

Deploying a Model with Flask

 <https://github.com/learn-co-curriculum/dsc-flask-deployment>  <https://github.com/learn-co-curriculum/dsc-flask-deployment/issues/new>

Introduction

Now that you have learned the basics of the Flask web framework, you will combine that knowledge with your prior knowledge of cloud functions to deploy a machine learning model as an HTTP API with Flask!

Clone this repository and work locally so that you can run and test your Flask app. Start by running `jupyter notebook` so that you can run the code examples in this notebook.

Objectives

In this lesson you will:

- Recall the model pickling and unpickling process from the cloud function approach
- Incorporate a model prediction function into a Flask web app
- Deploy a machine learning model as an HTTP API using Flask and Heroku

Recall: Cloud Functions

In a previous lesson, you were introduced to cloud functions. With a cloud function, you need:

1. A pickled model file
2. A Python file defining the function
3. A requirements file

We will reuse the model file and Python code from the previous cloud functions lesson, so you may want to go back and review that lesson if you're confused about any of the details.

The model file has already been included in this repository as `model.pkl` :

```
! ls
```

We'll also be reusing this code from the cloud function:

```
import joblib

def iris_prediction(sepal_length, sepal_width, petal_length, petal_width):
    """
```

```
Given sepal length, sepal width, petal length, and petal width,
predict the class of iris
```

```
"""
```

```
# Load the model from the file
```

```
with open("model.pkl", "rb") as f:
```

```
    model = joblib.load(f)
```

```
# Construct the 2D matrix of values that .predict is expecting
```

```
X = [[sepal_length, sepal_width, petal_length, petal_width]]
```

```
# Get a list of predictions and select only 1st
```

```
predictions = model.predict(X)
```

```
prediction = int(predictions[0])
```

```
return {"predicted_class": prediction}
```

Finally, we'll also build our environment starting with the `requirements.txt` from that lesson:

```
scikit-learn==0.23.2
```

```
joblib==0.17.0
```

Cloud Functions without Flask

Previously, we deployed this cloud function using this `predict` function:

```
import json
```

```
def predict(request):
```

```
    """
```

```
`request` is an HTTP request object that will automatically be passed
in by Google Cloud Functions
```

```
You can find all of its properties and methods here:
```

```
https://flask.palletsprojects.com/en/1.0.x/api/#flask.Request
```

```
"""
```

```
# Get the request data from the user in JSON format
```

```
request_json = request.get_json()
```

```
# We are expecting the request to look like this:
```

```
# {"sepal_length": <x1>, "sepal_width": <x2>, "petal_length": <x3>, "petal_width": <x4>}
```

```
# Send it to our prediction function using ** to unpack the arguments
```

```
result = iris_prediction(**request_json)
```

```
# Return the result as a string with JSON format
return json.dumps(result)
```

Then bundling the model file, Python file, and requirements file into a single archive and uploading that to Google Cloud Functions.

That required a fair amount of configuration within Google Cloud Functions to specify the function to be invoked (`predict`), the permissions (public on the web), and the storage location for the archive.

Cloud Functions with Flask

When using Flask directly (rather than via the Google Cloud Functions implementation) and deploying on Heroku, we will need to import and declare a few more things within the code itself, but at the same time we won't need to configure as much within the website interface. We'll also be able to test our code locally!

Recall: Flask App Basics

Here was the source code of our previous simple Flask app:

```
# import flask here
from flask import Flask

# create new flask app here
app = Flask(__name__)

# define routes for your new flask app
@app.route('/', methods=['GET'])
def index():
    return 'Hello, world!'
```

We imported the Flask library, created a Flask app, and defined a single route `/`, which just returns the text `'Hello, world!'`.

Now let's add in those functions from our cloud function.

