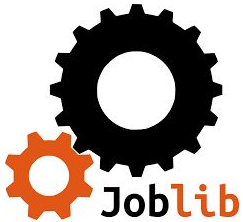
Scaling Hyperparameter Tuning with Dask-ML on Clusters

Jinhui Qin
SHARCNET | Compute Ontario | Compute Canada
Western University

Scikit-learn

- An open source python package for machine learning
- Build on *Numpy*, *SciPy*, and *matplotlib*
- Commonly for predictive data analysis
- Parallel backend support via *Joblib* on a single machine
- No GPU support



What is hyperparameter tuning?

An example with Scikit-learn

- Choosing a model
- Training the model
- Evaluating the model
- Improving the model

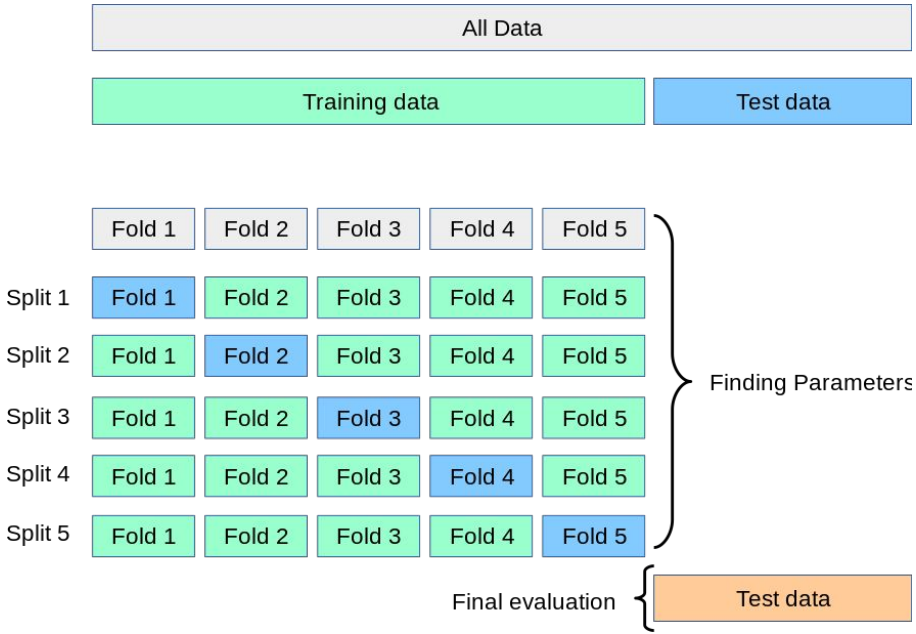
Hyperparameters

- Chosen before training
- Tuning with two generic approaches
  - GridSearchCV(...)
  - RandomizedSearchCV(...)

```
from sklearn.datasets import make_classification
X, y = make_classification(n_samples=10000, n_features=4, n_classes=2)

from sklearn.svm import SVC
estimator = SVC(C=1.0, kernel='rbf', gamma='auto', random_state=0)
estimator.fit(X, y)
estimator.score(X, y)

from sklearn.model_selection import GridSearchCV

estimator = SVC(gamma='auto', random_state=0)

param_grid = {
    'C': [0.001, 0.1, 10.0],
    'kernel': ['rbf', 'poly']
}
grid_search = GridSearchCV(estimator, param_grid, cv=5)
grid_search.fit(X, y)
print(grid_search.best_params_, grid_search.best_score_)
```

Understand the workload

5-fold Cross-Validation (CV)




```
param_grid = {
    'C': [0.001, 0.1, 10.0],
    'kernel': ['rbf', 'poly']
}
Grid_search = GridSearchCV(estimator, param_grid, cv=5)
grid_search.fit(X, y)
```

Total number of fittings:

3 x 2 x 5 + 1 = 31

Ref: Scikit-learn User Guide

Challenges in scaling hyperparameter tuning

- Memory constrained
  - E.g. dataset size is too large to fit in memory
- Compute constrained
  - E.g. search space is large and it takes too long to complete
- Parallel considerations
  - E.g. *Scikit-learn* with a parallel-backend via *Joblib*
    - Dataset fits into the memory
    - Many cores are available on *a single machine*

Scikit-learn in parallel via Joblib

GridSearchCV(*estimator*, *param_grid*, ..., *n_jobs*, ..., *cv*, .., *pre_dispatch*, ...)

