[2. Kubeflow Components]

2.0 서론

kubeflow = **ML workflow에 필요한 component들로 이루어진 tool**

- ex) Jupyter Notebook server : for **모델 개발**
- ex) Katib : for **하이퍼파라미터 최적화**
- ex) Pipeline : for **workflow 구축**

(이들 중 몇몇은 python SDK를 가짐)

2.1 Dashboard

2.1.1 개요

- 각 component의 UI를 접근할 수 있는 gateway
- 이를 통해 kubeflow의 **컴포넌트들의 endpoint에 접근** 가능
- kubernetes의 컴포넌트들은 **Istio**를 통해 사용자 접근을 허용
  - 0.6 버전 이전 : Ambassador라는 API 게이트웨이
  - 0.6 버전 이후 : **Istio라는 서비스 메쉬 오픈소스**
- 컴포넌트 예시
  - Jupyter Notebook Server
  - Katib
  - Pipeline
  - Artifact Store
  - Manage Contributors (현재 사용자가 현재 namespace의 소유자일 경우에만!)

2.1.2 로컬에서 Dashboard 접속

- `kubectl`을 사용해 **포트포워딩**으로 **로컬호스트 주소**로 대쉬보드에 접근 가능
- 준비 과정
  - step 1) (`kubectl` 설치 과정은 생략)
  - step 2) **로컬호스트에서 kubernetes의 cluster의 API 서버에 접속할 수 있도록 설정**
    - 필요한 정보 : API 서버에 접근 가능한 **"인증서 정보"**

      (in `$HOME/.kube/config` 에 저장)

      (위 config 파일의 구조는 "kubernetes의 리소스 템플릿 형태")

```
apiVersion : v1
kind : Config
clusters :
...
contexts :
...
users :
...
```

- kubeflow 설치 후, 프로파일 생성되면, **프로파일명과 같은 namespace**가 생성됨

  해당 namespace안에는, **3개의 service account**가 생성됨

  - 1) default
  - 2) default-viewer
  - **3) default-editor (이것을 사용할 것)**

`default-editor`

```
$ kubectl get sa default-editor -n seunghan96 -o yaml

apiVersion : v1
kind : ServiceAccount
metadata:
...
secrets:
 - name : name=default-editor-token-dhlmq
```

→ 여기에 있는 **"secret" 정보**를 이용하여, "로컬호스트에서 **kubectl을 통해 kubeflow가 설치된 kubernetes cluster에 접근**"

step 1) kubnernetes cluster에서, **SA의 "certificate-authority-data" 값을 가져옴**

```
$ name=default-editor-token-dhlmq
$ ca=$(kubectl get secret/$name -o jsonpath='{.data.ca\.crt}' -n seunghan96)
```

step 2) SA의 시크릿을 **base 64 디코딩하여 토큰으로 변환**

```
$ token=$(kubectl get secret/$name -o jsonpath='{.data.ca\.crt}' -n seunghan96 |
base64 --decode)
```

step 3) 생성된 값으로 **config파일 생성** (`sa.kubeconfig`)

```
echo "
apiVersion : v1
kind : Config
clusters :
```

```
 - name : kubeflow-cluster
   cluster :
     certificate-authority-data : ${ca}
     server : ${server}
context :
 - name : kubeflow-cluster
   context :
     cluster : kubeflow-cluster
     namespace : seunghan96
     user : default-editor
current-context : kubeflow-cluster
users:
 - name : default-editor
   user :
     token : ${token}
" > sa.kubeconfig
```

step 4) 생성된 `sa.kubeconfig` 파일을, **로컬호스트의 `${HOME}/.kube/config`로 저장**

step 5) `kubectl`을 사용하여 **포트 포워딩**

```
$ export NAMESPACE=istio-system
$ kubectl port-forward -n istio-system svc/istio-ingressgateway 8080:80
```

→ 에러 뜰 것! 권한 X

step 6) `default-editor` SA에 관련 롤을 바인딩 & 다시 포트포워딩

```
# role-bind-default-editor.yaml

apiVersion : ..
kind : Role
metadata :
...
----

apiVersion : ..
kind : RoleBinding
metadata :
...
subjects :
- kind : ServiceAccount
  name : default-editor
  namespace : seunghan96
...
```

```
$ kubectl apply -f role-bind-default-editor.yaml
$ kubectl port-forward -n istio-system svc/istio-ingressgateway 8080:80
```

step 7) `http://localhost:8080` 으로 대쉬보드 접속!

