Deploying a Model with Dash

(https://github.com/learn-co-curriculum/dsc-dash-deployment) (https://github.com/learn-co-curriculum/dsc-dash-deployment/issues/new)

Introduction

In addition to allowing you to create complex dynamic web pages using Python code, Dash is "the most downloaded, trusted framework for building machine learning web apps in Python" (source (https://plotly.com/building-machine-learning-web-apps-in-python/)). Let's build one of those machine learning web apps!

Objectives

In this lesson you will:

- Combine Flask and Dash apps to serve API responses as well as dynamic web pages
- Incorporate machine learning predictions into a Dash app
- Deploy a machine learning dashboard using Dash and Heroku

Clone this lesson locally so you can follow along using your local `dash-env` .

Combining Flask and Dash

We previously mentioned that Dash is built on top of Flask. Specifically, when the Dash app is instantiated, an underlying Flask app is created by default.

If we want to customize the behavior of the underlying Flask app, we can actually instantiate it separately, then pass it in as an argument when we instantiate the Dash app.

Below is a simple "Hello, World!" example where the Flask and Dash apps have been combined in this way:

```
from jupyter_dash import JupyterDash as Dash
from dash import html
from flask import Flask

# create a new flask app
flask_app = Flask(__name__)
# create a new dash app built on that flask app
dash_app = Dash(__name__, server=flask_app)
```

```
# set the layout to include a single <p> tag containing "Hello, World!"
dash_app.layout = html.P("Hello, World!")

import warnings
warnings.filterwarnings('ignore')

dash_app.run_server(mode="inline", height=150, host="localhost", port=5000)
```

So far, this works identically to our "Hello, World!" Dash app without Flask.

However, now we can also add a *route* to the Flask app that returns JSON:

```
import json

# create a new flask app
flask_app = Flask(__name__)
# create a new dash app built on that flask app
dash_app = Dash(__name__, server=flask_app)

# set the layout to include a single <p> tag containing "Hello, World!"
dash_app.layout = html.P("Hello, World!")

# add a route to the flask app that returns json
@flask_app.route("/get_json", methods=["GET"])
def hello_json():
    return json.dumps({
        "hello": "world!",
        "key": "value!"
    })


dash_app.run_server(mode="inline", height=150, host="localhost", port=5000)
```

The web page view looks exactly the same, but we can also run a query directly to the backend and get a JSON result!

```
import requests

response = requests.get("http://localhost:5000/get_json")
response.json()
```

Now that we have those basic elements, let's combine our Iris Dataset Flask API and Dash dashboard from the previous lessons.

Creating a Machine Learning Dash App

Recall: Previous Flask and Dash Apps

Our previous Flask app looked like this:

```
from flask import Flask, request
import joblib
import json

app = Flask(__name__)

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
    """
    with open("model.pkl", "rb") as f:
        model = joblib.load(f)
    X = [[sepal_length, sepal_width, petal_length, petal_width]]
    predictions = model.predict(X)
    # model.predict takes a list of records and returns a list of predictions
    # but we are only making a single prediction
    prediction = int(predictions[0])
    return {"predicted_class": prediction}

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

@app.route('/predict', methods=['POST'])
def predict():
    request_json = request.get_json()
    result = iris_prediction(**request_json)
    return json.dumps(result)
```

This is a basic web server that returns the text "Hello, world!" when it receives a `GET` request to view the home page (`/` route), and returns an iris classification prediction when it receives a `POST` request to the `/predict` route.

