Lab - Machine Learning Model Deployment

Deploying a machine learning, known as model deployment, simply means 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.

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 does not need to continuously train a machine learning model. Instead, it can be trained once and pushed to production until its performance deteriorates enough to need some adjustments. Batch training allows up-to-date version of the model. 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, via a web framework. 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 organised in webpages. A webpage is a document that can be displayed in a web browser. To access these webpages, a website should be visited by typing its corresponding URL or address. Therefore, a browser such as 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, there has to be a web server. A web server is a computer software that processes clients' requests and sends back a response through the Internet. The client and the server communicate with each other using the Hypertext Protocol (HTTP Protocol). In many of today's applications, a dynamic server is more common which contains a static web server and maintains an application server that can interact with other servers as shown in the figure below.

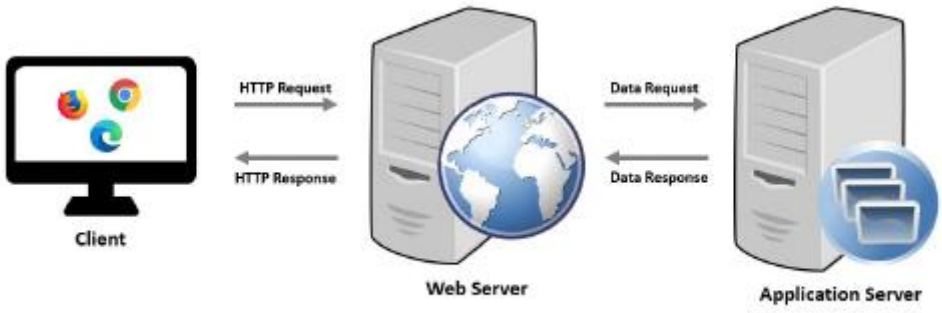
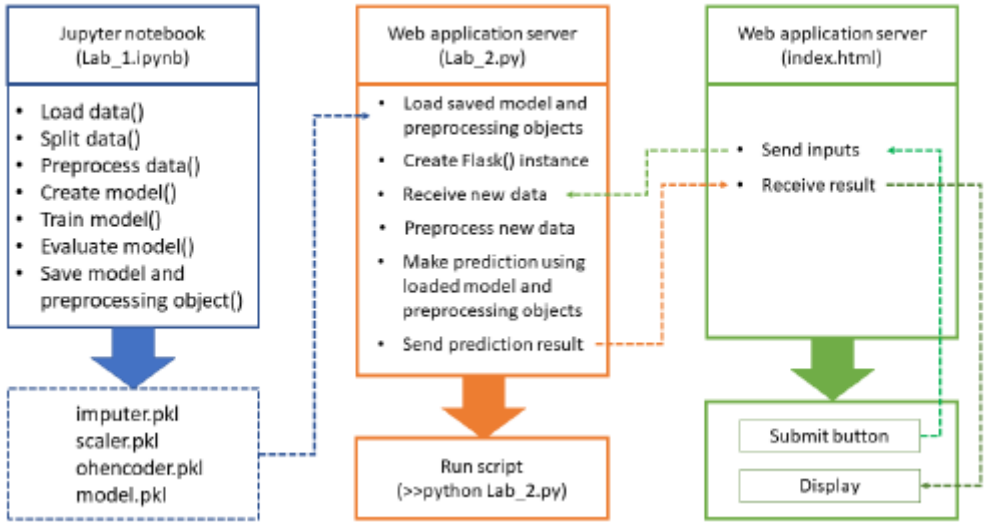


Figure 1. Web Server and Communication Protocol

The application server updates the files before sending the response to the client. Therefore, whenever a client makes a HTTP request to the web server through the browser, the web server handles the request and runs the web application through an application server. The web server will then send the client the response.

There are various web servers available, including Apache HTTP server, Microsoft Internet Information Server (IIS), and Nginx. There are also web frameworks, such as Flask and Django, that provide simpler ways to leverage Python to build applications that can run on the application server. In this lab guide, Flask will be used.

The flow of deploying a machine learning model is presented in the figure below:



After creating a model using Jupyter Notebook, save the trained model and preprocessed objects. In the web application server, the web application script, html template, and saved models are stored in the same directory. Next, run the web application script to load the saved models and render the html template. When the web application script is running, it will wait for the html template to send new data (input by the user) and return the result back to 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.

Creating the model

In this section, we will save our trained model into a pickle file (.pkl) that will be used in our web application.

