Building Kubeflow Components (Reusable and Lightweight)

Building Components

- A component is a block of code that performs one step in the Pipeline.
- Components can be created either by using Python function or YAML files
- Using python function leverages the standard features of python language making it easier to use
- YAML file requires engineers to learn new syntax
- Generally more verbose than the python equivalent
- Using YAML file eases the distribution of readily parseable file format to a variety of client applications

Take Note

Before building and compiling your pipeline, there are some steps required to ensure a smooth run especially because we are working with microk8s;

- Ensure you have docker installed in your environment.

  ```
  sudo snap install docker --classic
  ```

- Ensure you have your base images pulled from the container registry. The base images we used for the labs are python:3.7.1, pytorch/pytorch:latest and tensorflow/tensorflow:latest-gpu-py3.

  ```
  docker pull python:3.7.1
  ```

  ```
  docker pull pytorch/pytorch:latest
  ```

  ```
  docker pull tensorflow/tensorflow:latest-gpu-py3
  ```

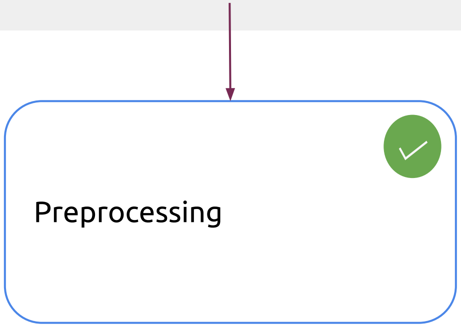
Building a Lightweight component.

Converting the steps into Components

- To explain the different ways kubeflow components are built, we would use the preprocessing step as our example.

- The codes for creating component and compiling of the pipeline are written using the jupyter notebook server on the kubeflow UI.

```
def preprocess(data):
     .
     .
if __name__ == '__main__':
  parser = 
argparse.ArgumentParser()
  parser.add_argument('--data')
  args = parser.parse_args()
  train_model(args.data)
```



Building Components - Python function (light weight)

my_python_func

Create a component and optionally write it to a file
`kfp.components.func_to_container_op`

`Func_to_container_op`
Pipeline component
my_python_func

If you wrote the component to a file, load it
`kfp.components.load_component_fom_file`

`load_component_from_file`
Pipeline component
my_python_func

Building Components - Python function(light weight)

- Install and import `kfp` and `dsl` libraries in the Kubeflow notebook.
- Write the stand-alone python function that represents a step in your ml workflow
- Specify inputs and return outputs

```python
def preprocess(data_path):
    import pickle
    # import Library
    import sys, subprocess;
    subprocess.run([sys.executable, '-m', 'pip', 'install','scikit-learn==0.22'])
    subprocess.run([sys.executable, '-m', 'pip', 'install','pandas==0.23.4'])
    import pandas as pd
    import numpy as np
    from sklearn.preprocessing import LabelEncoder
    from sklearn.preprocessing import OneHotEncoder
    from sklearn.model_selection import train_test_split
    from sklearn.preprocessing import StandardScaler

    # loading data from source
    data =
pd.read_csv("https://raw.githubusercontent.com/MavenCode/KubeflowTraining/master/Data/Churn_Modelling.csv")
    #dropping some columns that are not needed
    data = data.drop(columns=['RowNumber','CustomerId','Surname'], axis=1)
    #data features
    X = data.iloc[:,:-1]
    #target data
    y = data.iloc[:,-1:]
    #encoding the categorical columns
    le = LabelEncoder()
    ohe = OneHotEncoder()
    X['Gender'] = le.fit_transform(X['Gender'])
    geo_df = pd.DataFrame(ohe.fit_transform(X[['Geography']]).toarray())
    #getting feature name after onehotencoding
    geo_df.columns = ohe.get_feature_names(['Geography'])
    #merging geo_df with the main data
    X = X.join(geo_df)
    #dropping the old columns after encoding
    X.drop(columns=['Geography'], axis=1, inplace=True)
    #splitting the data
    X_train,X_test,y_train,y_test = train_test_split( X,y, test_size=0.2, random_state = 42)
    #feature scaling
    sc =StandardScaler()
    X_train = sc.fit_transform(X_train)
    X_test = sc.transform(X_test)

    # write predictions to results.txt
    with open(f'{data_path}/results.txt','w') as result:
        result.write(f'X_test: {X_test} | Actual {y_test}')
    return(print('Done!'))
```