Our previous Dash app looked like this:

```
from jupyter_dash import JupyterDash as Dash
from dash import html, dcc, dash_table, Input, Output
import dash_bootstrap_components as dbc
```

```
import pandas as pd
from sklearn.datasets import load_iris

external_stylesheets = [dbc.themes.BOOTSTRAP]
app = Dash(__name__, external_stylesheets=external_stylesheets)

markdown = dcc.Markdown("""
# Iris Dataset

Below is a DataTable showing a sample of 20 records from the
 [Iris Dataset](https://en.wikipedia.org/wiki/Iris_flower_data_set).

Select any record to view more information!
""")

data = load_iris()
X = pd.DataFrame(data.data, columns=data.feature_names)
y = pd.Series(data.target, name="class")
full_dataset = pd.concat([X, y], axis=1)
table = dash_table.DataTable(
    data=full_dataset.sample(10, random_state=1).to_dict(orient="records"),
    row_selectable="single",
    cell_selectable=False,
    id="tbl"
)

modal = dbc.Modal(children=[
    dbc.ModalHeader(dbc.ModalTitle("Iris Information")),
    dbc.ModalBody(id="modal-body")
],
                  id="modal",
                  is_open=False
                 )

app.layout = html.Div(children=[
    html.Div(markdown),
    html.Div(table),
    modal
])

def create_list_group(selected_row_data):
    return dbc.ListGroup([
        dbc.ListGroupItem(f"{k}: {v}") for k, v in selected_row_data.items()
    ])
```

```
def create_image_card(selected_row_data):
    iris_class = selected_row_data["class"]
    if iris_class == 0:
        name = "Iris setosa "
        img_url = "https://upload.wikimedia.org/wikipedia/commons/thumb/5/56/Kosaciec_szc
        img_source = "https://commons.wikimedia.org/wiki/File:Kosaciec_szczecinkowaty_Iri
    elif iris_class == 1:
        name = "Iris versicolor "
        img_url = "https://upload.wikimedia.org/wikipedia/commons/thumb/4/41/Iris_versico
        img_source = "https://commons.wikimedia.org/wiki/File:Iris_versicolor_3.jpg"
    else:
        name = "Iris virginica "
        img_url = "https://upload.wikimedia.org/wikipedia/commons/thumb/9/9f/Iris_virgini
        img_source = "https://commons.wikimedia.org/wiki/File:Iris_virginica.jpg"

    return dbc.Card(children=[
        dbc.CardImg(src=img_url),
        dbc.CardBody(children=[
            html.Em(name),
            html.Small(html.A("(image source)", href=img_source, target="blank_"))
        ])
    ])

@app.callback(Output("modal", "is_open"), Input("tbl", "selected_rows"))
def toggle_modal(selected_rows):
    if selected_rows:
        return True
    else:
        return False

@app.callback(
    Output("modal-body", "children"),
    [Input("tbl", "derived_virtual_data"), Input("tbl", "selected_rows")])
def render_information(rows, selected_rows):
    if selected_rows:
        # selection is set to "single" so there will be exactly 1 selected row
        selected_row_data = rows[selected_rows[0]]
        return html.Div(dbc.Row(children=[
            dbc.Col(create_image_card(selected_row_data)),
            dbc.Col(create_list_group(selected_row_data))
        ]))
```

This is an interactive web page that displays a table of data, where the user can click on a record and a modal will pop up with additional information.

