Server-based Machine Learning Model Deployment

Introduction

Deploying a machine learning model, known as model deployment, simply means as integrating a machine learning model into an existing production environment where it can take in an input and return an output. The purpose of the model deployment is to make the predictions from a trained machine learning model available to others. After training a model and evaluating it on the test set, it can be served in a format where it can be used by others when needed.

In this lab, you will be guided to turn your trained machine learning models into a web application where other users can interact with. In the last part of this guide, the designated homework, which you are required to turn in, is explained.

Objectives

The students are expected to learn how to deploy their machine learning models into production where other users can accessed it.

Background

Machine learning models can be deployed in the server or on-device. In server-based deployment, there are three general ways to deploy the machine learning model: (1) one-off, (2) batch and (3) real-time. One-off deployment doesn't need to continuously train a machine learning model, but instead it can be trained once and pushed to production until its performance deteriorates enough to need some fixing. Batch training allows to constantly have an up-to-date version of the model. Lastly, real-time training is possible with "online machine learning" where sequentially available data are used to update the best predictor for future data.

In order for others to make use of the trained machine learning model, it should be accessed online through the Web. With this, we will have some background on web framework first. A web framework is a library of code that enables easier and rapid web application development and maintains the applications over time. The content that we interact on the web is organized in webpages where webpage is a document that can be displayed in a web browser. And to access these webpages, a website should be visited by typing its corresponding URL or address. So, a browser like Google Chrome, Mozilla Firefox and Microsoft Edge are used to interact with a website as a client.

For a client to retrieve information from the web, it needs a web server. A web server is a computer software that processes clients request and sends back a response through the internet. The client and the server communicate with each other using the Hypertext Protocol (HTTP protocol). In todays many applications, a dynamic server is more common where it contains a static web server and maintains an application server that can interact with other servers as shown in Figure 1. The application server updates the files before sending the response to the client. So, whenever a client makes an HTTP request to the web server, through the browser, the web server handles the request and runs the web application through an application server and then, the web server will send the client the response.

There are quite a few web servers available, including Apache HTTP server,

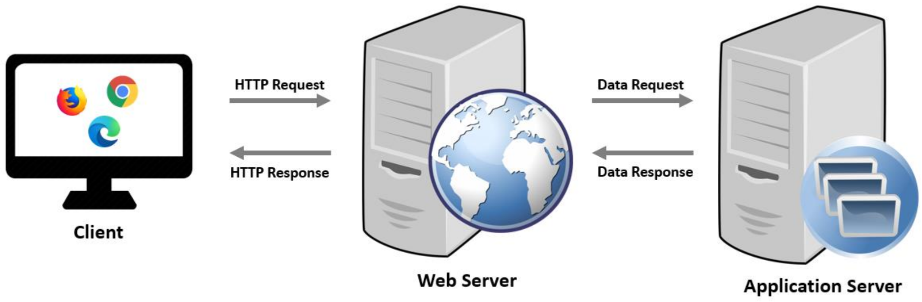
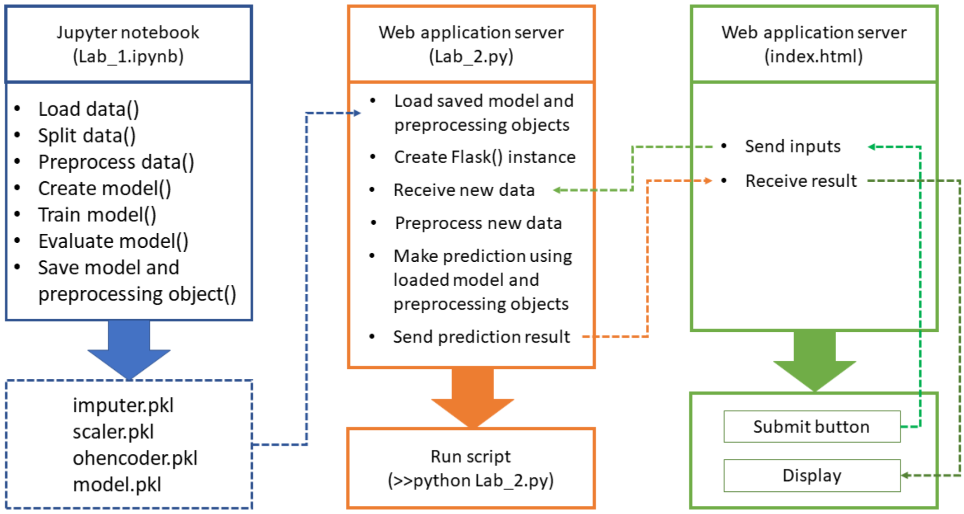


