Lab 6 – Deploy a model (95)

I copied the final version of Lab4 and striped out most of the cells that we used to explore a regression model. We are going to use linear regression with polynomial features, but it is relatively easy to change models as needed.

The base code in in lab6_starter and includes a basic linear regression on the full Howell dataset.

Build a Pipeline.

Copy the markdown cell and associated code cell for ***Train and Evaluate a Linear Regression Model*** just before the ***Results***.

In the markdown cell add **Pipelined** in front of Linear.

In the Train and Evaluate cell, add the following imports after the existing imports.

In the code cell we need to add lines to import the transformers we will use.

```
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import PolynomialFeatures
from sklearn.preprocessing import StandardScaler
```

We next create each of the components for the pipeline. In the code cell after y_test is assigned a value, add the following lines.

```
imp_mean = SimpleImputer(missing_values=np.nan, strategy='mean')
poly4 = PolynomialFeatures(degree=4, include_bias=False)
scale = StandardScaler()

lr_model = LinearRegression()
```

1. While the Howell data is clean, our input may not be. We handle any missing features by replacing them with the average for that feature.
2. We know that the linear model is underfitting, so we will add in higher degree features. We want to be judicious, so we will only add terms up to degree 4.
3. While the original data does not suffer from a severe imbalance in scale, adding in polynomial features does grow the range, so we add in a standard scaler. It will compute z-scores for all the features.

   Online references for the transforms can be found here:

https://scikit-learn.org/stable/modules/classes.html#module-sklearn.preprocessing

The creation of the pipeline will look familiar. We use a list of pairs, each of which is comprised of a name and the component. I made the tag and component variable name to same, but that is not required.

```
stages = [('imp_mean', imp_mean),
          ('poly4', poly4),
          ('scale', scale),
          ('lr_model', lr_model),
         ]

pipe model = Pipeline(stages)
```

Train the Pipeline.

Now we just need to be careful that all the lr_model references are replaced by pipe_model. The lines that need to be fixed are:

```
pipe_model.fit(X,y)

y_pred = pipe_model.predict(X)
print('Results for pipeline linear regression on training data')
#print('  Default settings')
#print('Internal parameters:')
print('   Bias is ', pipe_model.predict([[0]]))
#print('   Coefficients', pipe_model.coef_)
print('   Score', pipe_model.score(X,y))

y_test_pred = pipe_model.predict(X_test)
print()
print('Results for pipeline linear regression on test data')
```

1. Pipe_model is a model, so we can train it on the data using fit.
2. While we can get the intercept from the linear regression model, it will not be the bias for the entire pipeline. We can always get the bias by setting all the input parameters to have a zero value and the response will be the bias.
3. Unfortunately, there is no similar easy trick for the coefficients.

Run the cell and you should see results that look something like.

```
Results for pipeline linear regression on training data
  Bias is  [-166.29034128]
  Score 0.936895492823189
MAE is  2.7616342586301963
RMSE is  3.700602610294623
MSE is  13.694459679319376
```

R^2 0.936895492823189

Results for pipeline linear regression on test data
MAE is 3.090635447854583
RMSE is 4.136777328711979
MSE is 17.112926667345416
R^2 0.9196783079821763

Submission 1 of 11: Progress mark (10)
Screen shot results.

Record the results in the final cell by a row in the table

```
|Linear Regression Pipe|Weight|3.70|93.69|4.14|91.97|
```

Submission 2 of 11: Analysis (5)
Compare performance of linear and the pipeline which is polynomial.

Graph the Polynomial Curve.

Create a new text cell after the **Train and Evaluate Pipelined Linear Regression** cell and add in the following

```
### Plot pipelined linear regression model
Using height to predict weight
```

We need the graphing code to use the model when it creates the curve

```
y_values = pipe_model.predict(inputs)
```

Run the graphing code cell

Submission 3 of 11: Progress mark (10)
Screen shot graph

Predictions.

Create a new text cell after the graphing cell and add in the following

```
### Plot pipelined linear regression model
Using height to predict weight
```

Create a code cell and add the following code

```
#heights to check
heights = [[0], [50], [100], [150], [175], [200]]
weights = pipe_model.predict(heights)
print('Input heights: ', heights)
print('Predicted weights are: ', weights)
```

1. We create a list of input lists. We just want to check that the deserialized model matchs the original over its basic range.
2. Feed those input sets into the model and make corresponding predictions.
3. Print the results for later verification.