A Basic Combination

If we just want to combine the functionality of the two applications, we just need to declare the two apps appropriately, and change the names from `app` to either `flask_app` or `dash_app`. We'll go ahead and remove the `/` route from the Flask app because the Dash app is using the home page to display our interactive table.

```
########## IMPORTS ##########

from flask import Flask, request
import joblib
import json

from jupyter_dash import JupyterDash as Dash
from dash import html, dcc, dash_table, Input, Output
import dash_bootstrap_components as dbc

import pandas as pd
from sklearn.datasets import load_iris

########## SETTING UP THE APPS ##########

flask_app = Flask(__name__)
external_stylesheets = [dbc.themes.BOOTSTRAP]
dash_app = Dash(__name__, external_stylesheets=external_stylesheets, server=flask_app)

########## DECLARING LAYOUT COMPONENTS ##########

markdown = dcc.Markdown("""
# Iris Dataset

Below is a DataTable showing a sample of 20 records from the
 [Iris Dataset](https://en.wikipedia.org/wiki/Iris_flower_data_set).

Select any record to view more information!
""")

data = load_iris()
X = pd.DataFrame(data.data, columns=data.feature_names)
y = pd.Series(data.target, name="class")
full_dataset = pd.concat([X, y], axis=1)
table = dash_table.DataTable(
    data=full_dataset.sample(10, random_state=1).to_dict(orient="records"),
    row_selectable="single",
```

```
    cell_selectable=False,
    id="tbl"
)

modal = dbc.Modal(children=[
    dbc.ModalHeader(dbc.ModalTitle("Iris Information")),
    dbc.ModalBody(id="modal-body")
],
                  id="modal",
                  is_open=False
                 )

dash_app.layout = html.Div(children=[
    html.Div(markdown),
    html.Div(table),
    modal
])

########## HELPER FUNCTIONS ##########

def create_list_group(selected_row_data):
    return dbc.ListGroup([
        dbc.ListGroupItem(f"{k}: {v}") for k, v in selected_row_data.items()
    ])

def create_image_card(selected_row_data):
    iris_class = selected_row_data["class"]
    if iris_class == 0:
        name = "Iris setosa "
        img_url = "https://upload.wikimedia.org/wikipedia/commons/thumb/5/56/Kosaciec_szc
        img_source = "https://commons.wikimedia.org/wiki/File:Kosaciec_szczecinkowaty_Iri
    elif iris_class == 1:
        name = "Iris versicolor "
        img_url = "https://upload.wikimedia.org/wikipedia/commons/thumb/4/41/Iris_versico
        img_source = "https://commons.wikimedia.org/wiki/File:Iris_versicolor_3.jpg"
    else:
        name = "Iris virginica "
        img_url = "https://upload.wikimedia.org/wikipedia/commons/thumb/9/9f/Iris_virgini
        img_source = "https://commons.wikimedia.org/wiki/File:Iris_virginica.jpg"

    return dbc.Card(children=[
        dbc.CardImg(src=img_url),
        dbc.CardBody(children=[
            html.Em(name),
            html.Small(html.A("(image source)", href=img_source, target="blank_"))
```

```
            ])
        ])

    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
        """
        with open("model.pkl", "rb") as f:
            model = joblib.load(f)
        X = [[sepal_length, sepal_width, petal_length, petal_width]]
        predictions = model.predict(X)
        # model.predict takes a list of records and returns a list of predictions
        # but we are only making a single prediction
        prediction = int(predictions[0])
        return {"predicted_class": prediction}

    ########## CALLBACKS ##########

    @dash_app.callback(Output("modal", "is_open"), Input("tbl", "selected_rows"))
    def toggle_modal(selected_rows):
        if selected_rows:
            return True
        else:
            return False

    @dash_app.callback(
        Output("modal-body", "children"),
        [Input("tbl", "derived_virtual_data"), Input("tbl", "selected_rows")])
    def render_information(rows, selected_rows):
        if selected_rows:
            # selection is set to "single" so there will be exactly 1 selected row
            selected_row_data = rows[selected_rows[0]]
            return html.Div(dbc.Row(children=[
                dbc.Col(create_image_card(selected_row_data)),
                dbc.Col(create_list_group(selected_row_data))
            ]))

    ########## ROUTES ##########

    @flask_app.route('/predict', methods=['POST'])
    def predict():
        request_json = request.get_json()
        result = iris_prediction(**request_json)
        return json.dumps(result)
```

As you can see, the Dash app works like it did before:

```
dash_app.run_server(mode="inline", height=500, host="localhost", port=5000)
```

And so does the Flask app!

```
response = requests.post(
    url="http://localhost:5000/predict",
    json={"sepal_length": 5.1, "sepal_width": 3.5, "petal_length": 1.4, "petal_width": 0.2
)
response.json()
```

Incorporating Predictions into Our Dash App

Right now our modal is interactive, but it only displays static data from the table. Let's zoom in on that code with some example data:

```
example_data = {
    "sepal length (cm)": 5.8,
    "sepal width (cm)": 4,
    "petal length (cm)": 1.2,
    "petal width (cm)": 0.2,
    "class": 0
}

modal_body = dbc.ModalBody(html.Div(dbc.Row(children=[
    dbc.Col(create_image_card(example_data)),
    dbc.Col(create_list_group(example_data))
])))

dash_app.layout = dbc.Modal(children=[
    dbc.ModalHeader(dbc.ModalTitle("Iris Information")),
    modal_body
], is_open=True)

dash_app.run_server(mode="inline", height=500, host="localhost", port=5000)
```

What if instead of just displaying the `class` from the table, we showed both the actual class and the predicted class?