2.2 Notebook Servers

2.2.1 개요

Notebook Servers = **kubernetes 상의 Jupyter Notebook**

특징

- kubeflow를 설치하고 나서, 별 다른 설정없이 사용가능한 component
- 쿠버네티스에서 resource를 scheduling 해주기 때문에,

  사용자는 notebook 설정만으로도 간단히 할당 받을 수 있음!
- notebook 생성 시, 해당 kubeflow 및 namespace의 리소스를 사용할 수 있는 권한을 바인딩 하기 때문에,

  노트북 상에서도 `kubectl` 을 통해 kubernetes의 리소스를 관리할 수 있음

2.2.2 Notebook 생성

step 1) 대쉬보드에서 NEW SERVER를 클릭

step 2) 노트북 이름 & 자신의 namespace 설정

- 이름 : 영문자&숫자 (O), 공백 (X)
- 이름 형식 : (자동으로) `workspace-{이름}`

step 3) 노트북에 사용할 docker image 선택

- ex) tf 버전, CPU/GPU 등
- kubernetes 노트북에서는 standard image & custom image 지원
  - standard image : TF, kubectl, gcp, kubeflow 라이브러리 등이 포함된 도커 이미지
  - custom image : 사용자가 만든 이미지

step 4) 노트북이 사용할 CPU, MEMORY 입력

- 노트북 상에서는, 사용가능한 현재 리소스 확인 hard
- kubernetes cluster에서 확인해야!

```
$ kubectl get nodes "-o=custom-
columns=NAME:.metadata.name,CPU:.status.allocatable.cpu,MEMORY:.status.allocatab
le.memory"
NAME       CPU    MEMORY
instance-1  4   32835736Ki
instance-2  8   32835000Ki
```

step 5) workpace 볼륨, 데이터 볼륨 입력

step 6) 추가 설정 입력 / 추가 리소스 설정

step 7) 녹색 체크로 바뀌면, 생성 완료! CONNECT 눌러서 접속

2.2.3 Kubernetes 리소스 확인

위에서 생성한 Jupyter Notebook는 **kubernetes의 리소스를 사용**할 수 있음

노트북 terminal창에서, `kubectl get pod`를 실행함으로써,

현재 namespace에 생성되어 있는 pod들 확인 가능! (권한 없는 다른 namespace거는 확인 불가)

사용 가능한 kubernetes의 리소스들

- Pods
- Deployments
- Services
- Jobs
- TFJobs
- PyTorchJobs

2.2.4 Custom Image 생성

(kubeflow에서 제공하는 이미지가 아닌) **사용자의 custom image** 생성 가능!

- 물론, Jupyter Notebook이 실행되는 이미지여야!

Jupyter Notebook 실행 시, 아래의 항목들은 포함해야!

- 작업 디렉토리 설정 :
  - `--notebook-dir=/home/jovyan`
- 주피터 노트북이 모든 IP에 대응할 수 있도록 :
  - `--ip=0.0.0.0`
- Jupyter Notebook을 사용자가 root 권한으로 사용 :
  - `--allow-root`
- 포트 번호 설정 :
  - `--port=8888`
- 인증 해제 :
  - `--NotebookApp.token='' --NotebookApp.password=''`
- allow origin :
  - `--NotebookApp.allow_origin='*'`
- base URL 설정 :
  - `--NotebookApp.base_url=NB_PREFIX`

- kubeflow notebook controller는, NB_PREFIX라는 환경변수로 base URL을 관리!

example)

```
ENV NB_PREFIX /

CMD ["sh", "-c", "jupyter notebook --notebook-dir=/home/jovyan --ip=0.0.0.0
..... "]
```

실습

- Jupyter Lab 이미지를, kubeflow Jupyter Notebook용으로 만들기
- Jupyter Lab의 기본 image dockerfile

```
FROM python:3.6
WORKDIR /jup
RUN pip install jupyter -U && pip install jupyterlab
EXPOSE 8888
ENTRYPOINT ["jupyter", "lab", "--ip=0.0.0.0", "--allow-root"]
```