In this lab guide, we will try to deploy the trained model we did on our 4th assignment (text processing): a classification model that determines if a text message is a spam message or not. You can refer back to the previous lab guide for detailed explanations of the code. The basic code for training the classification model will be shown here. We will use Random Forest classifier as our model of choice. At the end, once the model is trained, we will use "pickle" to save the trained model and the preprocessing objects.

We will be using TfidfVectorizer to create the features.

```
import numpy as np
import pandas as pd
import string
import re
import matplotlib.pyplot as plt
import warnings

from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split

import nltk
import pickle

%matplotlib inline
warnings.filterwarnings('ignore', category=DeprecationWarning)
```

```
data = pd.read_csv('SMSSpamCollection.tsv', sep='\t', names=['label', 'body_text'], header=None)
data.head()
```

	label	body_text
0	ham	I've been searching for the right words to tha...
1	spam	Free entry in 2 a wkly comp to win FA Cup fina...
2	ham	Nah I don't think he goes to usf, he lives aro...
3	ham	Even my brother is not like to speak with me. ...
4	ham	I HAVE A DATE ON SUNDAY WITH WILL!!

```
stopwords = nltk.corpus.stopwords.words('english')
ps = nltk.PorterStemmer()

def clean_text(text):
    text = ''.join([word.lower() for word in text if word not in string.punctuation])
    tokens = re.split('\W+', text)
    text = [ps.stem(word) for word in tokens if word not in stopwords]
    return text

def count_punct(text):
    count = sum([1 for char in text if char in string.punctuation])
    return round(count/(len(text) - text.count(' ')), 3) * 100

# calculate the length of messages excluding spaces
data['body_len'] = data['body_text'].apply(lambda x: len(x) - x.count(' '))
data['punct%'] = data['body_text'].apply(lambda x: count_punct(x))
data.head()

tfidf_vect = TfidfVectorizer(analyzer=clean_text)
tfidf_fit = tfidf_vect.fit(data['body_text'])
X_tfidf = tfidf_fit.transform(data['body_text'])
X_tfidf_feat = pd.concat([data['body_len'], data['punct%'], pd.DataFrame(X_tfidf.toarray())], axis=1)

X_tfidf_feat.head()
```

	body_len	punct%	0	1	2	3	4	5	6	7	...	8097	8098	8099	8100	8101	8102	8103	8104	8105	8106
0	160	2.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	128	4.7	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	49	4.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	62	3.2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4	28	7.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

5 rows × 8109 columns

```
X_train, X_test, y_train, y_test = train_test_split(X_tfidf_feat, data['label'], test_size=0.2)
```

```
rf = RandomForestClassifier(n_estimators=150, max_depth=None)
rf.fit(X_train, y_train)

y_pred = rf.predict(X_test)

print('Prediction accuracy: ', accuracy_score(y_test, y_pred))
```

```
Prediction accuracy:  0.9712746858168761
```

As we can see, this yields a prediction accuracy of above 97%. We will save two things:

1. The trained random forest model
2. The trained TF-IDF model

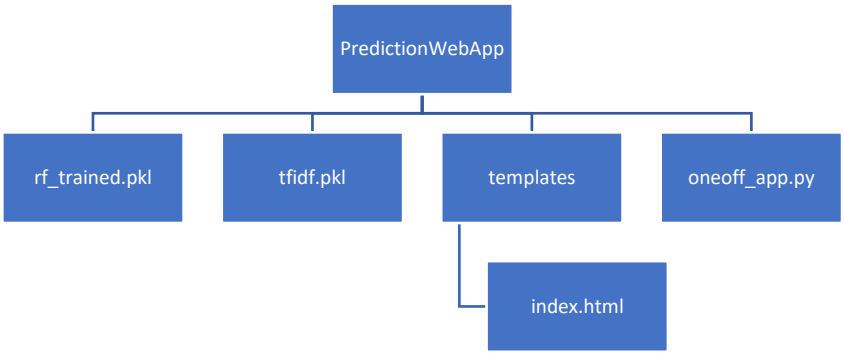
These models will be called back later for implementation.

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:



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. It is not a notebook file, but rather, a separate Python script file.

First, we import the necessary modules. Flask module is a Python framework for building web apps.

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

import string
import re
from sklearn.feature_extraction.text import TfidfVectorizer
import nltk

import pickle  # for importing model

from flask import Flask, request, jsonify, render_template  # for handling web service
```

Next, we will initiate and create functions for preprocessing, such as stopwords and NLTK's Porter Stemmer. Functions for cleaning text, for counting punctuations, and the overall preprocessing will be described first to be initiated.