Remember, we already have a helper function from our Flask app called `iris_prediction`. Let's make an additional helper function that takes in the dictionary of data, passes the relevant arguments to `iris_prediction`, then displays information about whether the prediction was correct.

```
def check_prediction(selected_row_data):
    # make a copy so we don't mess up the original data record
    data_copy = selected_row_data.copy()
    # remove and store the actual class
    actual_class = data_copy.pop("class")
    # remove " (cm)" from labels and replace spaces with underscores
    data_cleaned = {k.split(" (cm)")[0].replace(" ", "_"):v for k, v in data_copy.items()}

    # get the result dictionary from iris_prediction
    result = iris_prediction(**data_cleaned)
    # extract the predicted class from the result dictionary
    predicted_class = result["predicted_class"]

    # determine whether the prediction was correct
    correct_prediction = predicted_class == actual_class
    if correct_prediction:
        color = "success"
    else:
        color = "danger"

    return dbc.Alert(f"Predicted class: {predicted_class}", color=color)

modal_body = dbc.ModalBody(html.Div(dbc.Row(children=[
    dbc.Col(create_image_card(example_data)),
    dbc.Col(children=[
        create_list_group(example_data),
        # adding a horizontal rule divider here
        html.Hr(),
        # adding the prediction check here
        check_prediction(example_data)
    ])
])))

dash_app.layout = dbc.Modal(children=[
    dbc.ModalHeader(dbc.ModalTitle("Iris Information")),
    modal_body
], is_open=True)
```

```
dash_app.run_server(mode="inline", height=500, host="localhost", port=5000)
```

Combining that all together with our data table, our complete app now looks like this:

```
########## IMPORTS ##########

from flask import Flask, request
import joblib
import json

from jupyter_dash import JupyterDash as Dash
from dash import html, dcc, dash_table, Input, Output
import dash_bootstrap_components as dbc

import pandas as pd
from sklearn.datasets import load_iris

########## SETTING UP THE APPS ##########

flask_app = Flask(__name__)
external_stylesheets = [dbc.themes.BOOTSTRAP]
dash_app = Dash(__name__, external_stylesheets=external_stylesheets, server=flask_app)

########## DECLARING LAYOUT COMPONENTS ##########

markdown = dcc.Markdown("""
# Iris Dataset

Below is a DataTable showing a sample of 20 records from the
 [Iris Dataset](https://en.wikipedia.org/wiki/Iris_flower_data_set).

Select any record to view more information!
""")

data = load_iris()
X = pd.DataFrame(data.data, columns=data.feature_names)
y = pd.Series(data.target, name="class")
full_dataset = pd.concat([X, y], axis=1)
table = dash_table.DataTable(
    data=full_dataset.sample(10, random_state=4).to_dict(orient="records"),
    row_selectable="single",
    cell_selectable=False,
    id="tbl"
)

modal = dbc.Modal(children=[
    dbc.ModalHeader(dbc.ModalTitle("Iris Information")),
```

```
        dbc.ModalBody(id="modal-body")
    ],
                      id="modal",
                      is_open=False
                     )

    dash_app.layout = html.Div(children=[
        html.Div(markdown),
        html.Div(table),
        modal
    ])

    ########## HELPER FUNCTIONS ##########

    def create_list_group(selected_row_data):
        return dbc.ListGroup([
            dbc.ListGroupItem(f"{k}: {v}") for k, v in selected_row_data.items()
        ])

    def create_image_card(selected_row_data):
        iris_class = selected_row_data["class"]
        if iris_class == 0:
            name = "Iris setosa "
            img_url = "https://upload.wikimedia.org/wikipedia/commons/thumb/5/56/Kosaciec_szc
            img_source = "https://commons.wikimedia.org/wiki/File:Kosaciec_szczecinkowaty_Iri
        elif iris_class == 1:
            name = "Iris versicolor "
            img_url = "https://upload.wikimedia.org/wikipedia/commons/thumb/4/41/Iris_versico
            img_source = "https://commons.wikimedia.org/wiki/File:Iris_versicolor_3.jpg"
        else:
            name = "Iris virginica "
            img_url = "https://upload.wikimedia.org/wikipedia/commons/thumb/9/9f/Iris_virgini
            img_source = "https://commons.wikimedia.org/wiki/File:Iris_virginica.jpg"

        return dbc.Card(children=[
            dbc.CardImg(src=img_url),
            dbc.CardBody(children=[
                html.Em(name),
                html.Small(html.A("(image source)", href=img_source, target="blank_"))
            ])
        ])

    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
    """
    with open("model.pkl", "rb") as f:
        model = joblib.load(f)
    X = [[sepal_length, sepal_width, petal_length, petal_width]]
    predictions = model.predict(X)
    # model.predict takes a list of records and returns a list of predictions
    # but we are only making a single prediction
    prediction = int(predictions[0])
    return {"predicted_class": prediction}

def check_prediction(selected_row_data):
    """
    Return an Alert component with information about the model's prediction
    vs. the true class value
    """
    data_copy = selected_row_data.copy()
    actual_class = data_copy.pop("class")
    # remove " (cm)" from labels
    data_cleaned = {k.split(" (cm)")[0].replace(" ", "_"):v for k, v in data_copy.items()}
    result = iris_prediction(**data_cleaned)
    predicted_class = result["predicted_class"]
    correct_prediction = predicted_class == actual_class
    if correct_prediction:
        color = "success"
    else:
        color = "danger"
    return dbc.Alert(f"Predicted class: {predicted_class}", color=color)

########## CALLBACKS ##########

@dash_app.callback(Output("modal", "is_open"), Input("tbl", "selected_rows"))
def toggle_modal(selected_rows):
    if selected_rows:
        return True
    else:
        return False

@dash_app.callback(
    Output("modal-body", "children"),
    [Input("tbl", "derived_virtual_data"), Input("tbl", "selected_rows")])
def render_information(rows, selected_rows):
    if selected_rows:
        # selection is set to "single" so there will be exactly 1 selected row
        selected_row_data = rows[selected_rows[0]]
```

```
        return html.Div(dbc.Row(children=[
            dbc.Col(create_image_card(selected_row_data)),
            dbc.Col(children=[
                create_list_group(selected_row_data),
                html.Hr(),
                check_prediction(selected_row_data)
            ])
        ]))

########## ROUTES ##########

@flask_app.route('/predict', methods=['POST'])
def predict():
    request_json = request.get_json()
    result = iris_prediction(**request_json)
    return json.dumps(result)
```