Building Components - Python function(light weight)

- Convert your function into a component with `kfp.components.func_to_container_op(my_python_func)` and specify a base image

```
preprocess_op = kfp.components.func_to_container_op(preprocess, base_image="python:3.7")
```

Building a Reusable component

Building Components - Python Function (reusable)

my_code.py

Build a Docker container image and upload it
to a container registry

Docker container image

my_code.py

Use the pipelines DSL to create a function defining the
communication with the component's Docker container.
Optionally decorate with @`kfp.dsl.component`
Return a `@kfp.dsl.ContainerOp`

`@kfp.dsl.component`
`@kfp.dsl.ContainerOp`
Pipeline Component

Docker container image

my_code.py

Build components - Python function (reusable)

Here is how to build a reusable pipeline component using python function :

- Write your step in a python script
- In the function specify your libraries, methods, imports, arguments and outputs.

```
#importing libraries
import argparse

def preprocess(data):
    #importing libraries
    import joblib
    import pandas as pd
    import numpy as np
    from sklearn.preprocessing import LabelEncoder
    from sklearn.preprocessing import OneHotEncoder
    from sklearn.model_selection import train_test_split
    from sklearn.preprocessing import StandardScaler

     # loading data from source
    data =
pd.read_csv("https://raw.githubusercontent.com/MavenCode/KubeflowTraining/master/Data/Churn_Modelling.csv")
    #dropping some columns that are not needed
    data = data.drop(columns=['RowNumber','CustomerId','Surname'], axis=1)
    #data features
    X = data.iloc[:,:-1]
    #target data
    y = data.iloc[:,-1:]
    #encoding the categorical columns
    le = LabelEncoder()
    ohe = OneHotEncoder()
    X['Gender'] = le.fit_transform(X['Gender'])
    geo_df = pd.DataFrame(ohe.fit_transform(X[['Geography']]).toarray())
    #getting feature name after onehotencoding
    geo_df.columns = ohe.get_feature_names(['Geography'])
    #merging geo_df with the main data
    X = X.join(geo_df)
    #dropping the old columns after encoding
    X.drop(columns=['Geography'], axis=1, inplace=True)
    #splitting the data
    X_train,X_test,y_train,y_test = train_test_split( X,y, test_size=0.2, random_state = 42)
    #feature scaling
    sc =StandardScaler()
    X_train = sc.fit_transform(X_train)
    X_test = sc.transform(X_test)

    #saving output file to path
    np.save('X_train.npy', X_train)
    np.save('X_test.npy', X_test)
    np.save('y_train.npy', y_train)
    np.save('y_test.npy', y_test)

#defining and parsing arguments
if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('--data')
    args = parser.parse_args()
    print('Done with preprocessing')
    preprocess(args.data)
```

Build components - Python function (reusable)

- Create and push a docker image that packages the python script written above and upload the container image to a registry. For this walkthrough, we will be using DockerHub as our container registry.

```
FROM python:3.7.1
WORKDIR /preprocess_data
RUN pip install --upgrade pip \
&& pip install -U scikit-learn numpy pandas
COPY preprocess.py /preprocess_data
ENTRYPOINT ["python", "preprocess.py"]
```

Dockerfile

Find the codes to push a docker image in this repository.