- kubeflow jupyter notebook이 되기 위한 유틸들 포함

  (출처 : https://github.com/mojokb/handson-kubeflow/blob/master/notebook/Dockerfile)

```
FROM python:3.6
WORKDIR /home/jovyan
USER root
RUN pip install jupyter -U && pip install jupyterlab

RUN apt-get update && apt-get install -yq --no-install-recommends \
  apt-transport-https \
  build-essential \
  bzip2 \
  ca-certificates \
  curl \
  g++ \
  git \
  gnupg \
  graphviz \
  locales \
  lsb-release \
  openssh-client \
  sudo \
  unzip \
  vim \
  wget \
  zip \
  emacs \
  python3-pip \
  python3-dev \
  python3-setuptools \
  && apt-get clean && \
```

```
  rm -rf /var/lib/apt/lists/*

RUN curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo
apt-key add -
RUN echo "deb https://apt.kubernetes.io/ kubernetes-xenial main" | sudo tee
-a /etc/apt/sources.list.d/kubernetes.list
RUN apt-get update
RUN apt-get install -y kubectl

RUN pip install jupyterlab && \
    jupyter serverextension enable --py jupyterlab --sys-prefix

RUN pip install kubernetes kubeflow kfp redis
RUN pip install kubeflow-katib==0.0.2

ARG NB_USER=jovyan

EXPOSE 8888


ENV NB_USER $NB_USER
ENV NB_UID=1000
ENV HOME /home/$NB_USER
ENV NB_PREFIX /

CMD ["sh", "-c", "jupyter lab --notebook-dir=/home/jovyan --ip=0.0.0.0 --no-
browser --allow-root --port=8888 --LabApp.token='' --LabApp.password='' --
LabApp.allow_origin='*' --LabApp.base_url=${NB_PREFIX}"]
```

- 위와 같은 custom iamge를 만든 후,

  notebook에서 이 image를 선택 후 실행을 해본다.

2.2.5 TroubleShooting

- volume이 pending이라, 노트북이 생성되지 않는 상태
- notebook도 하나의 파드이다! log 확인가능

```
$ kubectl logs -f ${노트북명-n} - c {파드명} -n {네임스페이스}
```

- 노트북은 StatefulSet으로 관리되므로, 아래 명령어로도 확인 가능

```
$ kubectl describe statefulsets {노트북명} -n {네임스페이스}
```

2.3 Fairing

2.3.1 소개

Fairing이란?

- ML 모델 학습/배포를 위한 python 패키지
- 몇줄의 코드만으로도 kubeflow의 jupyter notebook에서 ML 모델 생성/학습/배포등의 작업을 kubernetes의 cluster로 요청할 수 있음

Fairing 프로젝트의 목표

- 1) 쉬운 ML 모델 트레이닝 잡 패키징 (작성한 코드 도커화)
- 2) 쉬운 학습 (클라우드 infrastucture에 대한 깊은 지식 필요 X)
- 3) 쉬운 배포

2.3.2 아키텍쳐

- step 1) Jupyter notebook, Python 함수/파일을 **docker image로 빌드함**
- step 2) 빌드된 이미지를 **docker registry로 푸시함**
- step 3) 푸시 이후, 설정된 배포 리소스 타입에 따라 **Job, TFJob, KFServing 등의 리소스로 변환** 하여

  kubernetes API 서버로 요청

→ 이 과정을 페어링 패키지는 크게 **"preprocessor", "builder", "deployer"** 구조로 나눠 실행

1. preprocessor :
   - 작성된 코드를 "docker image에 넣을 수 있도록 패키지화"
2. builder :
   - 패키지된 파일을 "docker image화"
3. deployer :
   - 생성된 이미지를 "kubernetes cluster에 배포"

2.3.3 Fairing 설치

- 요건 : Python 3.6 버전 이상

1. 로컬 개발 환경
   - docker client가 설치되어 있어야

```
$ pip install kubeflow-fairing
$ pip show kubeflow-fairint
```

2. jupyter notebook에서의 설치

- 2-1) 일반적인 jupyter notebook은 위와 동일
- 2-2) kubeflow notebook에서는, 별도의 설치과정 X

2.3.4 Fairing 설정

fairing = docker image & kubernetes 리소스를 사용하는 패키지

→ 따라서, docker image를 저장할 수 있는...