Run the cell and you should see results that look something like.

Input heights: [[0], [50], [100], [150], [175], [200]]
Predicted weights are: [-166.29034128 -1.39346992 13.32603623 41.40189304 56.42986106
44.77557652]

Notice that as long as we are within the range of height from about 50 to 180 the variance is pretty good. But, it starts to balloon quickly outside that range.

Submission 4 of 11: Progress mark (10)
Screen shot predicted values

Submission 5 of 11: Analysis (5)
How would you decide on the appropriate limits on the input for valid results.

Pickle.

In the same cell add the following code.

```
# save the model to disk
import pickle
filename = 'finalized_model.sav'
pickle.dump(pipe_model, open(filename, 'wb'))
```

1. We keep the file name all lowercase. There is nothing special about the extension.
2. Dump takes two arguments, the first is the object to be serialized and the other is the file to save it in.
3. We open the file for writing in binary mode. It will overwrite any existing content.

Run the cell. Look in your files and verify that the file finalized_model.sav exists.

Submission 6 of 11: Progress mark (10)
Screen shot of directory listing showing the saved model

Restore the Model

We now want to read the model into a second notebook and verify that model matches on our test values.
Create a new notebook with the name **lab6compare.ipynb** and in the same cell add the following code.

```
import pickle
filename = 'finalized_model.sav'
loaded_model = pickle.load(open(filename, 'rb'))
```

1. We use the same name as before.
2. load takes a single argument, the file to read from in binary mode.

Check the Model

We can now run the check code from before and verify that the values match.

```
#heights to check
heights = [[0], [50], [100], [150], [175], [200]]
weights = pipe_model.predict(heights)
print('Input heights: ', heights)
print('Predicted weights are: ', weights)
```

Submission 7 of 11: Progress mark (10)
- Screen shot of predicted values results from the second notebook.
- Verify match

Here are some some important considerations when finalizing your machine learning models.

- **Python Version**. Take note of the python version. You almost certainly require the same major (and maybe minor) version of Python used to serialize the model when you later load it and deserialize it.
- **Library Versions**. The version of all major libraries used in your machine learning project almost certainly need to be the same when deserializing a saved model. This is not limited to the version of NumPy and the version of scikit-learn.

- **Manual Serialization**. You might like to manually output the parameters of your learned model so that you can use them directly in scikit-learn or another platform in the future. Often the algorithms used by machine learning algorithms to make predictions are a lot simpler than those used to learn the parameters and may be easy to implement in custom code that you have control over.

Building a Server

I have provided a python program that can be customized to work with one or more models

```
# Using flask to make an api
# import necessary libraries and functions
from logging import exception
from flask import Flask, jsonify, request, abort

# creating a Flask app
app = Flask(__name__)

# on the terminal type: curl http://127.0.0.1:5000/
# returns API when we use GET.
@app.route('/', methods = ['GET'])
def home():
    if(request.method == 'GET'):
        data = "API Example: /weight?height=130.3"
        return data
```

1. Flask will be used to create the web application. Jsonify allows us to create JSON. Request gives us access to the values bundled with the URL. Abort gives a way to respond with an error page.
2. App is going to be the name of a Flask object
3. The decorator with the @ attaches the URL and method to the function that follows
4. In this case we use the base URL.
5. If the method is GET, we return the data string.
6. Any other method can be caught later, but since we don't, it will result in a page not found.

```
import pickle
filename = 'finalized_model.sav'
loaded_model = pickle.load(open(filename, 'rb'))

# Use the model to predict the weight for the input height.
# the input height is a query parameter of the URL with GET
# on the terminal type: curl http://127.0.0.1:5000/weight?height=33.3
# this returns {"height": 33.3, "weight": -25.860609898265146}
@app.route('/weight', methods = ['GET'])
def disp():
    # get the query parameter as string and fail if not there
    height = request.args.get("height")
    if height is None:
        abort(400, 'Missing height query parameter')
    try:
        height = float(height)
    except ValueError:
        abort(400, 'Height query parameter not float')
```