How to upload Dockerfile to DockerHub

- Make sure you have an account on Docker hub before proceeding

- After creating the Dockerfile, you build the Dockerfile and specify the name and its tag. Then you run it on your local machine with docker run --rm <image-name:tag> to ensure it works.

```
docker build --tag=preprocess-component:v.0.1
```

- After accessing the Docker Hub UI, navigate to repositories, then name the repository (whatever name you want), set visibility to public and then click Create.

How to upload Dockerfile to DockerHub

Docker commands

Public View

To push a new tag to this repository,

```
docker push mavencodev/preprocess-component:tagname
```

Before pushing your image to the repository, it needs to first be associated with your Docker Hub repository . To do that you tag the local image with the new image using the `tag` command on your command prompt or desired terminal;

```
docker tag preprocess-component:v.0.1 mavencodev/preprocess-component:v.0.1
```

Then you finally push to the repository using the `push` command

```
docker push mavencodev/preprocess-component:v.0.1
```

Building Reusable components

- Now define your python function as a kubeflow component with the Kubeflow Pipelines DSL. The DSL defines your pipeline's interactions with the component's Docker container

- If you wish to enable static type checking in the DSL compiler, you can use `kfp.dsl.component`. The component function must return a `kfp.dsl.ContainerOp.`

```
def preprocess_op(data):
    return dsl.ContainerOp(
        name = 'Preprocess Data',
        image = 'mavencodev/preprocess-component:v.0.1',
        arguments = ['--data', data],
        file_outputs={
            'X_train': '/preprocess_data/X_train.npy',
            'X_test': '/preprocess_data/X_test.npy',
            'y_train': '/preprocess_data/y_train.npy',
            'y_test': '/preprocess_data/y_test.npy'
        }
    )
```

Building Components - YAML file

Use the YAML file from your python func component shown in the previous slide.

Pipeline Component YAML

Load the component from a URL

`@kfp.components.load_components_from_url`

Pipeline component

Converting python function into yaml file

- A yaml file can be created from your python functions. All you need do is conclude your python function with the following code:

```
#to export component into yaml file
    if __name__ == '__main__':
        kfp.components.create_component_from_func(
        preprocess, #function name
        output_component_file = 'preproess_component.yaml'),
        base_image = "python:3.7",
        packages_to_install = ['pandas==0.23.4', 'scikit-learn==0.22']
```