- **"docker registry"** 정보
- **"kubeflow cluster에 접속"**할 수 있는 정보

가 필요하다!

1) docker registry

- 페어링이 생성한 docker image를 registry에 pull/push해야
- 따라서, docker registry는 **공인 인증서를 가지고 있는 서버**여야!

2) kubectl

- kubernetes cluster에 접속하려면, kubeconfig 정보를 사용해야
- 따라서, kubeectl이 설치되어 있어야!

2.3.5 fairing.config

- fairing 패키지의 핵심은 Config
- fairing에 코드를 적용할 때, 코드를 건드리지 않고 `fairing.config`로 시작하는 코드를~

(https://github.com/mojokb/handson-kubeflow/blob/master/fairing/input_files_example/with_input_files.py)

```
import os
import tensorflow as tf
from kubeflow import fairing

DOCKER_REGISTRY = 'kubeflow-registry.default.svc.cluster.local:30000'

# (1)
fairing.config.set_builder('append',
                           base_image='tensorflow/tensorflow:1.14.0-py3',
                           registry=DOCKER_REGISTRY, push=True)
# (2)
fairing.config.set_deployer('job')


def train():
    hostname = tf.constant(os.environ['HOSTNAME'])
    sess = tf.Session()
    print('Hostname: ', sess.run(hostname).decode('utf-8'))
```

```
if __name__ == '__main__':
    # (3)
    remote_train = fairing.config.fn(train)
    remote_train()
```

코드 해석

- 환경변수 "HOSTNAME"를 출력하는 예시

Config 클래스는 fairing 패키지 구조인

- 1) preprocessor
- 2) builder
- 3) deployer

에 대응하는 setter들을 가짐.

기본 값 :

- preprocessor : (노트북 환경) "notebook", (아니면) "python"
- builder : "append"
- deployer : "job"

2.3.6 preprocessor

- 역할 : **docker image로 패키지화**할 대상을 설정
- 4개의 type으로 나뉨
  - 1) python : python 파일을 패키징

    ```
    fairing.config.set_preprocessor('python',
                                    executable=file_name,
                                    input_files=[file_name])
    ```

  - 2) notebook : jupyter notebook 파일을 python 파일로 변환 후 패키징

    ```
    fairing.config.set_preprocessor('notebook',
                                    notebook_file='test_notebook.ipynb')
    ```

  - 3) full_notebook : jupyter notebook 파일을 수행 후, 결과를 notebook 파일로 생성
  - 4) function : 단일 함수를 패키징

    ```
    fairing.config.set_preprocessor('function',
                                     function_obj=train,
                                     input_files=['data.csv',
                                                  'requirements.txt'])
    ```

2.3.7 Builder

- 역할 : preprocessor가 생성한 **패키지를 docker image화**
- 3개의 빌드 type으로 나뉨
  - 1) append : (docker client 사용 X) python library 로 이미지 생성
  - 2) cluster : (구글 컨테이너 툴인) Kaniko로 이미지 생성
  - 3) docker : docker client로 이미지 생성

- 코드 :

  (https://github.com/mojokb/handson-kubeflow/blob/master/fairing/fairing_mnist.tf2.0.cluster.minio.py)

```
# 생략
fairing.config.set_builder(
     'append', #-------------------------------------------------- (1)
     image_name='fairing-job', #---------------------------------- (2)
     base_image='brightfly/kubeflow-jupyter-lab:tf2.0-gpu', #------- (3)
     registry=DOCKER_REGISTRY, #---------------------------------- (4)
     push=True) #------------------------------------------------- (5)
# 생략
```

- 코드 해석 :
  - (1) 빌드 type
  - (2) 생성될 image명
  - (3) base image
  - (4) docker registry
  - (5) registry push 여부

2.3.8 Deployer

docker image 생성이 완료되면, "배포"하기!

```
fairing.config.set_deployer('job', #--------------------- (1)
                            namespace='test', #----------- (2)
                            pod_spec_mutators=[ #---------- (3)
                                k8s_utils.get_resource_mutator(cpu=2, memory=5)
                            ]) #----------- (4)
```