1. We need to get the model. It needs to be in the same directory as this app.py program and we read it in using pickle. If this fails, the server app will crash on installation.
2. We have a new endpoint with query parameters. If we had multiple models, we can create an endpoint for each and use similar code.
3. We are looking for /weight with GET.
4. The comments give an example that you can run from the command line to hit this endpoint. We will just use a browser, but CURL (create URL) is there for you as well.
5. The associated function is disp and we are going to require that we get a height query parameter. If it is not there, we abort out with a 400 error code.
6. We have no control over what has been sent to us, so we need to convert it into a floating point value. This conversion either succeeds and we continue, or we abort.

```
    # we get back an array holding the predicted value/s
    try:
        weight = loaded_model.predict([[height]])[0]
    except Exception:
        abort(400, 'Model prediction failed')

    # put response in a dictionary
    response = {}
    response["height"] = height
    response["weight"] = weight
    return jsonify(response)

# driver function
if __name__ == '__main__':
    app.run(debug = True)
```

1. In the previous section, we validated the input, now we need to use the model to make a prediction.
2. I don't expect applying the model to fail, but I would rather not have the server crash.
3. The prediction from the model is going to come back in a list and will be the first (and only value).
4. I want my response to be a JSON object and the easiest way for me to do that is to create a python dictionary. The dictionary will have an entry for each of the input parameters and the target prediction. We don't have to provide the input values, but it is a courtesy for the end client.
5. Create an empty dictionary.
6. Map "height" to the input value.
7. Map "weight" to the prediction.
8. Use jsonify to turn the dictionary into JSON which we then return.
9. The last little bit indicates what to do if we run this file through a python interpreter (instead of doing some kind of import.) In this case, we run the flask application we just created in debug mode. We will want to change this for production.

Installing Flask

We will want to use pip to install the flask package into your virtual environment.

Activate the environment

Make sure you are in a command line shell in the top level directory of your virtual environment do the following:

MacOS/Linux

```
source ds-venv/bin/activate
```

Windows

Make sure you use `cmd` or allow scripts to be run from PowerShell

```
ds-venv\Scripts\Activate
```

You should now see that your terminal has changed:

```
C:\Users\choot>ds-venv\Scripts\Activate

(ds-venv) C:\Users\choot>
```

Installation

At this point, we can install the package.

```
pip install flask
```

Submission 8 of 11: Progress mark (10)
Screen shot of installation on command line

Starting the server locally

From the command line we can start the server. Make sure your current working directory is the folder containing the app.py file. Enter the line:

```
python3 app.py
```

Submission 9 of 8: Progress mark (10)
Screen shot of server start

Checking the App

Open a browser and enter each of the following urls:

http://127.0.0.1:5000/weight

http://127.0.0.1:5000/weight?height=33.3

http://127.0.0.1:5000/weight?height=0

http://127.0.0.1:5000/weight?height=50

http://127.0.0.1:5000/weight?height=100

http://127.0.0.1:5000/weight?height=150

http://127.0.0.1:5000/weight?height=175

http://127.0.0.1:5000/weight?height=200

Record the predictions from the server and compare to the values from the original lab notebook.

Submission 10 of 11: Progress mark (10)
List each of the responses

Submission 11 of 11: Analysis (5)
Do the values match up? Explain any discrepancies.

Cloud Deploy (Not required)

In the past, I have used Heroku to host my server because it had a free level that would allow you to have up to five rate limited servers running. Heroku recently removed that option. The lowest level available is now 7$ a month. As a hosting service it is pretty nice.

If you want a free option, Render will work. It is slow to spin up servers and requires your project to be stored on GITHub.

At the lower service levels you will typically be rate limited. But during development, this is good enough. You may run into memory limits for the server as well. The problem is that some kinds of models can grow with the size of the input to be quite large. Our model is very small at 1k, and training on more instances would not increase its size. If we use a model like SVM, the size could grow quite a bit depending on the number of support vectors the model decided that it needed.

Heroku Links

https://www.heroku.com/

https://devcenter.heroku.com/categories/python-support

Render Links

https://render.com

https://render.com/docs/deploy-flask