**ML Model Deployments**

*Introduction:*

Model deployment is the last stage of ML/AI project and engineers/researchers must ensure the implemented model is available for end-users to use the same.

Further, many parameters to be considered before deploying the model, such as,

(i) data storage

(ii) end user/client/stakeholder

(iii) number of users

(iv) monitoring strategy and few more.

In this section, major deployment models have been explained in detail. In all our samples, IRIS model sample has been used. As the first part of learning the deployment approaches, we need to learn the workflow of machine learning model.

*Machine Learning Model Workflow*

In the normal development lifecycle of machine learning model, model will be implemented first and it will be made available to the use to test the performance or end-users' access. The simple model's workflow has been shown in the diagram [1]. Upon deploying the machine learning model, user will send the prediction request via any user interface. User interface will accept user's input and send or forward the same to web server, and webserver will delegate the prediction request to machine learning model, get the response back and send the response to the user interface. Finally, user will be given the prediction result for the submitted query.

*[Diagram 1] Visual Representation of Machine Learning Model Workflow*

Machine generated alternative text:
User access UI to send 
prediction requests 
and receive 
responses. 
User 
Interface 
Machine 
Learning Model 
to 
espoose f 
ro 
Serv 
Web Framework 
(Flask/etc) 

The workflow shown in the diagram[1] can be customized further based on the complexity of the model and environment setup.

*Save and Load the model file:*

We should be able to finalize the implemented model only after testing the model with new data and comparing the model with other models. In this section, we have taken a simple IRIS model for experimenting all the deployment types. As the main target is about learn the different deployment types, less priority has been given to train and validate the model. Once the basic model is trained, it will must be saved and restored/reloaded for the later of deployment. There are two approaches , namely (i) Pickle and (ii) Joblib has been selected to test the save/load logic.

TODO : Install Pickle and Joblib

(i) Pickle Method

Pickle is a python library, has been used for serialization/de-serialization. i.e. it has been used to store the structured data into file and restore the same from the desk. It offers dump() method to save and load() method to load the content from the disk.

| #pickle file name to store and load the model  pickle\_filename = "Iris\_Classifer.pkl"    #save the model  with open(pickle\_filename, "wb") as file:  pickle.dump(model, file)    #load the model  with open(pickle\_filename, 'rb') as file:  iris\_model\_pickle = pickle.load(file) |
| --- |

Above few lines of code shows that how to save and load the pickle file. First, filename string is initialized and next section saves the model (has the trained model) into the given filename (i.e. pickle\_filename = "Iris\_Classifer.pkl"). The file mode "wb" refers that write mode in binary format. Subsequently, same file has been opened in "rb" mode (read mode for binary file) and loaded from the disk for the further usage.

(ii) Joblib Method

The Joblib Module comes part of Scikit Learn package and it can be used as the replacement for Pickle. One of the advantages of using this module is, it doesn't require the additional file object to save/load the model. Further, Joblib library is more efficient when it deals with large array and below few lines of code shows show how the joblib can be used to save and restore the model.

| #joblib file name to store and load the model  joblib\_filename = "Iris\_Classifer.jbl"  joblib.dump(model, joblib\_filename)    #load the model  iris\_model\_joblib = joblib.load(joblib\_filename) |
| --- |

First, filename is initialized and model object is saved in disk using the method dump. Afterwards, load() method is called to restore the saved file. As Jobllib is more efficient and easy to use, Joblib library has been used in all the deployment samples for loading the file.

Full Code (filename : iris\_model.py)

TODO: copy Full code

*Explanation:*

The programs starts with importing all the required libraries. Upon importing the required libraries, iris dataset has been loaded from the github repository and saved in a variable "dataset". The first four columns' values and last 'Y' column values are extracted and saved in separate variables, 'X' and 'Y' respectively . The "train\_test\_split" API has been used to split the given dataset into 70% training and 30% test datasets. The SVM(Support Vector Machine) library's SVC(Support Vector Classification) implementation has been used as classification algorithm. The SVC method uses RBF(Radial Basis Function) Kernel as default and gamma value(SVM performance parameter) is set to auto. In the following lines, model is trained using "fit" method and validation results have been obtained by calling "predict" function from the model with "x\_validations" parameter.

Though we don’t focus more on the model accuracy here, the following lines prints the accuracy score, confusion matrix and classification report evaluation metrics. Lastly, trained model has been saved and restored using Pickle as well as Joblib approaches.

*Run the source code:*

> python iris\_model.py

Output :

| Accuracy Score : 97.77777777777777  Confusion Matrix :  [[11 0 0]  [ 0 17 1]  [ 0 0 16]]  Classification Report :  precision recall f1-score support    Iris-setosa 1.00 1.00 1.00 11  Iris-versicolor 1.00 0.94 0.97 18  Iris-virginica 0.94 1.00 0.97 16    accuracy 0.98 45  macro avg 0.98 0.98 0.98 45  weighted avg 0.98 0.98 0.98 45    Prediction result(Pickle model file) for [5.1, 3.5, 1.4, 0.2] : Iris-setosa  Prediction result(Joblib model file) for [5.1, 3.5, 1.4, 0.2] : Iris-setosa |
| --- |

The result shows that trained model's evaluation metrics proves that model has been trained well. Finally, Pickle and Joblib can store and restore the model, and restored model is able to predict the result as expected. The code saves two files, namely, "Iris\_Classifer.pkl" (Pickle) and "Iris\_Classifer.jbl" (Joblib) in the current folder and the joblib file will be used in the rest of the deployment experiments.

1. **Using Flask Approach**

Flask is a web framework, written in python and it lets the developers to implement the web applications very easily. Its framework is categorized as microframework and it's based on WSGI(Web Server Gateway Interface) framework. The reason for categorizing Flask as microframework is, it's designed to have the core of the web application and easy to scale based on the requirement.

*Steps involved in ML model deployment*

(i) Build the model and save the same using Joblib.

(ii) Install Flask

(iii) Implement server side python app and expose API using flask.

(iv) Run the server

(v) Access the API using user interface.

As the very first step has been already completed, the steps from (ii) to (v) been explained below:

*Command to install Flask*

pip install flask

The next important stage is to implement the python application to use Flask and expose API. The simple Flask application code looks below:

from flask import Flask

app = Flask(\_\_name\_\_)

@app.route('/')

def home():

return 'Flask is active...!'