Adding ML Prediction Functionality to Our Flask App

Imports

Instead of just importing Flask, we'll also need to add in the `joblib` and `json` imports from the cloud function. We also need to import `request` from Flask so that we can parse the request data.

```
# Flask is the overall web framework
from flask import Flask, request
# joblib is used to unpickle the model
import joblib
# json is used to prepare the result
import json
```

Flask App Setup

This is the same as in our simple Flask app:

```
# create new flask app here
app = Flask(__name__)
```

Adding Cloud Function

Then we include our `iris_prediction` function from previously. In a more complex Flask app, this would likely be stored in a separate `.py` file, but we're keeping it all in one place for the sake of simplicity.

```
def iris_prediction(sepal_length, sepal_width, petal_length, petal_width):
    """
    Given sepal length, sepal width, petal length, and petal width,
    predict the class of iris
    """

    # Load the model from the file
    with open("model.pkl", "rb") as f:
        model = joblib.load(f)

    # Construct the 2D matrix of values that .predict is expecting
    X = [[sepal_length, sepal_width, petal_length, petal_width]]

    # Get a list of predictions and select only 1st
    predictions = model.predict(X)
    prediction = int(predictions[0])

    return {"predicted_class": prediction}
```

Defining Routes

For now, let's keep the `/` route as-is, then also add the `/predict` route.

Some notes on this change:

- `/predict` accepts HTTP `POST` requests, which is conventional for a form submission. Therefore we specify `methods=['POST']`
- Instead of having `request` be a function parameter like it was in our cloud function, instead it's something we imported earlier. However it works the same way as the function parameter.

```
@app.route('/', methods=['GET'])
def index():
    return 'Hello, world!'

@app.route('/predict', methods=['POST'])
def predict():
    # Get the request data from the user in JSON format
    request_json = request.get_json()

    # We are expecting the request to look like this:
    # {"sepal_length": <x1>, "sepal_width": <x2>, "petal_length": <x3>, "petal_width": <x4>}
    # Send it to our prediction function using ** to unpack the arguments
    result = iris_prediction(**request_json)

    # Return the result as a string with JSON format
    return json.dumps(result)
```

Pulling It All Together

When we bring together the imports, app setup, cloud function, and routes, the entire contents of `app.py` looks like this:

```
# Flask is the overall web framework
from flask import Flask, request
# joblib is used to unpickle the model
import joblib
# json is used to prepare the result
import json

# create new flask app here
app = Flask(__name__)

# helper function here

def iris_prediction(sepal_length, sepal_width, petal_length, petal_width):
    """
    Given sepal length, sepal width, petal length, and petal width,
    predict the class of iris
    """
```

```

# Load the model from the file
with open("model.pkl", "rb") as f:
    model = joblib.load(f)

# Construct the 2D matrix of values that .predict is expecting
X = [[sepal_length, sepal_width, petal_length, petal_width]]

# Get a list of predictions and select only 1st
predictions = model.predict(X)
prediction = int(predictions[0])

return {"predicted_class": prediction}

# defining routes here

@app.route('/', methods=['GET'])
def index():
    return 'Hello, world!'

@app.route('/predict', methods=['POST'])
def predict():
    # Get the request data from the user in JSON format
    request_json = request.get_json()

    # We are expecting the request to look like this:
    # {"sepal_length": <x1>, "sepal_width": <x2>, "petal_length": <x3>, "petal_width": <x4>}
    # Send it to our prediction function using ** to unpack the arguments
    result = iris_prediction(**request_json)

    # Return the result as a string with JSON format
    return json.dumps(result)

```

Running the Flask App Locally

You should already have a local environment called `flask-env` from the **Introduction to Flask** lesson. If you do not, go back to that lesson and follow the steps under **Setting up a Flask Environment**.

Preparing the Environment

Run this code in a new terminal window (separate from where you are running `jupyter notebook`) to activate `flask-env`:

```
conda activate flask-env
```

This environment has everything you need to run a basic Flask app, but it doesn't have the cloud function dependencies yet.

Run these commands in the terminal to install those dependencies:

```
pip install joblib==0.17.0
pip install scikit-learn==0.23.2
```

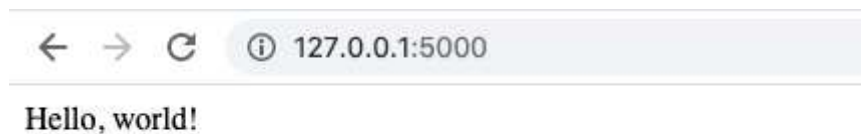
Now we should be ready to run our app!

