AI4EU Experiments Onboarding Tutorial: House Sales Price Prediction using gRPC

This tutorial provides a basic Python programmer's introduction to working with gRPC.

By walking through this example you'll learn how to:

- Define a service in a .proto file.
- Generate server and client code using the protocol buffer compiler.
- Use the Python gRPC API to write a simple client and server for your service.

It assumes that you have read the Overview and are familiar with protocol buffers. You can find out more in the proto3 language guide and Python generated code guide.

What is gRPC?

With gRPC you can define your service once in a .proto file and implement clients and servers in any of gRPC's supported languages, which in turn can be run in environments ranging from servers inside Google to your own tablet - all the complexity of communication between different languages and environments is handled for you by gRPC. You also get all the advantages of working with protocol buffers, including efficient serialization, a simple IDL, and easy interface updating.

This example is a Machine Learning Regression example that lets clients get the house sales prediction based on chosen attributes.

Steps

1. Write the service to be served.
2. Make a proto file to define the messages and services.
3. Use the proto file to generate gRPC classes for Python

4. Create the server.
5. Create the client.
6. Include a license
7. Prepare the docker file, run the docker and the run the client in a new tab.

Step 1: Write the Service:

In our case, the service is predicting house pricing. Below is code snippet.

```
import pandas as pd
import numpy as np



# Read the data and store in a dataframe called training_set
train_data_path ='train.csv'
training_set = pd.read_csv(train_data_path)

# Select the target variable and call it y
y = training_set.SalePrice

# Create a list of the predictor variables
predictors = ["MSSubClass","LotArea", "YearBuilt","BedroomAbvGr",
"TotRmsAbvGrd"]

# Create a new dataframe with the predictors list
X = training_set[predictors]

# Import DecisionTreeRegressor
from sklearn.tree import DecisionTreeRegressor

# Define the first model
tree_model = DecisionTreeRegressor()

# Fit model
tree_model.fit(X, y)


def predict_sale_price(MSSubClass, LotArea, YearBuilt, BedroomAbvGr,
TotRmsAbvGrd):
    prediction = tree_model.predict([[MSSubClass, LotArea, YearBuilt,
BedroomAbvGr,     TotRmsAbvGrd]])
    return (prediction)
```

This model has 5 input arguments (features of the house that we want to predict its sale price).

Since our motive here is to understand the gRPC, I have taken a simple DecisionTreeRegressor from sklearn.tree.

Step 2: Make the Proto File:

The proto file must be explicitly named model.proto as Acumos expects it that way. Also note that package names must be globally unique to let AI4EU Experiments distinguish the protobuf definitions for all onboarded models.

```
//Define the used version of proto:
syntax = 'proto3';

package fraunhofer.houseprediction;

//Define a message to hold the features input by the client :
message Features{
    float MSSubClass     = 1 ;
    float LotArea        = 2 ;
    float YearBuilt      = 3 ;
    float BedroomAbvGr   = 4 ;
    float TotRmsAbvGrd   = 5 ;
}
//Define a message to hold the predicted price :
message Prediction{
    float salePrice      = 1 ;
}

//Define the service :
service Predict{
    rpc predict_sale_price(Features) returns (Prediction){}
}
```

Here, we did not give values to the features, those numbers indicate the order of serializing the features.

Step 3: Generate gRPC classes for Python:

Open the terminal, change the directory to be in the same folder that the proto file is in.

To generate the gRPC classes we have to install the needed libraries first:

#Install gRPC :

python3 -m pip install grpcio

#To install gRPC tools, run:

python3 -m pip install grpcio-tools googleapis-common-protos

Now, run the following command:

python3 -m grpc_tools.protoc -I. --python_out=. --grpc_python_out=. model.proto

This command used model.proto file to generate the needed stubs to create the client/server.

The files generated will be as follows:

model_pb2.py — contains message classes

- model_pb2.Features for the input features
- model_pb2.Prediction for the prediction price

model_pb2_grpc.py — contains server and client classes

- model_pb2_grpc.PredictServicer will be used by the server
- model_pb2_grpc.PredictStub the client will use it

Step 4: Creating the Server:

The server will import the generated files and the function that will handle the predictions.

Then we will define a class that will take a request from the client and uses the prediction function to return a respond.

The request gives us the five features, the response is a prediction.

After that, we will use **add_PredictServicer_to_server** function from (**model_pb2_grpc.py**) file that was generated before to add the class PredictSevicer to the server.

Once you have implemented all the methods, the next step is to start up a gRPC server so that clients can actually use your service.

Below is the house_sale_prediction_client.py

```python
import grpc
from concurrent import futures
import time

# import the generated classes :
import model_pb2
import model_pb2_grpc

# import the function we made :
import predict_sale_price as psp
port = 50055

# create a class to define the server functions, derived from

class PredictServicer(model_pb2_grpc.PredictServicer):
    def predict_sale_price(self, request, context):
        # define the buffer of the response :
        response = model_pb2.Prediction()
        # get the value of the response by calling the desired function :
        response.salePrice =
psp.predict_sale_price(request.MSSubClass,request.LotArea, request.YearBuilt,
request.BedroomAbvGr, request.TotRmsAbvGrd)
        return response


# creat a grpc server :
server = grpc.server(futures.ThreadPoolExecutor(max_workers = 10))

model_pb2_grpc.add_PredictServicer_to_server(PredictServicer(), server)

print("Starting server. Listening on port : " + str(port))
server.add_insecure_port("[::]:{}".format(port))
server.start()

try:
    while True:
        time.sleep(86400)
except KeyboardInterrupt:
    server.stop(0)
```