The very first line import the Flask class and an instance of this class will act as WSGI application. In the following line, Flask class's instance has been created and the argument \_\_name\_\_ has been passed. We may pass different argument if required but this argument is essential to Flask to know the place to look for the templates and other resources. The route() decorator carries the URL detail and this has been mapped to the triggering function. When the user calls the URL from the UI, linked function will be executed to process and give response to the user request.

Finally, Flask's instance run() function to be called to activate the server and the "debug=True" parameter has been passed to run the application in debug mode and the server application will be restarted if there is any changes made while running and helps to track the errors if any:

app.run(debug=True)

Full code (filename : model-depoyment-flask.py)

TODO: copy Full code

The Flask based deployment model code starts with importing the required modules, such as, Flask, request, json and joblib. In the following section, Flask's instance has been created and assigned to a global variable "app". As we have already saved the prediction model in the file "Iris\_Classifer.jbl", it has been restored/loaded using joblib and kept it in "joblib\_model" variable.

There are two APIs have been exposed, one is home and other one is predict-class. The home API URL is enabled to test the server is active or not. However, other API predict-class does the real prediction job. It takes four essential parameters(sepal-length, sepal-width, petal-length and petal-width) that require for the model prediction. Inside the "predict\_class" method, all the four parameters' values have been read and assigned to the different variables. The "predict" function of the model has been called with the four parameters and the result has been converted as json and returned to the caller.

Switch to the command prompt and locate the "model-depoyment-flask.py" folder and run the below command:

>python model-depoyment-flask.py

\* Serving Flask app "model-depoyment-flask" (lazy loading)

\* Environment: production

WARNING: This is a development server. Do not use it in a production deployment.

Use a production WSGI server instead.

\* Debug mode: on

\* Restarting with windowsapi reloader

\* Debugger is active!

\* Debugger PIN: 349-372-776

\* Running on <http://127.0.0.1:5000/> (Press CTRL+C to quit)

As shown above, Flask application is running on the local host (its default IP address 127.0.0.1) and listening to the port 5000. As the two APIs have been exposed in the flask server, copy the following urls, paste it on a web browser and press enter to see the result:

(a) <http://127.0.0.1:5000/>

Machine generated alternative text:
e 127.0.0.1:5000 
C 0 1270015000 
IRIS Classification Model(Flask based) is running successfully!!! 

(b) <http://127.0.0.1:5000/predict-class?sepal-length=5.1&sepal-width=3.5&petal-length=1.4&petal-width=0.2>

Machine generated alternative text:
127.0.0.1 x + 
C 0 127.0.0.1 1.4&petal-width=0.2 
{"predicted-result": "Iris-setosa"} 

As of now, we have tested our first Flask APIs using web browsers and this must the called from standard UI.

**2. Streamlit based Solution**

As of now, we have created our very first and simple ML deployment without any standard UI.

However, User Interface is very essential to easily access, enter/type the input and test the model. As the standard UI designing using technologies like Angular and React JS require significant effort, and it may not require for quick deployment, Streamlit based UI design and deployment for ML model has been explained in this chapter.

Streamlit is a simple and popular open-source framework, can be used to design very fast python based User Interface for ML projects. If the model owner does not have deep knowledge on UI development, Streamlit will be an incredibly good option as the Streamlit users do not need any prior knowledge on UI design to build a simple UI using python based Streamlit.

*Installing Streamlit library:*

pip install streamlit

The following simple Streamlit methods have been used in the deployment sample:

import streamlit as st

st.title("title text")

string\_value = st.text\_input('label text',0.0)

st.button("button label")

st.success('success message')

The streamlit library has been given an alias "st". The title() method is used to set the application title, text\_input() method receives single input from the user and return it as string. The major parameters of text\_input() methods are label text and default value. The button() method has been used to draw the button control with the given label. When the button is clicked, its code block will be executed. Finally, success() method used to show the success message on the UI and the message to be displayed has to be passed as the parameter of success() method.

TODO: copy Full code

Upon importing the required libraries, model filename has been initialized and load the file from the disk for the prediction. The python user defined method "preduct\_class" with four parameters has been added. This function call the predict function with given four parameters and return the predicted result the caller of the method. The second user defined method show\_UI() has very simple Streamlit code to show the UI components. The Streamlit user shows the title and four input controls to collect user’s input. Further, one button control (label : Predict Class) has been added. When the user clicks this button, its code block calls predict class() method for the prediction result. Finally, success() method is called with the predicted result to show it on the UI.

Run the Streamlit application:

Switch to the command prompt and locate the "model-deployment-streamlit.py" folder and run the below command:

>streamlit run model-deployment-streamlit.py

*You can now view your Streamlit app in your browser.*

*Local URL:* [*http://localhost:8501*](http://localhost:8501)

*Network URL:* [*http://192.168.0.103:8501*](http://192.168.0.103:8501)

The "Streamlit run" command has been used to run the Streamlit application from command line and it uses the port 8501 by default. We can copy any one of the url, paste it on the web browser and press enter. It will trigger the Streamlit that listen to the port 8501 and it displays the designed UI. Now, user can enter the four values in the input controls and click "Predict Class" button. It shows the predicted result just below the button.

Machine generated alternative text:
model-deployment-stream lit • St X 
C O localhost:8501 
IRIS Flower Classification 
septal-length 
0.0 
septal-width 
0.0 
petal-length 
0.0 
petal-width 
0.0 
Predict Class 
model-deployment-stream lit • St X 
C O localhost:8501 
IRIS Flower Classification 
septal-length 
5.1 
septal-width 
3.4 
petal-length 
1.1 
petal-width 
0.4 
Predict Class 
{"predicted-result": "Iris-setosa"} 

The UI on the left side shows the empty input controls and right side UI shows the filled input control with predicted result value.

**3. FastAPI**

The FastAPI is a python-based web framework, helps to create REST APIs very fast. It can be used as an alternative of Flask due to the following reasons:

(i) It's performance is high comparing with Flask.

(ii) Very stable web framework with fewer bugs.

(iii) Easy to learn and implement. Also, amazingly simple syntaxes help to reduce the code size.

(iv) It provides free swagger documentation. It helps the users to get the REST APIs list and parameters with zero dev effort.