(We changed the random seed used to generate the sample so that it includes some places where the model makes a mistake. Try clicking through to find them!)

```
dash_app.run_server(mode="inline", height=500, host="localhost", port=5000)
```

Nice! Now we have a dashboard that dynamically generates machine learning predictions whenever the user clicks on a record!

Predictions on Unseen Data

So far we have an interesting view into our model's performance on the training data, but typically the value of a deployed model is to make predictions on new, unseen data.

Let's make an interface that allows the user to enter values into a form, then makes predictions using those user-supplied values.

In order to figure out what kind of inputs our form should have, let's look at our `X` training data:

```
X.describe()
```

Great, so it looks like these can all be numeric inputs. One way we might implement numeric inputs would be using a text box, but let's use a Dash **Slider (https://dash.plotly.com/dash-core-components/slider)** component instead. Sliders feel a bit more interactive, and they also let you set upper and lower bounds on what inputs the user can specify.

A basic slider looks like this:

```
dash_app.layout = dcc.Slider(
    0, # minimum value
    10, # maximum value
    value=3.1, # default value before the user changes anything
    tooltip={"always_visible": True} # always display selected value
)


dash_app.run_server(mode="inline", height=100, host="localhost", port=5000)
```

Let's make a set of four sliders, with ranges based on the four features:

```
def create_sliders(X):
    slider_items = []
    # loop over all of the columns in X
    for column in X:
        # make a label with the column name
        label = html.H5(column)

        # get the minimum, maximum, and median values for the column
        lower_bound = X[column].min()
        upper_bound = X[column].max()
        value = X[column].median()

        # make a slider with the right values
        slider = dcc.Slider(
            lower_bound,
            upper_bound,
            value=value, # set median as default
            marks=None,
            tooltip={"always_visible": True},
            id=column # set id based on column name
        )

        # make a list item with the label and the slider
        item = dbc.ListGroupItem(children=[
            label,
            slider
        ])
        slider_items.append(item)
    return dbc.ListGroup(slider_items)


dash_app.layout = html.Div(children=[
    create_sliders(X),
```

```
    dbc.Alert("Prediction will go here", color="info", id="prediction-output")
])

dash_app.run_server(mode="inline", height=450, host="localhost", port=5000)
```

Now let's add a callback so that the slider values are fed into the prediction function:

```
@dash_app.callback(
    Output("prediction-output", "children"),
    [
        # list comprehension to specify all of the input columns
        Input(column, "value") for column in X.columns
    ]
)
def generate_user_input_prediction(*args):
    return f"Predicted class: {iris_prediction(*args)['predicted_class']}"
```

Try out the slider values to see if you can get the predicted class to change!

```
dash_app.run_server(mode="inline", height=450, host="localhost", port=5000)
```

Awesome, now we are making predictions on unseen data!