- (1) 배포 형태 (job, tfjob, pytorchjob, serving ...)
- (2) 네임스페이스
- (3) 배포가 될 pod의 spec
- (4) 리소스 limit을 "cpu 2개, memory 5Gi"로

2.3.9 Config.run

- 설정된 값을 기준으로 fairing을 실행
- steps
  - step 1) preprocessor 통해서 builder 생성하고, builder를 실행
  - step 2) builder의 docker image 생성이 완료되면,
    해당 image의 정보 & preprocessor의 설정 정보를 deployer에게 전달
  - step 3) deployer는 이 정보들로 kubernetes 리소스를 생성

2.3.10 Config.fn

- input : 함수
- output : run()을 실행하는 함수

```
remote_train = fairing.config.fn(train)
remote_train()
```

2.3.11 fairing.ml_tasks

- fairing은 config 클래스에 좀 더 ML 트레인에 특화된 class를 제공함

2.4 Katib

2.4.1 소개

- 역할 2가지
  - 1) Hyperparameter Optimization (HyperOpt)
  - 2) Neural Architecture Search (NAS)
- kubeflow 설치 시, Jupyter notebook과 함께 쉽게 실행할 수 있는 컴포넌트

2.4.2 Hyperparameter tuning

Hyperparameter의 예시

- learning rate
- dropout rate
- # of layer
- cost function

→ 이 tuning 과정을 자동화 할 수 있도록 도와줌!

2.4.3 Neural Architecture Search (NAS)

AutoML의 하나인 NAS : 최적의 NN을 디자인하기 위해 사용

다양한 방법으로 NAS를 실행

- Katib는 **강화학습 기반**으로 탐색을 함

2.4.4 Architecture

Katib는 크게 4가지 개념으로 구성

- 1) Experiment : 최적화 "실행 단위" (하나의 job)
  - 총 5개의 영역으로 나뉨
  - 1-1) Trial Count
  - 1-2) Trial Template
  - 1-3) Objective
  - 1-4) Search Parameter
  - 1-5) Search Algorithm
- 2) Trial : 최적화 과정의 "반복 단위"
  - 1-1)의 Trial Count 만큼 Trial이 생성됨
  - 하나의 Trial에서 하나의 Worker Job이 실행됨
- 3) Suggestion :
  - 1-3)의 Search Algorithm을 통해 생성된 하이퍼파라미터 값의 모음(후보)
  - 하나의 Experiment 당 하나의 Suggestion이 생성됨
  - Experiment에서 설정된 Paramter & Algorithm이 만들어낸 value를 각 trial에 제공
- 4) Worker Job :
  - paramter & suggestion 값을 가지고 trial를 평가하며, 목표값을 계산

2.4.5 Experiment

- CRD (Customer Resource Definition)으로 정의

example

```
apiVersion: "kubeflow.org/v1alpha3"
kind: Experiment
metadata:
  namespace: kubeflow
  labels:
    controller-tools.k8s.io: "1.0"
  name: handson-experiment-1
spec:
  parallelTrialCount: 5 #----------------(1)
  maxTrialCount: 30     #----------------(2)
  maxFailedTrialCount: 3 #---------------(3)
  objective: #---------------------------(4)
    type: maximize
    goal: 0.99
    objectiveMetricName: Validation-accuracy
    additionalMetricNames:
      - accuracy
```

```
      - loss
      - Validation-loss
  algorithm: #--------------------------(5)
    algorithmName: random
  trialTemplate: #----------------------(6)
    goTemplate:
        rawTemplate: |-
          apiVersion: batch/v1
          kind: Job
          metadata:
            name: {{.Trial}}
            namespace: {{.NameSpace}}
          spec:
            template:
              spec:
                containers:
                - name: {{.Trial}}
                  image: brightfly/katib-job:handson
                  command:
                  - "python"
                  - "/app/katib_keras_mnist.py"
                  {{- with .HyperParameters}} #-----------------(7)
                  {{- range .}}
                  - "{{.Name}}={{.Value}}"
                  {{- end}}
                  {{- end}}
                restartPolicy: Never
  parameters: #---------------------------------------(8)
    - name: --learning_rate
      parameterType: double
      feasibleSpace:
        min: "0.01"
        max: "0.03"
    - name: --dropout_rate
      parameterType: double
      feasibleSpace:
        min: "0.1"
        max: "0.9"
```