Installing FastAPI :

*pip install fastapi*

Upon installing FastAPI, uvicorn also need to be installed to run the FastAPI. The following command can be used to install the uvicorn package:

*pip install uvicorn*

Minimum python code for FastAPI

from fastapi import FastAPI

#declare FastAPI instance

fastAPIInstance = FastAPI()

@fastAPIInstance.get('/')

def home():

return 'Fast API Home....'

The FastAPI gets enabled in python application by importing the FastAPI class from fastapi package. The following line, creates the instance of FastAPI and assign the class instance to the variable. Once the instance is created ( "fastAPIInstance "), routing methods need to be linked to the code for the response. In the above code, "get('/') refers the FastAPI home has been linked to the home() method. So when the user access the home URL from the browser, the linked method home() will be executed and it returns the response. So user can add different URIs and map different methods depends on the requirements.

TODO: copy Full code

The simple FastAPI based python code exposes two APIs outside, namely, (i) default or home and (ii) predict-class.

The first four lines, import the required packages for the application. In addition to the known packages, uvicorn has been added additionally.

Uvicorn is a lightning-fast ASGI(Asynchronous Server Gateway Interface) server implementation, ASGI is a successor of WSGI and it provide the interface between asynchronous behaviour python web servers, frameworks and applications. The WSGI provides synchronous behaviour and ASGI provides one for both asynchronous and synchronous apps.

Upon importing the packages, FastAPI() call is made to create an instance of FastAPI and it has been saved in the local variable "fastAPIInstance".

The subsequent lines, load the saved classification model from the disk for the prediction purpose. The first default REST API returns the text "ClassificationModel(FastAPIbased)isrunningsuccessfully!!!" when the url is home or default url is accessed. The second API has the main implementation to accept the users four input parameters, send the same to the trained/loaded model and receives the response back. Finally, received response has been converted to JSON and returns to the caller.

Running the FastAPI server code:

Switch to the command prompt and locate the "model-deployment-fast-api.py" file's folder and run the below command:

>uvicorn model-deployment-fast-api:fastAPIInstance --reload

[32mINFO[0m: Uvicorn running on [1mhttp://127.0.0.1:8000[0m (Press CTRL+C to quit)

[32mINFO[0m: Started reloader process [[36m[1m20544[0m] using [36m[1mstatreload[0m

[32mINFO[0m: Started server process [[36m16376[0m]

[32mINFO[0m: Waiting for application startup.

[32mINFO[0m: Application startup complete.

FastAPI runs on port number 8000 by default and the following table gives the clarity on the command and parameters being used to run the application.

| **uvicorn** | **ASGI server.** |
| --- | --- |
| model-deployment-fast-api | python file name which has the FastAPI server code. |
| fastAPIInstance | FastAPI class object. |
| --reload | flag that helps to reload automatically if there is any code changes. |

As we are already familiar with the endpoints and parameters, the two urls can be accessed as follows:

(i) Access the default/home URL : <http://127.0.0.1:8000/>

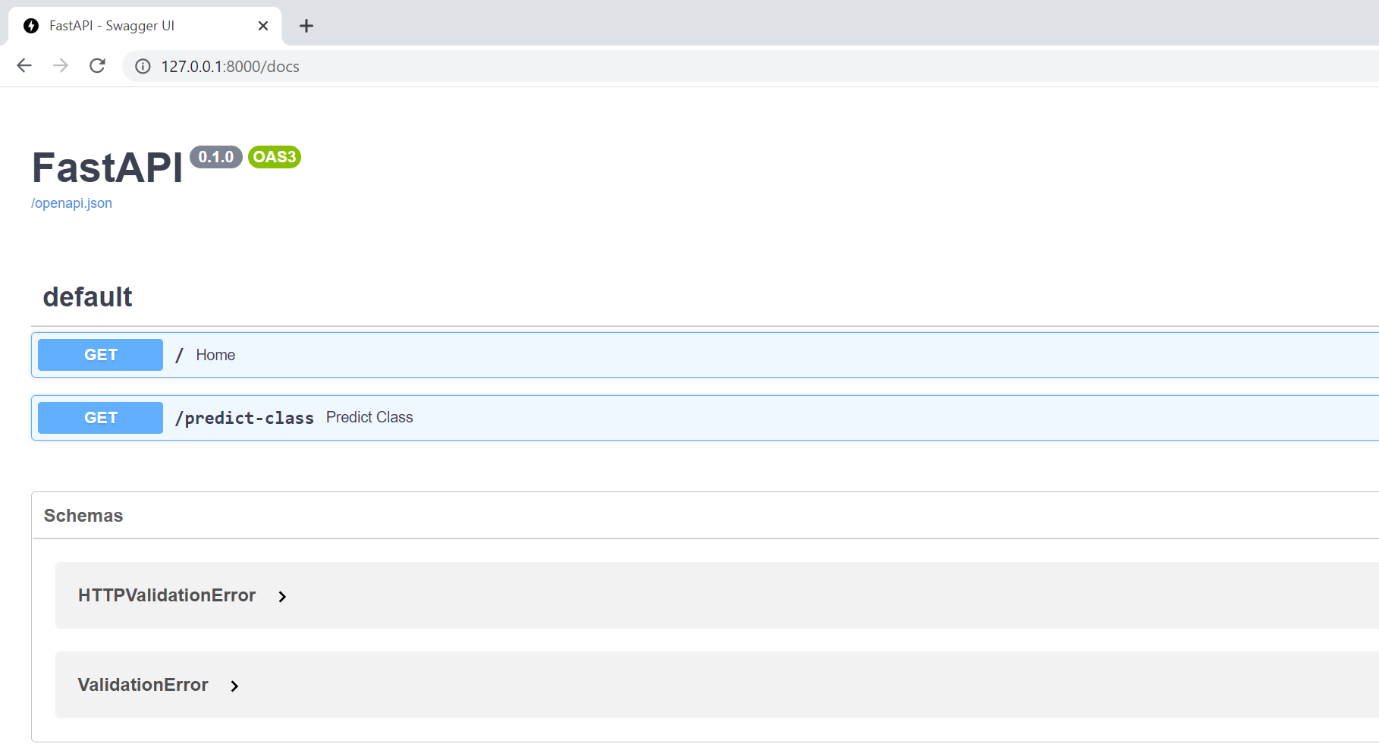
Machine generated alternative text:
e 127.0.0.1:8000 
C 0 127.0.0.1:8000 
"Classification Model(FastAPI based) is running successfully! ! ! " 

(ii) Access the predict-class for the given parameters: <http://127.0.0.1:8000/predict-class?sepal_length=5.1&sepal_width=3.5&petal_length=1.4&petal_width=0.2>

Machine generated alternative text:
127.0.0.1 x + 
C 0 127.0.0.1 width=3.5&petal length=l .4&petal width=0.2 
" { \ "predicted-result\ 
\ "Iris-setosa\ "Y' 

(iii) Accessing swagger documentations on the end points

The url "<http://127.0.0.1:8000/docs>" gives the list of exposed APIs and its parameters list for the easy access.