Step 5: Creating the Client:

In the client file we will do the following:

- Open a gRPC channel
- Create a stub
- Create a request message
- Use the stub to call the service

Below is the code snippet for client:

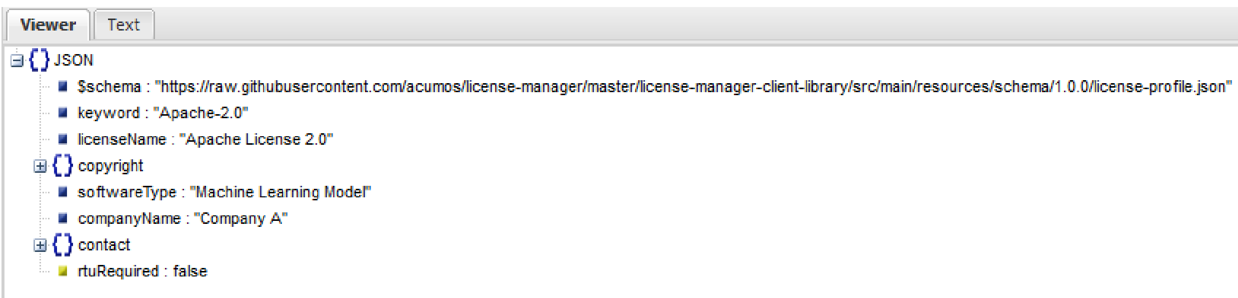
```
import grpc
from random import randint
from timeit import default_timer as timer

# import the generated classes
import model_pb2
import model_pb2_grpc

start_ch = timer()
port_addr = 'localhost:50055'
# open a gRPC channel
channel = grpc.insecure_channel(port_addr)
# create a stub (client)
stub = model_pb2_grpc.PredictStub(channel)
end_ch = timer()
MSSubClass   = [randint(1,11) for i in range(0,1000)]
LotArea      = [randint(100,1500) for i in range(0,1000)]
YearBuilt    = [randint(1915,2000) for i in range(0,1000)]
BedroomAbvGr = [randint(2,10) for i in range(0,1000)]
TotRmsAbvGrd = [randint(2,12) for i in range(0,1000)]
ans_lst = []
start = timer()
for i in range(0,len(MSSubClass)-1):
    # create a valid request message
    requestPrediction  = model_pb2.Features(MSSubClass = MSSubClass[i], LotArea =
LotArea[i],
                                        YearBuilt = YearBuilt[i], BedroomAbvGr =
BedroomAbvGr[i],
                                        TotRmsAbvGrd = TotRmsAbvGrd[i])
    # make the call
    responsePrediction = stub.predict_sale_price(requestPrediction)
    ans_lst.append(responsePrediction.salePrice)
    print('The prediction is :',responsePrediction.salePrice)
print('Done!')
end = timer()
all_time = end - start
ch_time = end_ch - start_ch
print ('Time spent for {} predictions is {}'.format(len(MSSubClass),(all_time)))
print('In average, {} second for each prediction'.format(all_time/len(MSSubClass)))
print('That means you can do {} predictions in one
second'.format(int(1/(all_time/len(MSSubClass)))))
print('Time for connecting to server = {}'.format(ch_time))
```

Step 6: Include a license File

We need to include a license file before building a docker image. I have added a sample apache license in this example.



Step 7: Prepare the Docker file

```
FROM ubuntu:16.04

MAINTAINER Tejas  "tejas.morbagal.harish@iais.fraunhofer.de"

RUN apt-get update -y
RUN apt-get install -y python3-pip python3-dev
RUN pip3 install --upgrade pip
RUN pip3 install numpy
RUN pip3 install pandas
RUN pip3 install sklearn

COPY ./requirements.txt /requirements.txt
COPY license-1.0.0.json model.proto model_pb2.py model_pb2_grpc.py
house_prediction_server.py predict_sale_price.py train.csv test.csv ./

WORKDIR /

RUN pip3 install -r requirements.txt

COPY . /

ENTRYPOINT [ "python3","house_price_prediction_server.py" ]
```

In the docker file, we copy the license file along with other files required to the container.

Whenever any layer is re-built all the layers that follow it in the Dockerfile need to be rebuilt too. It's important to keep this fact in mind while creating Dockerfiles.

The dockerfile here separates out the gRPC specific requirements in a separate file called requirements.txt. The reason for doing this is to separate the application dependency from the gRPC dependency. gRPC dependency in requirements.txt will be built as a separate layer when the Docker image is built. This avoids rebuild of this layer every time a change is made in the application. Below is the contents of gRPC requirement.txt.

```
# GRPC Python setup requirements
coverage>=4.0
cython>=0.29.8
enum34>=1.0.4
protobuf>=3.5.0.post1
six>=1.10
wheel>=0.29
```

Build the docker image

1) docker build -t ml_regression_grpc:latest .

Run the docker image

2) docker run --name regression_grpc -p50055:50055 ml_regression_grpc:latest

The -p option maps the port on the container to the host.

The Docker run internally executes house_price_prediction_server.py.

Open one more terminal and run the client which now can access the docker server

3) python3 house_price_prediction_client.py