Yaml File created

preprocess_component.yaml

```yaml
name: Preprocess
inputs:
- {name: data_path}
implementation:
  container:
    image: python 3.7
    command:
    - sh
    - -c
    - (PIP_DISABLE_PIP_VERSION_CHECK=1 python3 -m pip install --quiet --no-warn-script-location
      'pandas==0.23.4' 'scikit-learn==0.22' || PIP_DISABLE_PIP_VERSION_CHECK=1 python3
      -m pip install --quiet --no-warn-script-location 'pandas==0.23.4' 'scikit-learn==0.22'
      --user) && "$0" "$@"
    - sh
    - -ec
    - |
      program_path=$(mktemp)
      echo -n "$0" > "$program_path"
      python3 -u "$program_path" "$@"
    - "def preprocess(data_path):\n    #importing libraries\n    import pickle\n \
      \    import sys, subprocess;\n    subprocess.run([sys.executable, '-m', 'pip',\
      \ 'install', 'pandas==0.23.4'])\n    subprocess.run([sys.executable, '-m', 'pip',\
      \ 'install', 'scikit-learn==0.22'])\n    import numpy as np\n    import pandas\
      \ as pd\n    from sklearn.preprocessing import LabelEncoder\n    from sklearn.preprocessing\
      \ import OneHotEncoder\n    from sklearn.model_selection import train_test_split\n\
      \     from sklearn.preprocessing import StandardScaler \n\n    # Load and unpack\
      \ the clean_data\n    with open(f'{data_path}/data','rb') as f:\n        data\
      \ = pickle.load(f)\n    #dropping some columns that are not needed\n    data\
      \ = data.drop(columns=['RowNumber','CustomerId','Surname'], axis=1)\n    #data\
      \ features\n    X = data.iloc[:,:-1]\n    #target data\n    y = data.iloc[:,-1:]\
      \   \n    #encoding the categorical columns\n    le = LabelEncoder()\n    ohe\
      \ = OneHotEncoder()\n    X['Gender'] = le.fit_transform(X['Gender'])\n    geo_df\
      \ = pd.DataFrame(ohe.fit_transform(X[['Geography']]).toarray())\n\n    #getting\
      \ feature name after onehotencoding\n    geo_df.columns = ohe.get_feature_names(['Geography'])\n\
      \     #merging geo_df with the main data\n    X = X.join(geo_df) \n    #dropping\
      \ the old columns after encoding\n    X.drop(columns=['Geography'], axis=1,\
      \ inplace=True)\n\n    #splitting the data \n    X_train,X_test,y_train,y_test\
      \ = train_test_split( X,y, test_size=0.2, random_state = 42)\n    #feature scaling\n\
      \     sc =StandardScaler()\n    X_train = sc.fit_transform(X_train)\n    X_test\
      \ = sc.transform(X_test)\n\n    #Save the test and train data as a pickle file\
      \ to be used by the train component.\n    with open(f'{data_path}/train_data',\
      \ 'wb') as f:\n        pickle.dump((X_train,  y_train), f)\n    with open(f'{data_path}/test_data',\
      \ 'wb') as f:\n        pickle.dump((X_test,  y_test), f)\n\nimport argparse\n\
      _parser = argparse.ArgumentParser(prog='Preprocess', description='')\n_parser.add_argument(\"\
      --data-path\", dest=\"data_path\", type=str, required=True, default=argparse.SUPPRESS)\n\
      _parsed_args = vars(_parser.parse_args())\n\n_outputs = preprocess(**_parsed_args)\n"
    args:
    - --data-path
    - {inputValue: data_path}
```

Creating and compiling a Pipeline from yaml files

- Load component

```
#importing SDK package
from kfp import components
#loading the component yaml file
component_op = components.load_component_from_url('https://raw.githubusercontent.com/...../component.yaml')
```

- Pipeline definition and description

```
@kfp.dsl.pipeline(
  name = "pipeline_name",
  description = "pipeline description")
```

Creating and compiling a Pipeline from yaml files

- Description of how each component would run in the pipeline

```
def any_name_pipeline():

  component_1 = component_1_op(#specify the parameters and define your outputs(.outputs))
```

- Compile and run pipeline

```
#compile your pipeline
kfp.Compiler.compiler().compile(pipeline_name, 'pipeline_name.zip')
kfp_endpoint=None
kfp.Client(host=kfp_endpoint).create_run_form_pipeline_func(pipeline_name, arguments={})
```

Creating and defining a Pipeline (reusable)

From the previous section:

- A pipeline is made up of connected components representing an ML workflow
- Kubeflow Pipelines SDK (set of python packages that can be used to specify and run ML workflows) are necessary for creation and compilation
- Use `kfp.dsl.pipeline()` – decorator for python functions which returns a pipeline.

```
#defining pipeline
@kfp.dsl.pipeline(
  name="pipeline_name",
  description="pipeline description"
)
#including all components and describing how it will run in the pipeline
def my_pipeline(
    #volume to share data between components can also be defined here

    #specify each components and specify its outputs using (.outputs)
```

Compiling a Pipeline with python func (reusable)

- Compiled using `kfp.compiler.Compiler.compile()`
- This gives a single static configuration in yaml format that the Kubeflow Pipelines service can process by compiling your python DSL code. The YAML file can be stored in .zip format as done below.

```
pipeline_func = my_pipeline #name of function used to define components during pipeline definition

experiment_name = "any_name"
# Compile pipeline to generate compressed YAML definition of the pipeline.
kfp.compiler.Compiler().compile(pipeline_func,
  '{}.zip'.format(experiment_name))
```