- n_jobs
  - Controls the number of jobs to run in parallel
  - Default parallel backend: spawning *processes*
  - Dataset is copied for each fitting task
- pre_dispatch
  - Controls the number of tasks that get dispatched during parallel execution
  - Default is 2*n_jobs

Avoid oversubscription

- A common problem: spawning too many threads
  - e.g. a nested routine has been implemented with OpenMP
  - e.g. optimized math routines from MKL, OpenBLAS, or BLIS

- Avoid using significantly more processes or threads than the number of CPUs available on a machine
  - Joblib with default backend, max_threads = n_cpus // n_jobs
  - Run a test job and monitor the resource usage
    - ssh to the compute node, run command `top` or `htop`, e.g. top -H -u $USER or htop -u $USER
  - Manually setting environment variables
    - `OMP_NUM_THREADS`, `MKL_NUM_THREADS`, `OPENBLAS_NUM_THREADS`, or `BLIS_NUM_THREADS`

General suggestions

- When dataset can fit into the memory of a single node
  - Request a full node, e.g. --nodes=1, --ntasks-per-node=32, --mem=125GB on Graham
  - Parallelized execution with Scikit-learn
  - Avoid oversubscription when running in parallel

Graham			Cedar			Beluga		
Number of nodes	Cores per node	Memory per node	Number of nodes	Cores per node	Memory per node	Number of nodes	Cores per node	Memory per node
903	32	125 GB	1408	48	187 GB	589	40	186 GB
56	32	250 GB	576	32	125 GB	160	40	92 GB
24	32	500 GB	96	32	250 GB	53	40	752 GB

General suggestions

- When the resource available on a full node is not enough
  - Consider a distributed approach with Dask-ML
- Dask-ML
  - Dask provides a distributed framework crossing multiple nodes on a cluster
  - Dask-ML extends the parallelism of Scikit-learn to a cluster
    - Works well for *medium-sized* datasets but *large computations*
    - pip install --no-index dask-ml
      - Dependencies, e.g. numpy, scipy, pandas, scikit-learn, joblib, etc. are included.
  - Dask-ML also provides tools for memory constrained cases, e.g.
    - dask_ml.model_selection.IncrementalSearchCV(…)
    - only for an estimator in Scikit-learn that supports partial_fit

Dask-ML

Deploy a dask distributed network

- One centralized dask scheduler
- One or multiple dask workers registered with the *scheduler*

```
#!/bin/sh

#SBATCH --account=def-user
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=8
#SBATCH --mem=16G
#SBATCH --time=DD-HH:MM

module load mpi4py
source /path/to/dask-ml-Env/bin/activate

dask-scheduler --scheduler-file /path/to/dask_scheduler.json \
--no-dashboard --interface ib0
```

```
#!/bin/sh

#SBATCH --account=def-user
#SBATCH --nodes=1
#SBATCH --cpus-per-task=16
#SBATCH --mem=64G
#SBATCH --time=DD-HH:MM

module load mpi4py
source /path/to/dask-ml-Env/bin/activate

dask-worker --scheduler-file /path/to/dask_scheduler.json \
--no-nanny --local-directory $SLURM_TMPDIR --interface ib0
```

Dask-ML

Distributed hyperparameter tuning

- Connect and submit the workload to the dask scheduler via a client
- Use parallelized *Scikit-learn* via *Joblib* with a dask backend

```
#!/bin/sh

#SBATCH --account=def-user
#SBATCH --cpus-per-task=2
#SBATCH --mem=4G
#SBATCH --time=DD-HH:MM

module load mpi4py
source /path/to/dask-ml-Env/bin/activate

python dask-ml-test.py
```

```
from sklearn.datasets import make_classification
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV
import joblib
from dask.distributed import Client

client = Client(scheduler_file="/path/to/dask_scheduler.json")

X, y = make_classification(n_samples=10000, n_features=4, n_classes=2)

estimator = SVC(gamma='auto', random_state=0)
param_grid = {
    'C': [0.001, 0.1, 1.0, 2.5, 5, 10.0],
    'kernel': ['rbf', 'poly', 'linear'],
    'shrinking': [True, False],
}
grid_search = GridSearchCV(estimator, param_grid, cv=5, n_job=-1)

with joblib.parallel_backend("dask", scatter=[X, y]):
      grid_search.fit(X, y)

print("Best params:", grid_search.best_params_)
print( "Best score: ", grid_search.best_score_ )
```

Demo on Graham