_weight', 2, 50)}
    pruning_callback = optuna.integration.XGBoostPruningCallback(trial, "validation-aucpr")
    gbm = xgb.train(param, dtrain, evals=[(dvalid, "validation")], callbacks=[pruning_callback])
    mlflow.log_param("Optuna_trial_num", trial.number)
    mlflow.log_params(param)
    preds = gbm.predict(dvalid)
    test_AUC =  roc_auc_score(valid_y, preds)
    if test_AUC > _best_auc:
        _best_auc = test_AUC
        with open('trial_%d.pkl' % trial.number, 'wb') as f:
            pickle.dump(gbm, f)
        mlflow.log_artifact('trial_%d.pkl' % trial.number)
    test_Recall = recall_score(valid_y, np.rint(preds))
    mlflow.log_metric("VAL-AUC", test_AUC)
    mlflow.log_metric("VAL-Recall", test_Recall)
    return test_AUC

study = optuna.create_study(study_name="french_motor_claim_probability",
                            pruner=optuna.pruners.HyperbandPruner(),
                            direction="maximize")
study.optimize(objective, n_trials=188, callbacks=[mlflc])
```

Thank You

Questions?

Assoc Prof Tham Chen Khong

E-mail: eletck@nus.edu.sg