Figure 1. Web Server and Communication Protocol

Microsoft Internet Information Server (IIS) and Nginx. Whereas there are web frameworks that provides a simpler way to leverage Python to build applications that can run on the application server such as Flask and Django. In this lab exercise, Flask will be used.

The flow of deploying the machine learning is presented in the figure below.



After creating a model using Jupyter notebook, saved the trained model and preprocessing objects. In the web application server, the web application script, html template and saved models are stored in the same directory, then, run the web application script to load the saved models and rendered the html template. When the web application script is running, it will wait for the html template to send new data (which inputted by the user) and return the result back which will be displayed on the html template. The interaction of the html template and the web application script is handled by the Flask framework which is created in the web application script.

Implementation: Creating a model

In this section, you will be guided in saving your trained model into a pickle file (.pkl) that will be used in your web application.

Step 1: Prepare your ML model. In this exercise, we will try to deploy the trained model you did in your Lab assignment 1 which a linear regression that predicts the house prices. The dataset can be found in here. You can do this by running the following scripts, which is also available as a Jupyter notebook.

First, importing all the necessary libraries like *numpy* and *pandas* to manipulate and data loading, *sklearn.model_selection* to split the data to training and testing, *sklearn.linear_model* to train the model using the Linear Regression and *sklearn.preprocessing* for data preprocessing. The *sklearn.metrics* are used for model evaluation And lastly, the *pickle* is used to save the trained model and fitted preprocessing objects.

```
import numpy as np  # for manipulation
import pandas as pd  # for data loading
import urllib.request  # for data downloading
import tarfile  # for extracting data

from sklearn.model_selection import StratifiedShuffleSplit  # for splitting data
from sklearn.preprocessing import StandardScaler  # for scaling the attributes
from sklearn.preprocessing import OneHotEncoder  # for handling categorical features
from sklearn.impute import SimpleImputer   # for handling missing data
from sklearn.linear_model import LinearRegression  # for creating model
from sklearn.metrics import mean_squared_error, r2_score  # for evaluation

import pickle  # for saving

```

Next, we create a custom class for creating new attribute by combining existing attributes.

```
# Custom class for combined attributes
class CombinedAttributesAdder():
    def fit(self, X, y=None):
        return self
    def transform(self, X):
        rooms_per_household = X[:, rooms_ix] / X[:, households_ix]
        population_per_household = X[:, population_ix] / X[:, households_ix]
        bedrooms_per_room = X[:, bedrooms_ix] / X[:, rooms_ix]

        X = np.delete(X, [households_ix, rooms_ix, population_ix, bedrooms_ix], 1)

        return np.c_[X, rooms_per_household, population_per_household, bedrooms_per_room]