- (1) parallelTrialCount : 병렬로 실행될 trial의 수
- (2) maxTrialCount : 최대로 실행될 trial의 수
- (3) maxFailedTrialCount : Trial의 실패 한도 수
- (4) objective : 수집할 대상에 대한 metric
- (5) algorithm : hyperparamter search algorithm
- (6) trialTemplate : trial의 template
- (7) 설정한 파라미터의 iteration 구문
- (8) hyperparameter의 입력 값

하나의 Trial 실행하기 :

```
$ python /app/katib_keras_mnist.py -learning_rate=0.012--dropout_rate=0.381
```

2.4.6 검색 알고리즘

- 1) grid search
- 2) random search
- 3) bayesian optimization
- 4) HYPERBAND
- 5) Hyperopt TPE
- 6) NAS based on reinforcement learning

2.4.7 Metric collector

각 trial의 metric (ex. accuracy, loss 등) 을 수집한다

collector의 타입을 정의

- 1) stdout (default 값)
- 2) file
- 3) tensorflow Flow Event
- 4) custom
- 5) None

2.4.8 Component

Katib는 여러 종류의 component로 구성됨

각 component는 kubernetes의 deployment로 실행

- katib-manager
- katib-db
- katib-ui
- katib-controller

2.5 Pipeline

2.5.1 소개

- **컨테이너 기반의 end-to-end ML workflow**를 만들고 배포할 수 있는 쿠버네티스 플랫폼
  (컨테이너 기반 : 확장성 & 재사용성 good)
- kubernetes의 자원을 관리하기 위해, 백엔드 프레임워크로 **argo**라는 worflow tool을 사용

kubeflow pipeline의 구성

- 1) **experiment, job, run을 추적/관리**하는 UI
- 2) ML workflow 단계별 **scheduling 엔진**
- 3) pipeline & 그 component들이 생성하는 **SDK**
- 4) SDK와 연동하는 **jupyter notebook**

kubeflow pipeline이 지향하는 바

- 1) 쉬운 pipeline 구성
- 2) 쉬운 pipeline 생성
- 3) 쉬운 재사용

2.5.2 Pipeline

- workflow의 component들이 **"그래프 형태로 결합"**된 것
- **입/출력**에 대한 정의도 포함
- pipeline을 통해 **업로드/공유 가능**
- pipeline component는 **docker image로 패키징**
  & **그래프 결합 형태에 따라 순서대로** 실행
- 과정)
  - 파이프라인이 실행되면,
    - 각 단계에 맞는 pod를 실행
      - 각 pod는 설정된 container를 실행
        - containter 안에 있는 application이 실행됨
  - **scheduler에 따라** 순서대로 container들이 실행됨

2.5.3 Component

- ML workflow의 한 단계 수행하는 코드 집합
- 함수와 유사
  - input, output, 이름, 상세 구현

2.5.4 Graph

- pipeline UI에서 runtime 실행을 나타내는 그림

2.5.5 Run, Recurring Run

- Run : pipeline의 단일 실행 단위
- Recurring Run : pipeline을 주기적으로 실행하는 run

2.5.6 Run Trigger

Run의 새로운 생성 여부를 알려주는 flag

2가지 type의 Run Trigger

- 1) periodic : 간격 기반의 scheduling
- 2) cron

2.5.7 Step

- step : pipeline에서 "하나의 component의 실행"

이후 생략

2.6 Training of ML models

- kubernetes job 뿐만 아니라, 다양한 ML 학습 모델 지원
- ex) TFJob, Pytorch, MPI, MXNet, Chainer 학습 등

2.7 Serving Models

2.7.1 개요

2가지의 serving system을 제공

- 1) KFServing (추천)
  - kubeflow ecosystem에 포함된 프로젝트
- 2) Seldon Core
  - kubeflow의 초기부터 지원

2.7.2 KFServing

- kubernetes에서 **severless 추론**을 가능하게 함
- ML framework를 운영 환경에서도 쉽게 사용