To combine the two interfaces, let's make a set of Tab components (**documentation here** (https://dash-bootstrap-components.opensource.faculty.ai/docs/components/tabs/)) so the user can switch between the table view and the form view.

Here is a minimal version of our Tabs:

```
tabs = dbc.Tabs(children=[
        dbc.Tab(dbc.Alert("Form will go here", color="secondary"), label="Generate Predic
        dbc.Tab(dbc.Alert("Table will go here", color="secondary"), label="Analyze Perfor
    ])

dash_app.layout = html.Div(children=[
    html.H1("Iris Classification Model"),
    tabs
])
```

Try clicking on the tab names to switch between them:

```
dash_app.run_server(mode="inline", height=200, host="localhost", port=5000)
```

Now let's add the actual content to those tabs. Below is the full, final version of our Dash + Flask app:

```
########## IMPORTS ##########

from flask import Flask, request
import joblib
import json

from jupyter_dash import JupyterDash as Dash
from dash import html, dcc, dash_table, Input, Output
import dash_bootstrap_components as dbc

import pandas as pd
from sklearn.datasets import load_iris

########## SETTING UP THE APPS ##########

flask_app = Flask(__name__)
external_stylesheets = [dbc.themes.BOOTSTRAP]
dash_app = Dash(__name__, external_stylesheets=external_stylesheets, server=flask_app)

########## HELPER FUNCTIONS ##########

def create_sliders(X):
    slider_items = []
    for column in X:
        label = html.H5(column)

        lower_bound = X[column].min()
        upper_bound = X[column].max()
        value = X[column].median()

        slider = dcc.Slider(
            lower_bound,
            upper_bound,
            value=value, # set median as default
            marks=None,
            tooltip={"always_visible": True},
            id=column # set id based on column name
        )

        item = dbc.ListGroupItem(children=[
            label,
            slider
        ])
        slider_items.append(item)
```

```
    return dbc.ListGroup(slider_items)

def create_list_group(selected_row_data):
    return dbc.ListGroup([
        dbc.ListGroupItem(f"{k}: {v}") for k, v in selected_row_data.items()
    ])

def create_image_card(selected_row_data):
    iris_class = selected_row_data["class"]
    if iris_class == 0:
        name = "Iris setosa "
        img_url = "https://upload.wikimedia.org/wikipedia/commons/thumb/5/56/Kosaciec_szc
        img_source = "https://commons.wikimedia.org/wiki/File:Kosaciec_szczecinkowaty_Iri
    elif iris_class == 1:
        name = "Iris versicolor "
        img_url = "https://upload.wikimedia.org/wikipedia/commons/thumb/4/41/Iris_versico
        img_source = "https://commons.wikimedia.org/wiki/File:Iris_versicolor_3.jpg"
    else:
        name = "Iris virginica "
        img_url = "https://upload.wikimedia.org/wikipedia/commons/thumb/9/9f/Iris_virgini
        img_source = "https://commons.wikimedia.org/wiki/File:Iris_virginica.jpg"

    return dbc.Card(children=[
        dbc.CardImg(src=img_url),
        dbc.CardBody(children=[
            html.Em(name),
            html.Small(html.A("(image source)", href=img_source, target="blank_"))
        ])
    ])

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
    """
    with open("model.pkl", "rb") as f:
        model = joblib.load(f)
    X = [[sepal_length, sepal_width, petal_length, petal_width]]
    predictions = model.predict(X)
    # model.predict takes a list of records and returns a list of predictions
    # but we are only making a single prediction
    prediction = int(predictions[0])
    return {"predicted_class": prediction}

def check_prediction(selected_row_data):
```

```
    """
    Return an Alert component with information about the model's prediction
    vs. the true class value
    """
    data_copy = selected_row_data.copy()
    actual_class = data_copy.pop("class")
    # remove " (cm)" from labels
    data_cleaned = {k.split(" (cm)")[0].replace(" ", "_"):v for k, v in data_copy.items()}
    result = iris_prediction(**data_cleaned)
    predicted_class = result["predicted_class"]
    correct_prediction = predicted_class == actual_class