```

Then, the preprocessing class which handles the missing data, combining attributes, scaling the features, handling the categorical input feature and concatenating all the features. This class contains *fit* and *transform* functions in which *fit* function is used during training while the *transform* function is used during evaluating on test data. In *fit* function, it fit and transform the data whereas in *transform* function, it just transform the data using the fitted preprocessing object. Saving the fitted preprocessing objects are included which is used later in deploying together with the trained model.

```
        # scale the features
        self.stdscale.fit(housing_addtl_attr)
        X_train_imp_scaled = self.stdscale.transform(housing_addtl_attr)

        # handle categorical input feature
        self.ohe.fit(house_cat)
        X_train_ohe = self.ohe.transform(house_cat)

        # concatenate features
        X_train = np.concatenate([X_train_imp_scaled, X_train_ohe], axis=1)

        return X_train

    def transform(self, X):
        # transform the test data (use the fitted imputer,
        #                         standardscaler, onehotencoder,
        #                         combinedattribute from training)
        house_num = X.drop("ocean_proximity", axis=1)
        house_cat = X[["ocean_proximity"]]

        # handle missing data
        X_test_imp = self.imputer.transform(house_num)
        X_test_imp = pd.DataFrame(X_test_imp, columns=house_num.columns, index=X.index)

        # combined attributes
        housing_addtl_attr = self.attr_add.transform(X_test_imp.values)

        # scale the features
        X_test_imp_scaled = self.stdscale.transform(housing_addtl_attr)

        # handle categorical input feature
        X_test_ohe = self.ohe.transform(house_cat)

        # concatenate features
        X_test = np.concatenate([X_test_imp_scaled, X_test_ohe], axis=1)

        return X_test

    def savefittedobject(self):
        pickle.dump(self.imputer, open('houseimputer.pkl', 'wb'))
        pickle.dump(self.stdscale, open('housescaler.pkl', 'wb'))
        pickle.dump(self.ohe, open('houseohencoder.pkl', 'wb'))

```

Then, loading the dataset and splitting into training and testing.

```python
# load the dataset
url = "https://raw.githubusercontent.com/ageron/handson-ml2/master/datasets/housing/housing.tgz"
urllib.request.urlretrieve(url, "housing.tgz")
tar = tarfile.open("housing.tgz")
tar.extractall()
tar.close()
housing = pd.read_csv("housing.csv")

# split the data
housing["income_cat"] = pd.cut(housing["median_income"],
                               bins=[0., 1.5, 3.0, 4.5, 6., np.inf],
                               labels=[1, 2, 3, 4, 5])
split = StratifiedShuffleSplit(test_size=0.2, random_state=42)
for train_index, test_index in split.split(housing, housing["income_cat"]):
    strat_train_set = housing.loc[train_index]
    strat_test_set = housing.loc[test_index]

# assign the training data
train_housing = strat_train_set.drop("median_house_value", axis=1)
train_housing_labels = strat_train_set["median_house_value"].copy()
```

Then, preprocessing the training dataset by calling the *data_preprocessing* class.

```python
# get the column indices to be used in getting additional attributes
col_names = ["total_rooms", "total_bedrooms", "population", "households"]
rooms_ix, bedrooms_ix, population_ix, households_ix = [
    train_housing.columns.get_loc(c) for c in col_names] # get the column indices

# preprocess the training data
house_preprocess = data_preprocessing()
data_X_train = house_preprocess.fit(train_housing)
```

Then, we can instantiate a regressor using *LinearRegression()*, train and evaluate it using the training and test data, respectively.

```
# create the model
lin_reg = LinearRegression()

# train the model
lin_reg.fit(data_X_train, train_housing_labels)

# evaluate the model on training dataset
housing_predictions = lin_reg.predict(data_X_train)
lin_mse = mean_squared_error(train_housing_labels, housing_predictions)
lin_r2 = r2_score(train_housing_labels, housing_predictions)
print("Performance for Train dataset: ", lin_mse, np.sqrt(lin_mse), lin_r2)

# assign the test data
X_test = strat_test_set.drop("median_house_value", axis=1)
y_test = strat_test_set["median_house_value"].copy()

# preprocess the test data
data_X_test = house_preprocess.transform(X_test)

# test the trained model on test data
final_predictions = lin_reg.predict(data_X_test)

# evaluate the model on test dataset
final_mse = mean_squared_error(y_test, final_predictions)
print("Final performance evaluation: ", final_mse, np.sqrt(final_mse),
      lin_reg.score(data_X_test, y_test))
```