Running the Flask Application

As previously, run this command in the terminal from the root of this repository:

```
export FLASK_ENV=development
env FLASK_APP=app.py flask run
```

If you open <http://127.0.0.1:5000/> in the browser, you should see this, just like before:



Leave the server running and let's use the `requests` library to send a request to our app!

```
import requests

response = requests.post(
    url="http://127.0.0.1:5000/predict",
    json={"sepal_length": 5.1, "sepal_width": 3.5, "petal_length": 1.4, "petal_width": 0.1}
)
response
```

```
<Response [200]>
```

The expected output of the above code cell is `<Response [200]>`. If you get a different response code, make sure that the code above matches the Flask app output where it says "Running on". For example, if you're running on port 5001 instead of 5000, make sure the `url` specified above matches.

```
response.json()
```


```
{'predicted_class': 0}
```

Great! You have now made an API request to a locally-running Flask app!

Go ahead and shut down the current Flask app by typing control-C in the terminal.

Deploying to Heroku

The real goal of deploying an app is not just to get a web server running on your local computer, it's to get it hosted live on the web!

[Heroku](https://www.heroku.com/)  (<https://www.heroku.com/>) is a platform-as-a-service company that is great for hosting this kind of application. We'll plan to use that, because it has a completely-free tier and allows you to host a Flask app with minimal setup steps.

Preparing the Repository for Heroku

Running a Production Server

Previously when we ran our Flask app, it was always in development mode. This is useful for playing around and editing code, but is unnecessarily slow for a published app.

Let's use a production-quality web server called [Waitress](https://docs.pylonsproject.org/projects/waitress/en/latest/)  (<https://docs.pylonsproject.org/projects/waitress/en/latest/>).

First, install it in the `flask-env`:

```
pip install waitress==2.1.1
```

Now instead of the `flask run` command, use this command to run the production server:

```
waitress-serve --port=5000 app:app
```

(You may need to allow Python to access the network, if your operating system gives you a pop-up.)

This should produce an output like this:


```
INFO:waitress:Serving on http://0.0.0.0:5000
```

Just like before, you should be able to copy the specified URL, paste it into the browser, and see your "Hello, World!" page.

The code below should also work:

```
response = requests.post(
    url="http://0.0.0.0:5000/predict",
    json={"sepal_length": 5.1, "sepal_width": 3.5, "petal_length": 1.4, "petal_width": 0.1}
)
response.json()
```

```
{'predicted_class': 0}
```

That's it! **Go ahead and shut down the server again using control-C.**

Requirements Files


When we made a cloud function for Google Cloud Functions, we used a `requirements.txt` file. For Heroku, we'll need three files like this:

1. `runtime.txt` : tells Heroku that we are running a Python application, and what version of Python
2. `requirements.txt` : lists the required Python packages (same as we did for the Google Cloud Function, adding Flask as a requirement)
3. `Procfile` : tells Heroku what command to run

All of these files are already located in this repository, but we'll explain how they work below so that you know how to make your own!

Our `runtime.txt` looks like this:

```
python-3.8.13
```

This is because we are running Python 3.8 in this conda environment. If you get an error about the "runtime" when trying to deploy with Heroku, it's possible that this version of Python is no longer supported. Look at the [supported runtimes](https://devcenter.heroku.com/articles/python-support#supported-runtimes)  (<https://devcenter.heroku.com/articles/python-support#supported-runtimes>) list to find other options.

Our `requirements.txt` looks like this:

```
Flask==2.0.3
joblib==0.17.0
```

```
scikit-learn==0.23.2  
waitress==2.1.1
```

Those are all the packages we installed with pip!

Finally, our **Procfile** looks like this:

```
web: waitress-serve --port=$PORT app:app
```

This is similar to what we ran in the terminal locally, except we added a **web:** to the beginning to indicate that this is a web process, and we parameterized **\$PORT** so that it will use whatever port Heroku is configured to use, rather than hard-coding it to 5000.

Setting Up the App on Heroku

Go to <https://signup.heroku.com/login>  (<https://signup.heroku.com/login>) and create an account (or log in if you already have one).