    if correct_prediction:
        color = "success"
    else:
        color = "danger"
    return dbc.Alert(f"Predicted class: {predicted_class}", color=color)

########## DECLARING LAYOUT COMPONENTS ##########

data = load_iris()
X = pd.DataFrame(data.data, columns=data.feature_names)
y = pd.Series(data.target, name="class")
full_dataset = pd.concat([X, y], axis=1)

prediction_layout = html.Div(children=[
    create_sliders(X),
    dbc.Alert("Prediction will go here", color="info", id="prediction-output")
])

markdown = dcc.Markdown("""
## Iris Training Dataset

Below is a DataTable showing a sample of 20 records from the
 [Iris Dataset](https://en.wikipedia.org/wiki/Iris_flower_data_set).

Select any record to view more information!
""")

table = dash_table.DataTable(
    data=full_dataset.sample(10, random_state=4).to_dict(orient="records"),
    row_selectable="single",
    cell_selectable=False,
    id="tbl"
)
```

```
modal = dbc.Modal(children=[
    dbc.ModalHeader(dbc.ModalTitle("Iris Information")),
    dbc.ModalBody(id="modal-body")
],
                  id="modal",
                  is_open=False
                 )

past_data_layout = html.Div(children=[
    html.Div(markdown),
    html.Div(table),
    modal
])

tabs = dbc.Tabs(children=[
        dbc.Tab(prediction_layout, label="Generate Predictions on New Data"),
        dbc.Tab(past_data_layout, label="Analyze Performance on Past Data")
    ])

dash_app.layout = dbc.Container(children=[
    html.H1("Iris Classification Model"),
    tabs
])

########## CALLBACKS ##########

@dash_app.callback(
    Output("prediction-output", "children"),
    [
        # list comprehension to specify all of the input columns
        Input(column, "value") for column in X.columns
    ]
)
def generate_user_input_prediction(*args):
    return f"Predicted class: {iris_prediction(*args)['predicted_class']}"

@dash_app.callback(Output("modal", "is_open"), Input("tbl", "selected_rows"))
def toggle_modal(selected_rows):
    if selected_rows:
        return True
    else:
        return False

@dash_app.callback(
    Output("modal-body", "children"),
```

```
    [Input("tbl", "derived_virtual_data"), Input("tbl", "selected_rows")])
def render_information(rows, selected_rows):
    if selected_rows:
        # selection is set to "single" so there will be exactly 1 selected row
        selected_row_data = rows[selected_rows[0]]
        return html.Div(dbc.Row(children=[
            dbc.Col(create_image_card(selected_row_data)),
            dbc.Col(children=[
                create_list_group(selected_row_data),
                html.Hr(),
                check_prediction(selected_row_data)
            ])
        ]))

########## ROUTES ##########

@flask_app.route('/predict', methods=['POST'])
def predict():
    request_json = request.get_json()
    result = iris_prediction(**request_json)
    return json.dumps(result)
```

```
dash_app.run_server(mode="inline", height=550, host="localhost", port=5000)
```

Note that the API backend also still works!

```
response = requests.post(
    url="http://localhost:5000/predict",
    json={"sepal_length": 5.1, "sepal_width": 3.5, "petal_length": 1.4, "petal_width": 0.
)
response.json()
```

Deploying to Heroku

Deploying our Dash app to Heroku is very similar to deploying our Flask app to Heroku. We'll need to:

1. Install a **production server** and make sure we can successfully run our app locally via the command line
2. Create our **requirements files** and push them to GitHub
3. Create a new app through the **Heroku web interface** and connect it to our GitHub repo

Let's get started!

Installing and Running a Production Server

Installing Waitress

Once again, we'll use a production-quality server called **Waitress (https://docs.pylonsproject.org/projects/waitress/en/latest/)**.

In the terminal, make sure you have `dash-env` activated, then run:

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

Organizing the Code into a `.py` File

Waitress needs the code to be located in a `.py` file rather than directly within a Jupyter Notebook. We have already copied the above code into a file called `app.py`.