Step 2: Save the trained model using the pickle's *dump()* method into a file specified in the argument. This saved model will be used in the web application.

```
# saving model as pickle file
pickle.dump(lin_reg, open('houseregressionmodel.pkl', 'wb'))
```

Step 3: Save the fitted preprocessing objects by using the pickle's *dump()* method into a file. In this script, saving the fitted objects are defined in the *data_processing* class. The saved fitted preprocessing objects will also be used in the web application for preprocessing new data.

```
# saving imputer, scaler and onehotencoder
house_preprocess.savefittedobject()
```

```
def savefittedobject(self):
    pickle.dump(self.imputer, open('houseimputer.pkl', 'wb'))
    pickle.dump(self.stdscale, open('housescaler.pkl', 'wb'))
    pickle.dump(self.ohe, open('houseohencoder.pkl', 'wb'))
```

Implementation: Creating a web app with Flask

There are several things that are needed to put together for creating a web app. The first two are:

1. The Python code that will load the saved model and fitted preprocessing objects, get the user's input from a web form, make prediction and then return the result.

2. The HTML templates that the Flask will render. These allow a user to input their own data and will present the corresponding result.

The web app will be structured as follows:

```
HousePredictionWebApp/
├── houseregressionmodel.pkl
├── houseimputer.pkl
├── housescaler.pkl
├── houseencoder.pkl
├── Templates
│   └── index.html
└── oneoff_app.py
```

oneoff_app.py

In this section, we'll prepare the *oneoff_app.py* file. The *oneoff_app.py* is the core of the web application where it processes inputs from the user. (You can see the full notebook in here)

Step 1: Import the Flask module, where Flask is a Python framework for building web apps, and all the necessary libraries.

```
import numpy as np  # for manipulation
import pandas as pd  # for data loading

from sklearn.preprocessing import StandardScaler  # for scaling the attributes
from sklearn.preprocessing import OneHotEncoder  # for handling categorical features
from

impor

from                                                          service
```

```
# model and fitted object loading
model = pickle.load(open('houseregressionmodel.pkl', 'rb'))
imputer = pickle.load(open('houseimputer.pkl', 'rb'))
scaler = pickle.load(open('housescaler.pkl', 'rb'))
ohencoder = pickle.load(open('houseohencoder.pkl', 'rb'))
```

Step 2: Load the model and the fitted preprocessing objects at the top of the app in order for it to load into memory once rather than being loaded every time a prediction will be made.

Step 3: Create a *Flask()* instance, then, different functions can be written. In flask, URLs get routed to different functions.

```
# Flask instantiation
app = Flask(__name__, template_folder='templates')
```

<u>Step 4</u>: Include the classes for preprocessing the data. Notice that, in the *data_preprocessing* class, only the *initialization* and *transform* functions are defined as we only test incoming data. No retraining is done in the trained model in one-off deployment.

```
# Custom class for combined attributes
class CombinedAttributesAdder():
    def fit(self, X, y=None):
        return self
    def transform(self, X, rooms_ix, bedrooms_ix, population_ix, households_ix):
        rooms_per_household = X[:, rooms_ix] / X[:, households_ix]
        population_per_household = X[:, population_ix] / X[:, households_ix]
        bedrooms_per_room = X[:, bedrooms_ix] / X[:, rooms_ix]

        X = np.delete(X, [households_ix, rooms_ix, population_ix, bedrooms_ix], 1)

        return np.c_[X, rooms_per_household, population_per_household, bedrooms_per_room]

# class for data preprocessing
class data_preprocessing():
    def __init__(self):
        self.imputer = imputer
        self.attr_add = CombinedAttributesAdder()
        self.stdscale = scaler
        self.ohe = ohencoder

    def transform(self, X, rooms_ix, bedrooms_ix, population_ix, households_ix):
        # transform the test data (use the fitted imputer,
        #                          standardscaler, onehotencoder,
        #                          combinedattribute from training)
        house_num = X.drop("ocean_proximity", axis=1)
        house_cat = X[["ocean_proximity"]]

        # handle missing data
        X_test_imp = self.imputer.transform(house_num)
        X_test_imp = pd.DataFrame(X_test_imp, columns=house_num.columns, index=X.index)

        # combined attributes
        housing_addtl_attr = self.attr_add.transform(X_test_imp.values, rooms_ix,
                                             bedrooms_ix, population_ix, households_ix)

        # scale the features
        X_test_imp_scaled = self.stdscale.transform(housing_addtl_attr)

        # handle categorical input feature
        X_test_ohe = self.ohe.transform(house_cat)

        # concatenate features
        X_test = np.concatenate([X_test_imp_scaled, X_test_ohe], axis=1)

        return X_test
```