```
stopwords = nltk.corpus.stopwords.words('english')
ps = nltk.PorterStemmer()

def clean_text(text):
    text = ''.join([word.lower() for word in text if word not in string.punctuation])
    tokens = re.split('\W+', text)
    text = [ps.stem(word) for word in tokens if word not in stopwords]
    return text

def count_punct(text):
    count = sum([1 for char in text if char in string.punctuation])
    return round(count/(len(text) - text.count(' ')), 3) * 100

def preprocessing(data):
    data['body_len'] = data['body_text'].apply(lambda x: len(x) - x.count(' '))
    data['punct%'] = data['body_text'].apply(lambda x: count_punct(x))
    X_tfidf = fitted_tfidf.transform(data['body_text'])
    X_tfidf_feat = pd.concat([data['body_len'], data['punct%'], pd.DataFrame(X_tfidf.toarray())], axis=1)
    return X_tfidf_feat
```

Next, we will load the random forest and TF-IDF models that have been trained.

```
# model and fitted object loading
model = pickle.load(open('rf_trained.pkl', 'rb'))
fitted_tfidf = pickle.load(open('tfidf.pkl', 'rb'))
```

We will then create a Flask() instance, inputting the templates to be used.

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

Next, we will define the base function which handles all the requests from clients to do the prediction given all the inputs. All the requests will be routed on this function. The web app will run in two modes. First, it will display the input form to the user. Second, it will retrieve the user's input. This uses two different HTTP methods: (1) GET and (2) POST. The function below gets the input from the client and further processs it. After preprocessing, the prediction will take place using our trained random forest model, and the result is returned.

```
@app.route('/', methods=['GET', 'POST'])
def index():
    if request.method == 'GET':
        return(render_template('index.html'))
    if request.method == 'POST':
        # get input values
        text_input = (request.form['text_message'])
        text_list = []
        text_list.append(text_input)
        text_df = pd.DataFrame(text_list, columns=['body_text'])
        X_tfidf_feat = preprocessing(text_df)

        # predict the price
        prediction = model.predict(X_tfidf_feat)

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

Next, we have to 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 applications. The address should change 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. 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 render the index.html with the results in place.

```
# running the application for serving

if __name__ == '__main__':
    app.run(host='223.194.33.86')
```

The index.html file that contains the information for the GUI is provided for this lab guide as it will not be the main focus here. If you are familiar with html codes, feel free to change the layout of the file to change the GUI.

Implementation: testing the web application

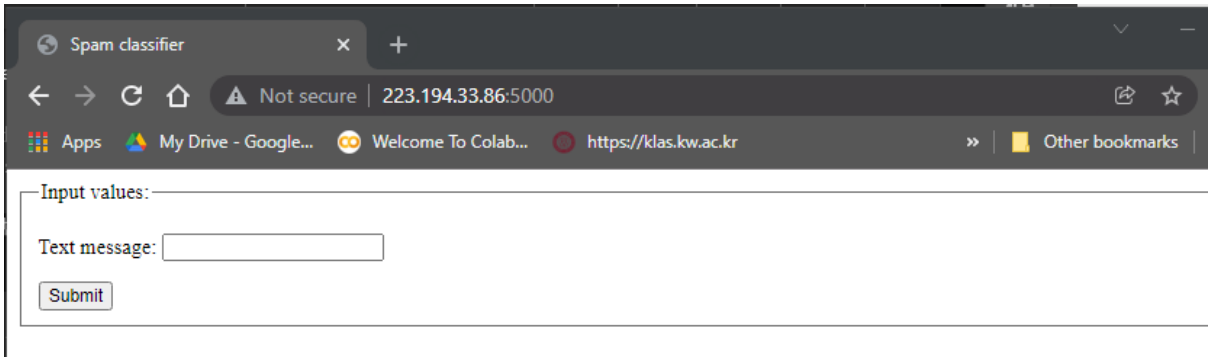
Now, we can test the web app locally by running the oneoff_app.py in the command shell. The command prompt should be in the correct directory, where all the required files are present.

```
(python7) PS C:₩Users₩Brian Jung₩Desktop₩Lab deployment> python oneoff_app.py
```