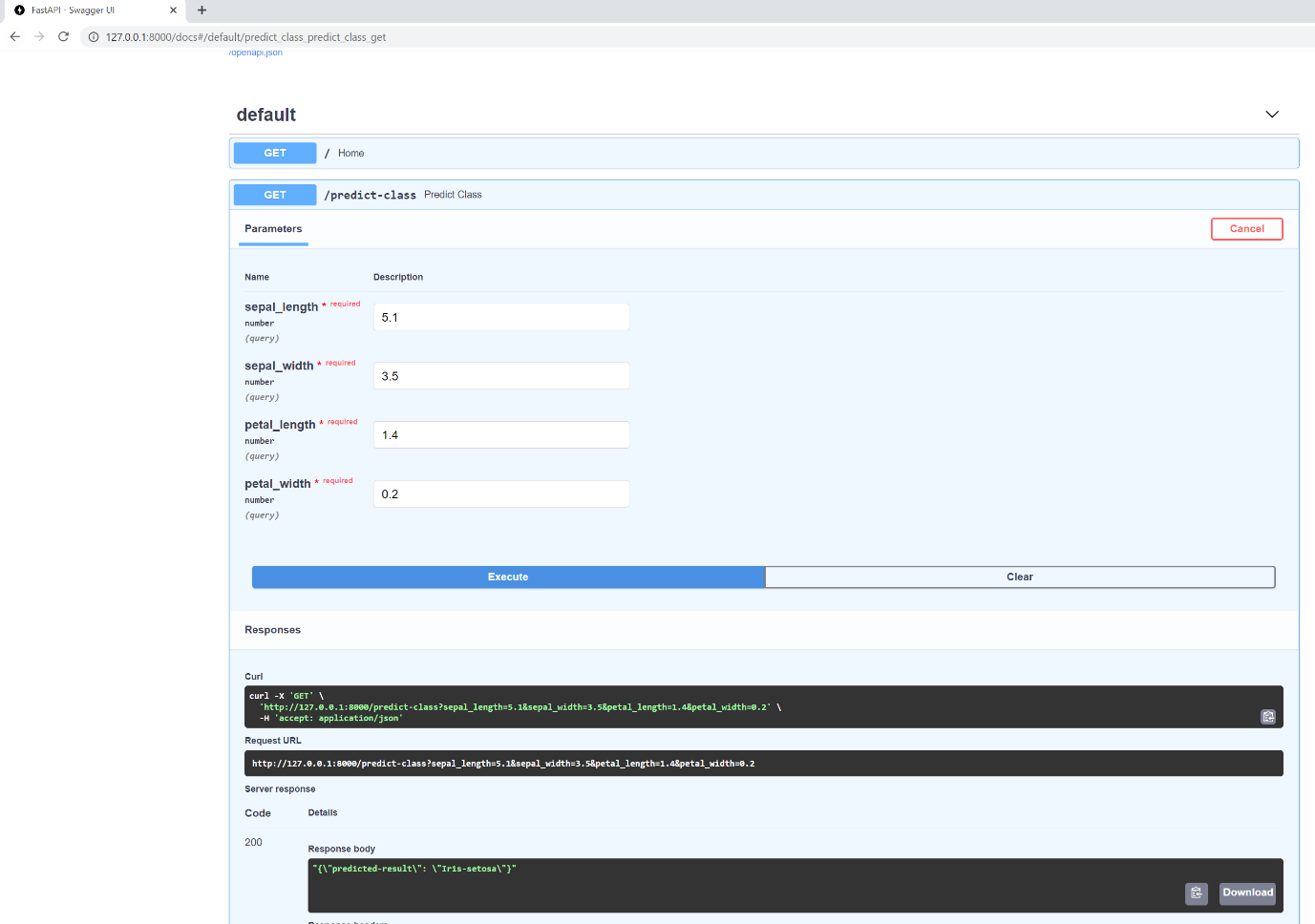
The APIs can be tested easily using the following steps:

(a) expand the predict class API detail by just clikicking on "Predict Class"

(b) click on the "Try out" button on the right top.

(c ) enter input values for the four parameters

(d) click on execute to see the below result:



(iv) Fancy documentation can be accessed using : <http://127.0.0.1:8000/redoc>

Machine generated alternative text:
x 
O 
FastAPl - ReDoc 
q Search... 
Home 
127.0.0.1 :8000/redoc 
FastAPI (0.1.0) 
Predict Class 
Download OpenAPl specification: 
Documentation Powered by ReDoc 
Home 
Responses 
> 200 Successful Response 
predict Class 
Download 
Response samples 
200 
Content type 
application/json 
null 
Copy 
/ predict—class 
QUERY PARAMETERS 
_H sepal length 
required 
sepal width 
required 
petal length 
required 
petal width 
required 
Response samples 
200 
422 
number (Sepal Length)l 
number (Sepal Width) 
number (Petal Length) 
number (Petal Width) 
Content type 
application/json 
Copy 
null 
Expand all 
Expand all 
Collapse all 
Collapse all 

4. Message Queue

Message queue mechanism gives the asynchronous solution to send request and receive response. In message queue, application send messages to receiver asynchronously and sender doesn't need to wait for the processed response immediately. A message will be sent to a queue and it will stay in queue until the receiver consume it.

A message can be any type of information and usually it will have task detail to be processed. The diagram [D12] shows the simple message queue. The enqueue typically insert/send the message inside the queue and all the messages inside the queue will be saved and served in FIFO manner. The Dequeue function remove the oldest message from the queue.