Step 5: Define the base function where this function handles all the requests from clients to do the prediction given all the inputs. All the requests will be routed on this function. This web app will run in two modes, first, it will just display the input form to the user and the other one will retrieve the user's input. This uses two different HTTP methods: (1) *GET* and (2) *POST*. In lines 74-82, it gets the inputs from the client and convert it to float for further processing. In lines 84-88, it detects empty input from the user which will be treated as missing data (in this exercise, just to include missing data condition, only the *total_bedroom* can be missed, all remaining attributes are

```
67
68  @app.route('/', methods=['GET', 'POST'])
69  def index():
70      if request.method == 'GET':
71          return(render_template('index.html'))
72      if request.method == 'POST':
73          # get input values
74          longitude = float(request.form['longitude'])
75          latitude = float(request.form['latitude'])
76          housingmedianage = float(request.form['housingmedianage'])
77          totalrooms = float(request.form['totalrooms'])
78          totalbedrooms = request.form['totalbedrooms']
79          population = float(request.form['population'])
80          households = float(request.form['households'])
81          medianincome = float(request.form['medianincome'])
82          oceanproximity = request.form['oceanproximity']
83
84          # handle missing input in total_bedrooms attribute
85          if totalbedrooms == '':
86              totalbedrooms = float('nan')
87          else:
88              totalbedrooms = float(totalbedrooms)
89
90          # new category creation by assuming median income is a very important attribute
91          income_cat = pd.cut([medianincome],
92                              bins=[0., 1.5, 3.0, 4.5, 6., np.inf],
93                              labels=[1, 2, 3, 4, 5])
94
95          # convert input data to dataframe
96          inputs_ = {'longitude': longitude,
97                     'latitude': latitude,
98                     'housing_median_age': housingmedianage,
99                     'total_rooms': totalrooms,
100                    'total_bedrooms': totalbedrooms,
101                    'population': population,
102                    'households': households,
103                    'median_income': medianincome,
104                    'ocean_proximity': oceanproximity,
105                    'income_cat': income_cat}
106
107         inputs_df = pd.DataFrame(inputs_)
108
109         # get the column indices to be used in getting additional attributes
110         col_names = ["total_rooms", "total_bedrooms", "population", "households"]
111         rooms_ix, bedrooms_ix, population_ix, households_ix = [
112             inputs_df.columns.get_loc(c) for c in col_names] # get the column indices
113
114         # preprocess the inputs
115         preprocessing_ = data_preprocessing()
116         inputs_preprocessed = preprocessing_.transform(inputs_df, rooms_ix, bedrooms_ix,
117                                                        population_ix, households_ix)
118
119         # predict the price
120         prediction = model.predict(inputs_preprocessed)
121
122         return render_template('index.html', result=prediction[0])
123
```

required to fill in). Before predicting the house price using the trained model in line 120, the inputs are preprocessed first as can be seen in lines 90-117. After predicting, the result is returned as shown in line 122.