Note that there is one small difference between the code above and the code in `app.py`: instead of importing `Dash` from the `jupyter_dash` library, we imported it directly from the `dash` library. In other words, instead of

```
from jupyter_dash import JupyterDash as Dash
```

`app.py` has

```
from dash import Dash
```

This is because when we're deploying the application, we don't want the version of Dash designed to run in a notebook, we want the standalone version. But otherwise the code is identical to the "final version" code shown in this notebook.

Running the Production Server

Run this command in the terminal:

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

Note that this is very similar to our command when we were deploying our Flask app, which was `waitress-serve --port=5000 app:app`. The difference is that the Flask app inside of `app.py` had a variable name of `app`, whereas now it has the name `flask_app`.

You should see an output like this:

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

and you should be able to open up that link in your browser!

Once you have confirmed that this works, go ahead and shut down the server using control-C.

Requirements Files

Again, this is very similar to when we deployed the Flask app. We just have a longer list of requirements, and our `Procfile` specifies that the app variable name is `flask_app` rather than app. These files have already been included in this repository.

`runtime.txt`

This is the same as when we deployed our Flask app. Remember that if you get an error related to the runtime, that means that the **supported runtimes list** (https://devcenter.heroku.com/articles/python-support#supported-runtimes) has changed. Check that link to find the most up-to-date Python 3.8 runtime and edit `runtime.txt` accordingly.

```
python-3.8.13
```

`requirements.txt`

Our previous `requirements.txt` looked like this:

```
Flask==2.0.3
joblib==0.17.0
scikit-learn==0.23.2
waitress==2.1.1
```

Our new `requirements.txt` has those same items, with additional ones added:

```
Flask==2.0.3
dash==2.3
dash-bootstrap-components==1.0
pandas==1.4
joblib==0.17.0
scikit-learn==0.23.2
waitress==2.1.1
```

To explain further:

- `dash` and `dash-bootstrap-components` have been added so that we can create layouts and callbacks with Dash
- `pandas` has been added so that we can manipulate the dataset in preparation for displaying it in a table. If you just want your app to make predictions on unseen data, you won't need `pandas` .

This is also slightly different from the `dash-env` requirements we installed earlier, because the deployed version of our app does not need `notebook` or `jupyter-dash` (or be tethered to a specific version of `Werkzeug`, which was needed at the time of this writing to make `jupyter-dash` work correctly).

`Procfile`

Our `Procfile` is also slightly different because our app variable name is different.

Our previous `Procfile` looked like this:

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

And our new one looks like this:

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

Deploying the App on Heroku

This is the same set of steps as before, but we'll include them here for convenience:

1. Make sure you are **logged in** to Heroku. You can go to https://dashboard.heroku.com/ (https://dashboard.heroku.com/) and it will either show you your list of apps or redirect you to the login page.
   - You should already have an account if you followed the steps from the Flask deployment lesson, but you can also create an account now.
2. Go to https://dashboard.heroku.com/new-app (https://dashboard.heroku.com/new-app) to make a new app on Heroku.
3. Either fill in a name for your app then click **Create app**, or just click **Create app** if you want a name to be generated for you.
4. Scroll down to **Deployment method** and choose **GitHub**.
   - You should already be signed in with GitHub if you followed the steps from the Flask deployment lesson, but you can also approve the connection in the pop-up window now if you haven't previously.
5. **Search** for the repository you want, then click **Connect** on the repository in the list of search results.
6. Scroll down to **Manual deploy**, choose the appropriate branch, and click **Deploy Branch**.
7. Wait for the app to build, then once you see the message "Your app was successfully deployed" click the **View** button to open up your Dash app!

The interface should be much more interesting than the "Hello, world!" from your Flask app. You should also be able to make API requests (replace `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.
)
response.json()
```

Level Up

Dash is created by Plotly, which also makes a well-known **Python graphing library (https://plotly.com/python/)** that creates interactive graphs using Python. Check out the examples **here (https://dash.plotly.com/dash-core-components/graph)** and try adding a visualization to your dashboard!

In addition to being used as layout elements, Plotly graphs can be attached to callbacks in a Dash page. So, for example, the graph axis scale could change using a slider input, or a user could click on a point on a graph and send that to their model to make a prediction. Try it out for yourself!

Summary

You have now learned how to combine Flask with Dash to build and deploy powerful dynamic web applications using machine learning. We can't wait to see what you'll make next!