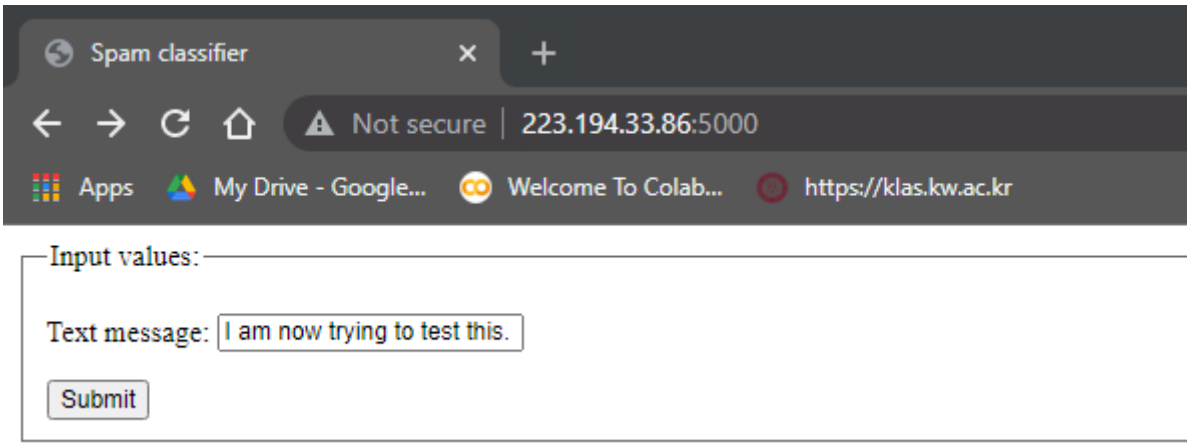
After running this, you will see the local server address displayed:

```
(python7) PS C:₩Users₩Brian Jung₩Desktop₩Lab deployment> 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://223.194.33.86:5000/ (Press CTRL+C to quit)
```

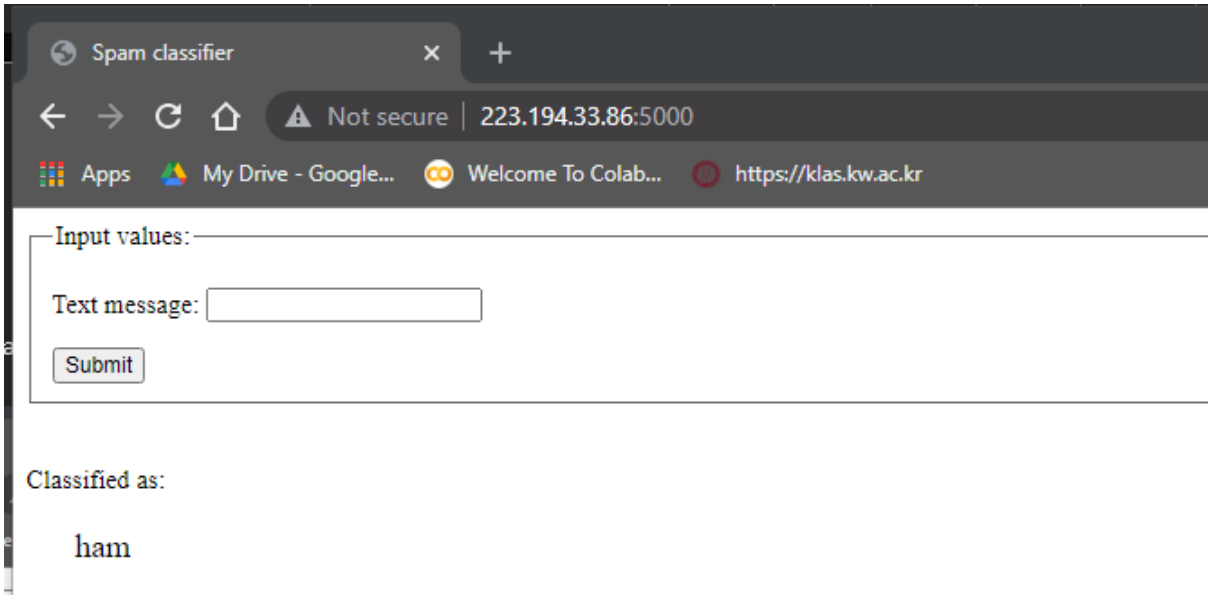
You can copy this address and paste it in your browser, which will display the web application.



You can now test your model by entering some text in the box. The text has to be in English.



Once you press submit, the result, ‘spam’ or ‘ham’ will come out.



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

Assignment

1. Train an SVM model with the same data and deploy it using the methods described above.
2. Train a Naïve Bayes model with the same data and deploy it using the methods described above.
3. Train a K-Nearest Neighbour model with the same data and deploy it using the methods described above.