Step 6: Define the host address where the web application will be running and accept query from users. This address will serve as the URL of web application (e.g. http:/128.134.65.180:5000, change this IP according to your computer's IP address). If a user loads the main URL for the app, flask will receive a GET request and render the *index.html*. While if the user fills in the form on the page and clicks on the *submit* button, flask receives a POST request, extracts the input, runs it through the model and finally render the *index.html* with the results in place.

```
# running the application for serving
if __name__ == '__main__':
    app.run(host="128.134.65.180")
```

index.html

The *index.html* can be written using a text editor and save as **.html* file. The HTML code for this exercise is shown below or you can access the file from here. The important components in this code is first, the *<form action="{{ url_for('index') }}" method="POST">*, in line 6, the *action* attribute tells flask which *route* (function) should be called when form is submitted, where in this example is the *index()* function. The *POST* method tells the function that it should expect input and therefore process it.

index.html

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4  <title>House Predictor</title>
5  </head>
6  <form action="{{ url_for('index') }}" method="POST">
7      <fieldset>
8          <legend>Input values:</legend>
9          <p>
10             Longitude:
11             <input name="longitude" type="number" step=0.01 required />
12         </p>
13         <p>
14             Latitude:
15             <input name="latitude" type="number" step=0.01 required />
16         </p>
17         <p>
18             Housing median age:
19             <input name="housingmedianage" type="number" step=0.01 required />
20         </p>
21         <p>
22             Total Rooms:
23             <input name="totalrooms" type="number" step=0.01 required />
24         </p>
25         <p>
26             Total Bedrooms:
27             <input name="totalbedrooms" type="number" step=0.01 />
28         </p>
29         <p>
30             Population:
31             <input name="population" type="number" step=0.01 required />
32         </p>
33         <p>
```

```
34                  Households:
35                  <input name="households" type="number" step=0.01 required />
36              </p>
37              <p>
38                  Median Income:
39                  <input name="medianincome" type="number" step=0.0001 required />
40              </p>
41              <p>
42                  Ocean Proximity:
43                  <select name="oceanproximity" required>
44                      <option><1H OCEAN</option>
45                      <option>INLAND</option>
46                      <option>NEAR OCEAN</option>
47                      <option>NEAR BAY</option>
48                      <option>ISLAND </option>
49                  </select>
50              </p>
51
52              <input type="submit" />
53          </fieldset>
54      </form>
55      <br>
56      <div>
57          {% if result %}
58              <br>Predicted Price: <p style="font-size:20px; margin-left:30px;">{{ result }}</p>
59          {% endif %}
60      </div>
61      </html>
```

The *<div>* component, in lines 56-60, handles the result returned by the function. This part of the template uses special syntax to render Python variables. If a result is return *({ % if result % })*, it displays the result.

Implementation: Testing the web application

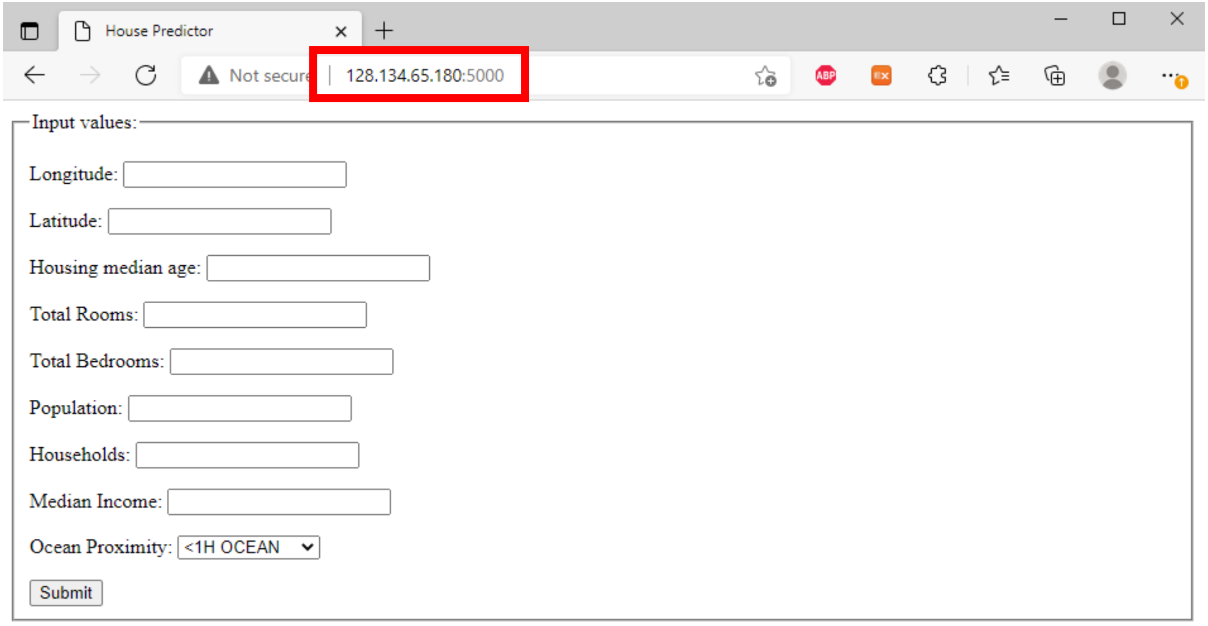
Step 1: Until here, you can test your web app locally by running your *oneoff_app.py* in your anaconda powershell, with correct path of your application.