Machine generated alternative text:
MESSAGE 
Enqueue 
MESSAGE QUEUE 
MESSAGE #2 
MESSAGE 
Dequeue 

D12: Simple Message Queue (diagram from : <https://www.cloudamqp.com/blog/part1-rabbitmq-for-beginners-what-is-rabbitmq.html>)

Message queue can be used as inter-process communication medium, where the queue live inside the server. The consumers or workers will connect the queue and consume the message to process the task.

*Message Queue architecture*

The basic architecture of a message queue is shown the below diagram. The client application(s) are usually called producers, that generate and deliver the message to the broker or message queue. The message queue will act as a buffer to store all the received message from the producer(s). The consumer application consumes the message from the message queue and process it. The messages will live in the queue until the consumer process it.

Machine generated alternative text:
Publish 
PRODUCER 
BROKER 
RabbitMQ 
Consume 
Subscribe 
CONSUMER 

D13 : RabbitMQ (diagram from : <https://www.cloudamqp.com/blog/part1-rabbitmq-for-beginners-what-is-rabbitmq.html>)

In ML model deployment, RabbitMQ has been used to gain the asynchronies behaviour. When the model is bit complicated, requires external resources' access and spend more time to process the prediction requests', synchronous response behaviour is the not approach. Because (i) current user session will stay for long time until the prediction request is fully processed and (ii) other requesters will not get a chance to access the prediction model until the existing request is processed. So the second issue is very critical one and message queue gives solution by the async behaviour.

*Proposed Customised flow*

The diagram D14 shows the customized architecture for gaining asynchronous behaviour for ML model prediction. There are three major software components involved in the diagram, namely, (i) Server application, (ii) Message Queue and (iii) Worker applications. In the diagram, only one server application and any number of workers are shown. The worker applications count will be decided based on the volume of incoming requests. A dictionary data structure has been used inside the server application. The dictionary data structure stores all the processed results and its job id. In the real-time application, we may need to use permanent datastore (e.g. RDNMS or File ) to save the results based on the requirement.

Machine generated alternative text:
Users 
Workers 
Message Queue 
Server 

D14 : Customized architecture diagram for ML deployment using RabbitMQ

*The sequence of steps for the proposed architecture has been summarised below:*

(i) User application or client application or producer will submit or send the model prediction request to the server application.

(ii) Once the server application receives the prediction request, it will create a job id for the current request and respond to the request with the created job id. This is Async behaviour as the server respond back to the producer soon after receiving the request. So, client application can show the user that submitted request is being processed now.

(iii) Server application inserts the request along with the job id into the message queue.

(iv) The active and workload free worker application will pick the oldest message or request from the queue and process the request.

(v) Once the request is fully processed by the worker application, the result will be sent to the server application along with its job id and it will be kept in the dictionary data structure. In the dictionary, job id will be used as the key and result will be saved as value. Upon processing the current request, worker application will check the message queue for the next task and continue its processing. As shown in the diagram, many worker applications would run and process the requests from the message queue and send the results back to the server.

(vi) The client application will check the Server application periodically for the status of job id. When the server application is asked for the status of the job id, server application will check the dictionary to confirm the required result is saved in the local dictionary or not. If the job id (or the key) exist, Server will reply with the prediction result. Otherwise, it will send the busy or in-progress status reply to the client application

The Rabbit Queue based ML deployment solution use the summarised steps in the source code.

*RabbitMQ Installation/Setup*

The message queue based ML model deployment approach require few extra steps comparing with previous simple deployment models.

(1)Erlang installation : As the first step, erlang software must be installed on the system. Erlang is a programming language being used to build scalable systems and RabbitMQ server program has been written in Erlang. The following url can be used to download and install Erlang software:

Erlang download url : <https://www.erlang.org/>

(2) RabbitMQ installation : The RabbitMQ can be installed subsequently. There are different versions for different operating systems and its available in the below url:

RabbitMQ download url : <https://www.rabbitmq.com/download.html>

The suitable RabbitMQ version can be downloaded and installed on the user’s PC.

(3) Pika Installation: As AMQP is a two-way RPC protocol where the client application or producer can send requests to the server application, and the server can send message to a client. The Pika implements Input Output loops in each of its asynchronous connection adapters and the implemented loops are blocking methods and listen for events.

So the Pika library is required along with RabbitMQ and the following command can be used to install pika:

 pip install pika

At this stage, all the required softwires have been installed for RabbitMQ.

*Start the RabbitMQ server*

The RabbitMQ server must be started after the successful installation. In Windows machine, we can locate the RabbitMQ server application from the start button as shown in the below screenshot:

Click on the "RabbitMQ Service - start" menu to start the RabbitMQ server.

Machine generated alternative text:
s 
RabbitMQ Server 
RabbitMQ Command Prompt (sbin... 
RabbitMQ Database Directory 
RabbitMQ Logs 
RabbitMQ Plugins 
RabbitMQ Service - (re)install 
RabbitMQ Service 
- remove 
RabbitMQ Service - start 
RabbitMQ Service - stop 
RStudio 
Settings 
Snip & Sketch 
Spotify 
Sticky Notes 
TeXstudio 
p ype here to search 

It will launch the command line window, run the start command and show the below screen if the installation is successful:

Machine generated alternative text:
Administrator: RabbitMQ Service - start 
The requested service has already been started. 
Vlore help is available by typing NET HELPMSG 2182. 
: \Windows \system32> 

All good in this stage to move to python implementation for the server as well as worker applications.

*Minimum code in server application*

import pika

SHARED\_REQUEST\_QUEUE = 'SHARED\_REQUEST\_CLASSIFICATION\_MODEL\_QUEUE';

requestConnectionParameters = pika.ConnectionParameters(host='localhost');

requestConnection = pika.BlockingConnection(requestConnectionParameters);

requestChannel = requestConnection.channel();

requestChannel.queue\_declare(queue=SHARED\_REQUEST\_QUEUE, durable=True);

requestChannel.basic\_publish(exchange='', routing\_key=SHARED\_REQUEST\_QUEUE, body= <<message detail>>,

properties=pika.BasicProperties(delivery\_mode=2));

The minimum service application code requires pika to make communication with queue. The shared queue name refers a name of the queue that will be used by the server (to enqueue the task ) as well as worker applications (to pick and process the task). The ConnectionParameters() method of pika takes the localhost as host parameter and creates BlockingConnection instance. The subsequent line create channel object to create/open the queue. The function "queue\_declare" takes queue name and Boolean flag durable as parameters. The durable prarameter with True value ensure the messages in the queue will be safe if there is a queue crash. Now, server application is ready with all the required setup for accepting client applications request. Once it receives any request, channel object's basic\_publish() function is called with empty exchange, routing key (queue name has been used as key here) , body parameter carries the message details. The basic property's delivery mode value 2 mark the messages as persisted.