Then go to <https://dashboard.heroku.com/new-app>  (<https://dashboard.heroku.com/new-app>) to make a new app on Heroku.

The name can be anything you want, but must be unique. You can fill in a name if you have one in mind, or you can just click **Create app** and you'll get a randomly-suggested name.

Scroll down to **Deployment method** and choose **GitHub**. This will open another menu section, where you should click the **Connect to GitHub** button. You will get a pop-up window where you will be asked to sign in with GitHub.

Once connected, a text box should appear where you can search for the repository you want to use. (If you're just practicing with this lesson repo, make sure you have forked this repo to your GitHub account, then search for the lesson repo name.) Click **Search**, then click **Connect** on the appropriate repository.

Scroll down to **Manual deploy**, choose the appropriate branch, and click **Deploy Branch**.

If everything goes smoothly, you should see a build log, then the message **Your app was successfully deployed**. Then if you click the **View** button, that should open the "Hello, World!" page in a new browser tab.

In the cell below, replace the value of **base_url** with your actual Heroku app URL.

```
# base URL (ending with .herokuapp.com, no trailing /)  
base_url = ""  
response = requests.post(  
    url=f"{base_url}/predict",  
    json={"sepal_length": 5.1, "sepal_width": 3.5, "petal_length": 1.4, "petal_width": 0.1}
```

```
)  
response.json()
```

Troubleshooting Workflow on Heroku

Especially because the [supported runtimes](https://devcenter.heroku.com/articles/python-support#supported-runtimes) list changes very frequently, it's likely that your deployment won't succeed on the first try. That's ok!

Identifying the Problem

First make sure that the code works on your local computer. It is MUCH easier to debug when working locally vs. working on a cloud service like Heroku!

Then make sure you read the error message to understand what is going on and why:

- Are any of the necessary files missing? Double-check that you used Git to add, commit, and push all of the relevant pieces:
 - `runtime.txt` : the Python version
 - `Procfile` : the terminal command for Heroku to run
 - `requirements.txt` : the Python package requirements
 - `app.py` : the actual Flask app source code
 - `model.pkl` : the pickled model file
- If the error message mentions the "runtime", you probably need to review the list of supported runtimes and modify `runtime.txt` so that it reflects the new version
- If the error message happens during the `pip install` step, that might mean that one of the packages you're using is no longer available from the Python Package Index (the source where `pip` installs things from). Go to <https://pypi.org/> to research the packages you are trying to use, make a new `conda` environment locally, and try installing the packages one by one until you have a working `requirements.txt` file.
- If the error message happens when you're actually trying to view a page or run `requests.post`, most likely you didn't include all of the requirements in `requirements.txt`. You can run `pip freeze` in the terminal to see all of the packages you're using locally
- If the build logs aren't giving you enough information, go to **More --> View logs** to see the logs from the actual application running. This will give you information about the incoming requests.

Updating the Source Code on Heroku

First, use Git to add, commit, and push your changes to GitHub. Then go back to the **Deploy** tab, scroll to the bottom, and click **Deploy Branch**. Then wait to see if you get the "Your app was successfully deployed" message, and repeat the "Identifying the Problem" steps as needed.

You can also enable automatic deploys if you want to, but we tend to find that the manual process is easier to debug.

Level Up

Currently we are mainly using Flask to serve JSON content, but Flask is also a web server that can serve HTML!

If you are comfortable writing HTML, try modifying the `/` route so that it displays useful information, e.g. explaining how to call the API and make a prediction.

You can write multi-line HTML directly within `app.py` using a triple-quoted Python string like this:

```
@app.route('/', methods=['GET'])
def index():
    return """
    <h1>API Documentation</h1>
    <p>
        Paragraph of text here
    </p>
    """
```

Alternatively, you can create a `static` folder containing a file called `index.html`, then re-write the `/` route so it looks like this:

```
from flask import send_from_directory

@app.route('/', methods=['GET'])
def index():
    return send_from_directory("static", "index.html")
```

Summary

That's it! You have now learned about how to incorporate a cloud function into a Flask app, and how to deploy that Flask app on Heroku!