```
(base) E:\class\7th sem\TA class>python oneoff_app.py
```

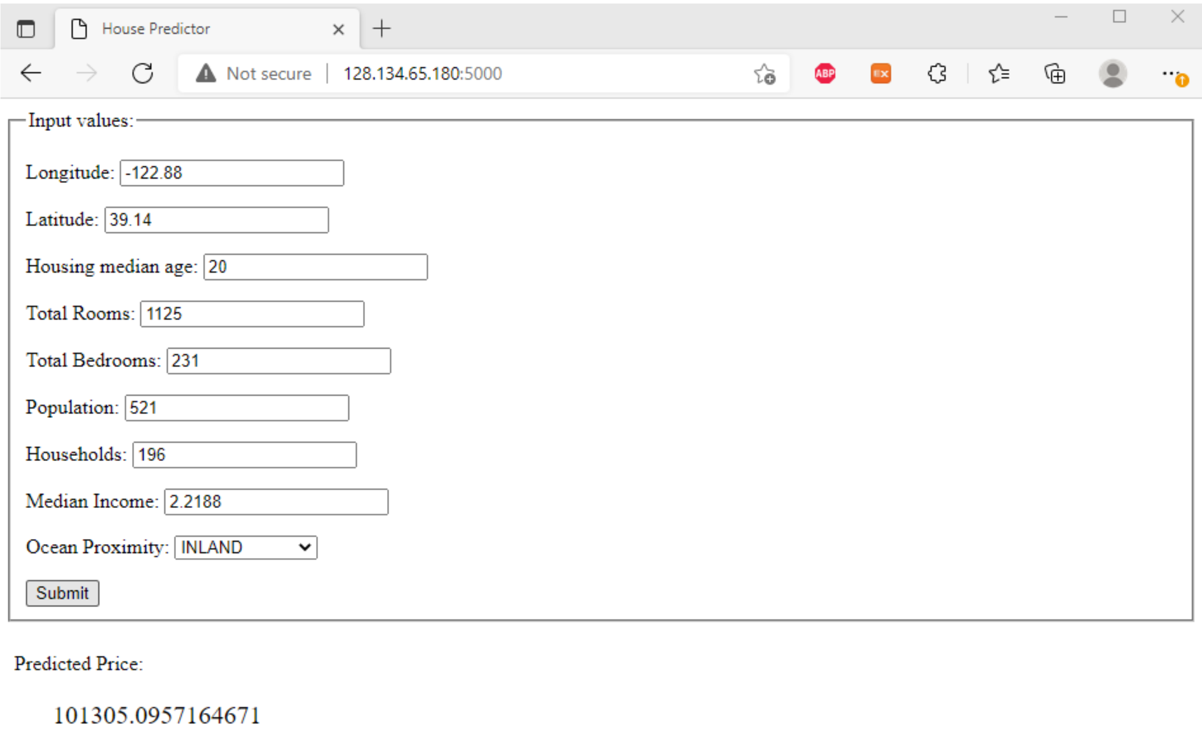
Step 2: After running the command, you will see the local server address displayed like “http:/128.134.65.180:5000”.