*Minimum code in client(worker) application*

import pika

SHARED\_REQUEST\_QUEUE = 'SHARED\_REQUEST\_CLASSIFICATION\_MODEL\_QUEUE';

requestConnectionParameters = pika.ConnectionParameters(host='localhost');

requestConnection = pika.BlockingConnection(requestConnectionParameters);

requestChannel = requestConnection.channel();

requestChannel.queue\_declare(queue=SHARED\_REQUEST\_QUEUE, durable=True);

def requestCallback(ch, method, properties, body):

#request hadling code

requestChannel.basic\_qos(prefetch\_count=1)

requestChannel.basic\_consume(queue=SHARED\_REQUEST\_QUEUE, on\_message\_callback=requestCallback)

requestChannel.start\_consuming()

The worker application requires Pika python client to communicate with RabbitMQ. Further, request queue "SHARED\_REQUEST\_CLASSIFICATION\_MODEL\_QUEUE" name is decided. The same queue name must be used by server application to enqueue the messages. The ConnectionParameters() method has been used to specify all the required parameters to establish the connection. The object of connection parameter has been used to create the blocking connection object to wait for the expected response.

The next step is about creating the channel object and this object has been used to interact with RabbitMQ. The channel() method creates the channel object which is used to create/open the queue. The "durable=True" is passed to ensure the messages are safe and restart the RabbitMQ if there is any crash in the queue level. In "basic\_qos" function, qos refers the quality of service and it helps to fetch the message in advance just before finishing the current task. The sample code shows the prefetch count 1.

The "basic\_consume" method consume the messages from the server. We have passed the queue name as well as call back function to process the request. Finally, "start\_consuming" method start/consume the incoming messages.

*Server application (model\_deployment\_rabbitMQ\_server.py)*

TODO: copy Full code

The server application imports the required packages. We have used FastAPI in the server to expose REST API and pika for RabbitMQ communication and JSON for message construction with JSON format. Other than the Pika libraries communication related logic, there are four python methods has been added. In which, one is just a helper function and other three are linked to the REST APIs.

The first function getJSONRequest(), takes job id and prediction request as parameter to construct the JSON object. So Other functions will call getJSONRequest() function to create JSON object for the JOB request. Other three, REST API methods are explained as follows:

(1) prediction\_result - This API will be called by the client application to know the status of the submitted job. The function takes job id as parameter and returns the prediction result from the dictionary if it exists. If the job id is unknown, it returns "Unknown Job Id..." to the caller.

(ii) updated\_predicted\_result - This API is being used by the worker application to update the prediction result. Once the worker application completes the request processing, it notify the server application with prediction request by calling this REST end point.

(iii)predict\_class - This API linked method triggered by the client application with the prediction request. Once the prediction request is submitted,(a) new Job Id will be created for the newly received request, (b)construct the JSON object for the request, (c) submit the job request into the queue and (d) return the created job id to the caller.

Worker application (model\_deployment\_rabbitMQ\_worker.py)

TODO: copy Full code

The worker application uses joblib library to load the prediction model from the disk. There are three user defined python functions have been added in worker application:

(i) getPredictionJSONResponse - takes job id and prediction result as parameters, create json object for the parameters and return the created JSON object to the caller.

(ii) getPredictionRequest - Extract the Job Id and prediction parameter and return the extracted values to the caller.

(iii) requestCallback - Call back function used to process the prediction request and notify the server application with prediction result. As the worker application doesn't interact with any external resources, five seconds manual delay has been introduced.

- Steps to test the server/client applications in localhost

As we have two major application components, client and server, the below steps to be followed carefully to run the components:

(i) make sure RabbitMQ is running

As stated earlier, we can start the rabbit queue from the start menu and ensure RabbitMQ is active.

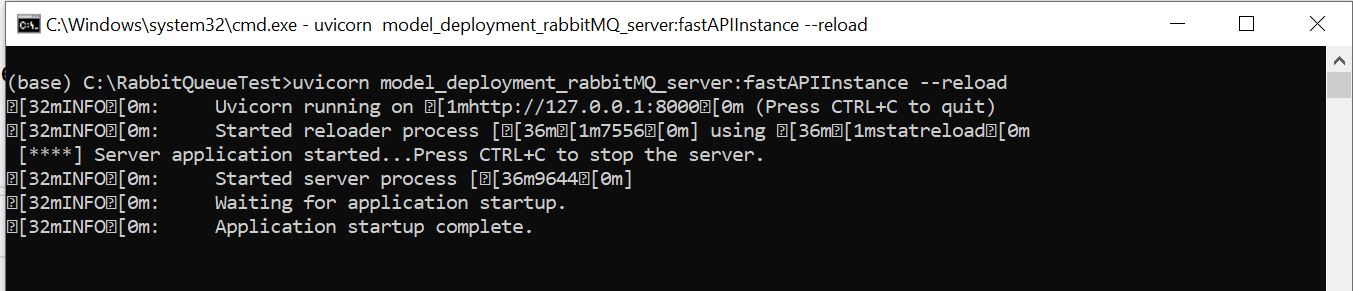
(ii) Copy the following files to a single folder for convenient:

Machine generated alternative text:
> This-PC > BOOTCAMP 
Name 
Iris Classifer.jbl 
RabbitQueueTest 
Date modified 
model deployment rabbitMQ server 
model deployment rabbitMQ worker 
17/4/2021 
21/4/2021 
21/4/2021 
3:13 pm 
8:47 pm 
8:45 pm 
Type 
JBL File 
Python File 
Python File 
Size 
4 KB 
4 KB 

(iii) run server application:

Switch to the command prompt, locate the folder where server application code is saved and run( only one) server application using the below command:

 uvicorn model\_deployment\_rabbitMQ\_server:fastAPIInstance --reload



(iii) run 3 worker application

Launch 3 instance of command prompt to run 3 worker applications. Locate the folder where worker application script is saved and use the below command to run the worker applications one by one:

python model\_deployment\_rabbitMQ\_worker.py

Machine generated alternative text:
C:\Windows\system32\cmd.exe - python model_deployment_rabbitMQ_worker.py 
(base) C: py 
Waiting for messages. To exit press CTRL+C 
C:\Windows\system32\cmd.exe - python model_deployment_rabbitMQ_worker.py 
:(base) C: py 
Waiting for messages. To exit press CTRL+C 
C:\Windows\system32\cmd.exe python model deployment rabbitMQ worker.py 
(base) C: py 
Waiting for messages. To exit press CTRL+C 

(iv) submit the prediction request

The following 5 (http)prediction request are going to be submitted using web browser without any major delay:

<http://127.0.0.1:8000/predict-class?sepal_length=5.3&sepal_width=3.7&petal_length=1.5&petal_width=0.2>

<http://127.0.0.1:8000/predict-class?sepal_length=5.0&sepal_width=3.3&petal_length=1.4&petal_width=0.2>

<http://127.0.0.1:8000/predict-class?sepal_length=5.1&sepal_width=2.5&petal_length=3.0&petal_width=1.1>

<http://127.0.0.1:8000/predict-class?sepal_length=5.7&sepal_width=2.8&petal_length=4.1&petal_width=1.3>

<http://127.0.0.1:8000/predict-class?sepal_length=5.9&sepal_width=3.0&petal_length=5.1&petal_width=1.8>

Launch the web browser, copy the above first http request, paste it on the address bar in the browser and enter. Upon pressing the enter, it will show the below response without any delay:

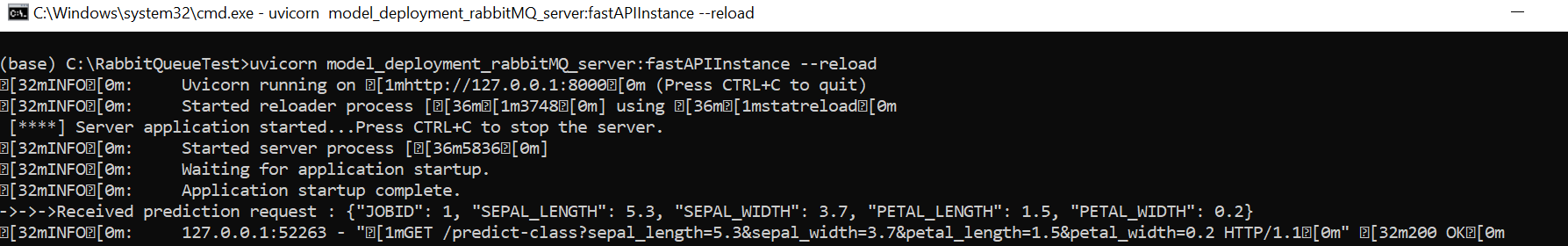
Machine generated alternative text:
127.0.0.1 x + 
C 
127.0.0.1 :8000/predict-class?sepal_length=5.3&sepal 1.5&petal_width=0.2 
5.3, : 
3.7, 

The job id is essential to query the prediction status second time. Now, open another web browser tab and submit the second prediction request:

Continue the same for rest of the 3 requests.

As and when the new request is submitted, server application will print the request detail along with job id and submit the job details to the queue for processing:

Server applications prints that prediction request is received for processing with additional info:



Similarly, worker application will pick the next available job from the queue, print the currently processing job detail and respond to the server application with the prediction result:

Machine generated alternative text:
C:\Windows\system32\cmd.exe python model deployment rabbitMQ worker.py 
(base) C: py 
Waiting for messages. To exit press CTRL+C 
<-<-<- Received '{"JOBID": 1, "SEPAL_LENGTH": 5.3, "SEPAL_WIDTH": 3.7, 
[ [5.3, 3.7, 1.5, e. 2]] 
prediction result 
['Iris-setosa'] 
{"JOBID": 1, "PREDICTION RESULT • 
"Iris-setosa"} 
Updated result successfully. status : {Status : Success} 
"PETAL LENGTH": 
1.5, 
"PETAL WIDTH": e 2} 
Processed the request for Job Id 1. Worker is ready to pick the next task. 

Now we can query the prediction result using the below command:

<http://127.0.0.1:8000/prediction-result?job_id=1>

The job id must be passed to query the result from the server. Upon pasting the above http request on the web browser and enter, it shows the below result:

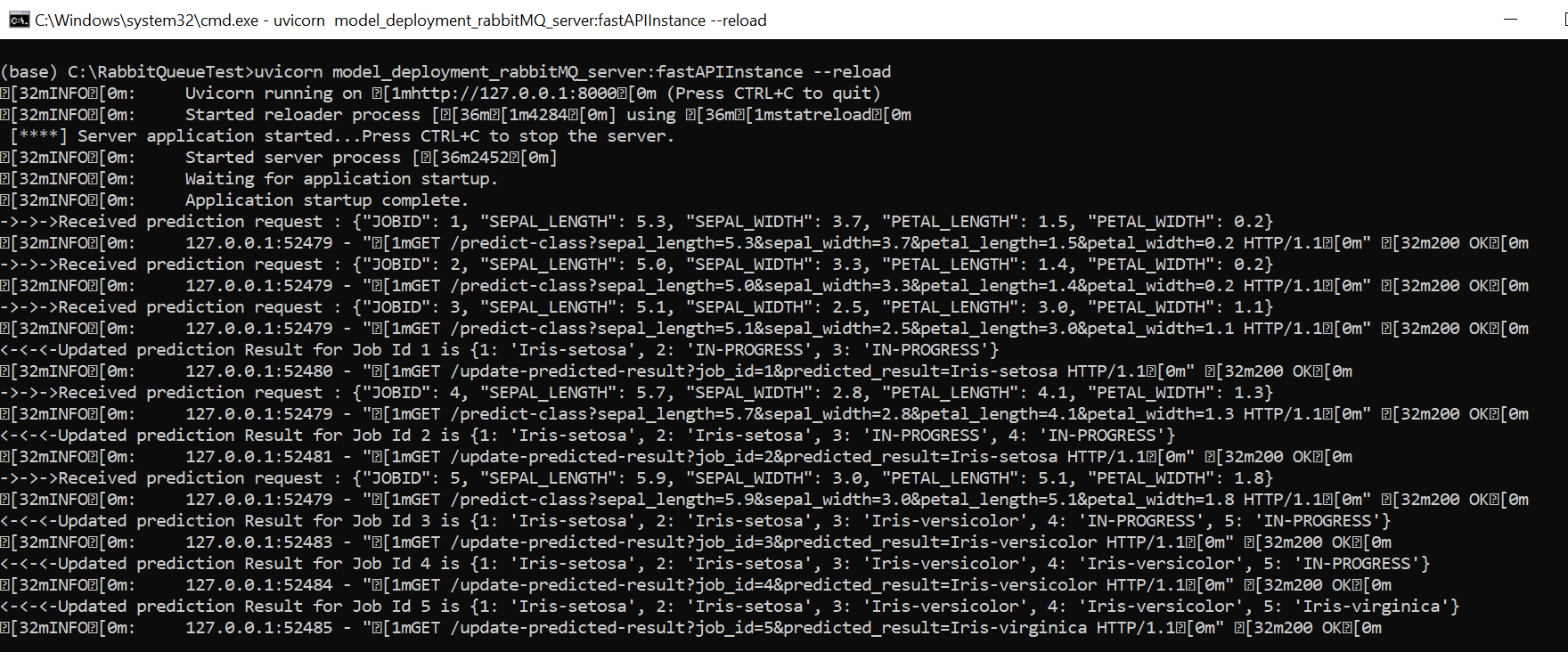
Machine generated alternative text:
127.0.0.1 :8000/prediction-result? x + 
C 0 127.0.0.1 :8000/prediction-result?job id=l 
" Iris- setosa " 

In case, the requested result for the job id is being processed, it will show the in-progress status as result. Below, screenshot shows the in-progress status as the prediction result is not yet ready for the job id 5:

Machine generated alternative text:
127.0.0.1 :8000/prediction-result? x + 
C 0 127.0.0.1 :8000/prediction-result?job id=5 
"IN-PROGRESS" 

Upon processing all the five requests, server and worker applications screen shows the processed tasks' status as follows:

Server application's screen:



Worker 1:

Machine generated alternative text:
C:\Windows\system32\cmd.exe python model deployment rabbitMQ worker.py 
(base) C: py 
Waiting for messages. To exit press CTRL+C 
Received '{"JOBID": 2, "SEPAL_LENGTH": 5.e, "SEPAL_WIDTH": 3.3, 
[ [5.e, 3.3, 1.4, e. 2]] 
•rediction result 
['Iris-setosa'] 
["JOBID": 2, "PREDICTION RESULT • 
"Iris-setosa"} 
Jpdated result successfully. status : {Status : Success} 
"PETAL LENGTH": 
1.4, 
5.1, 
"PETAL WIDTH": e 2} 
"PETAL WIDTH": 1 8} 
Processed the request for Job Id 2. Worker is ready to pick the next task. 
Received '{"JOBID": 5, "SEPAL_LENGTH": 5.9, "SEPAL_WIDTH": 3.e, "PETAL_LENGTH": 
[ [5.9, 3.e, 5.1, 1.8]] 
•rediction result 
[ ' Iris-virginica' ] 
["JOBID": 5, "PREDICTION RESULT 
"Iris-virginica"} 
Jpdated result successfully. status : {Status : Success} 
Processed the request for Job Id 5. Worker is ready to pick the next task. 

Worker 2:

Machine generated alternative text:
C:\Windows\system32\cmd.exe python model deployment rabbitMQ worker.py 
Waiting for messages. To exit press CTRL+C 
Received '{"JOBID": 1, "SEPAL_LENGTH": 5.3, "SEPAL_WIDTH": 
[ [5.3, 3.7, 1.5, e. 2]] 
prediction result 
'Iris-setosa'] 
{"JOBID": 1, "PREDICTION RESULT • 
"Iris-setosa"} 
Updated result successfully. status : {Status : Success} 
3.7, 
"PETAL LENGTH": 
1.5, 
4.1, 
"PETAL WIDTH": e 2} 
"PETAL WIDTH": 1 3} 
Processed the request for Job Id 1. Worker is ready to pick the next task. 
<-<-<- Received '{"JOBID": 4, "SEPAL_LENGTH": 5.7, "SEPAL_WIDTH": 2.8, "PETAL_LENGTH": 
[ [5.7, 2.8, 4.1, 1.3]] 
prediction result 
' Iris-versicolor' ] 
{"JOBID": 4, "PREDICTION RESULT 
"Iris-versicolor"} 
Updated result successfully. status : {Status : Success} 
Processed the request for Job Id 4. Worker is ready to pick the next task. 

Worker 3:

Machine generated alternative text:
C:\Windows\system32\cmd.exe python model deployment rabbitMQ worker.py 
(base) C: py 
Waiting for messages. To exit press CTRL+C 
<-<-<- Received '{"JOBID": 3, "SEPAL_LENGTH": 5.1, "SEPAL_WIDTH": 2.5, 
[ [5.1, 2.5, 3.e, 1.1]] 
prediction result 
' Iris-versicolor' ] 
{"JOBID": 3, "PREDICTION RESULT 
"Iris-versicolor"} 
Updated result successfully. status : {Status : Success} 
"PETAL LENGTH": 
3.e, 
"PETAL WIDTH": 1 1} 
Processed the request for Job Id 3. Worker is ready to pick the next task. 

The conclusion is, the queue-based ML model can be deployed if we need to achieve the asynchronous prediction processing behaviour. As explained earlier, one server application instance will just listen to the clients’ prediction request and send back the job Id immediately. So, we are not waiting for the request to be processed fully. Further, the job id helps to uniquely differentiate the request and reply back to the caller with correct prediction result once it’s available.

There are many worker applications will be running to process multiple prediction request in parallel, but one request by each worker at a time. Each worker application will check the queue for any available job to process and pick the next available/waiting job, process the same and notify the prediction result along with job is once it’s ready. Based on the request volume, worker applications’ count will be decided.