```
(base) E:\class\7th sem\TA class>python oneoff_app.py
 * Serving Flask app "oneoff_app" (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: off
 * Running on http://128.134.65.180:5000/ (Press CTRL+C to quit)
```

Step 3: You can copy this address and paste it in your browser which displays the web application.



Step 4: You can test your model by filling up the required inputs. Make sure everything is working fine until here.



Until here, you can access your web application from one machine into another provided that all your machines are in the same network.

Part 2: Batch Inference

Batch inference is the second method in deploying Machine Learning model. For batch inference, it involves two components, (1) the prediction saving, and (2) the schedule of retraining. To do this, an additional code on the *index()* function will be added (you can access the notebook from here). Also, in *data_processing* class, the definition of *fit* function and *savefittedobject* function is included for retraining. In batch training, after predicting the result of the user's input, it is saved in the database (but here, CSV is used for simplicity).

Now that the code is updated to save the inputs and predicted outputs, schedule of retraining should be defined next. In this example, the model will be retrained after more than forty (40) new data. Once this condition is satisfied, the model will be retrained with the new data. *(Note: In real application, the condition may depend on the time zone rather than on the number of new data (e.g., time of the day, day of the week, etc.))*

```
        # predict the price
        prediction = model.predict(inputs_preprocessed)

        # batch training
        # adding the median_house_value in the data for retraining
        inputs_df['median_house_value'] = int(prediction[0])
        # dropping the income_cat attribute before saving
        inputs_df = inputs_df.drop("income_cat", axis=1)
        # saving to csv the new data
        inputs_df.to_csv('housing_data.csv', mode='a', index=False, header=False)
        # retraining
        if len(housing_data) > 40:
            # new category creation by assuming median income is a very important attribute
            housing_data["income_cat"] = pd.cut(housing_data["median_income"],
                                          bins=[0., 1.5, 3.0, 4.5, 6., np.inf],
                                          labels=[1, 2, 3, 4, 5])

            # assign the training data
            train_housing = housing_data.drop("median_house_value", axis=1)
            train_housing_labels = housing_data["median_house_value"].copy()

            # preprocess the training data
            data_X_train = preprocessing_.fit(train_housing,rooms_ix, bedrooms_ix,
                                              population_ix, households_ix)

            # retrain the model
            model.fit(data_X_train, train_housing_labels)

            # save the model
            pickle.dump(model, open('houseregressionmodel_retrain.pkl', 'wb'))

            # save the fitted objects
            preprocessing_.savefittedobject()

        return render_template('index.html', result=prediction[0])
```

Part 3: Realtime Method

In real-time deployment, sequentially available data are used to update the model. In this way, the predictor will be retrained every time there's a user's input and consequently be saved right after as well as the fitted preprocessing objects. You can access the notebook in here.

```
        # predict the price
        prediction = model.predict(inputs_preprocessed)

        # realtime training
        # preprocess the training data
        data_X_train = preprocessing_.fit(inputs_df, rooms_ix, bedrooms_ix,
                                            population_ix, households_ix)

        # retrain the model
        model.fit(data_X_train, [prediction[0]])

        # save the model
        pickle.dump(model, open('houseregressionmodel_retrain.pkl', 'wb'))

        # save the fitted objects
        preprocessing_.savefittedobject()

        return render_template('index.html', result=prediction[0])
```

Homework:

Train a model that classify if a person is diabetic or not using this dataset. Deploy the trained model on a web application for server-based inference.

For this homework, you are required to turn in the following:

a. Python script of implementation of building the model and saving the model.

b. Python script of the web application for (1) one-off deployment, (2) batch inference, (3) real-time method.

c. Screen captures of the web application.

d. Discussion of the implementation procedure and results.