Pickling and Deploying Pipelines

Introduction

Now that you have learned about scikit-learn pipelines and model pickling, it's time to bring it all together in a professional ML workflow!

Objectives

In this lesson you will:

- Understand the purpose of deploying a machine learning model
- Understand the cloud function approach to model deployment
- Pickle a scikit-learn pipeline
- Create a cloud function

Model Deployment

Previously when we covered pickling, we introduced the idea of model ***persistence*** -- essentially that if you serialize a model after training it, you can later load the model and make predictions without wasting time or computational resources re-training the model.

In some contexts, model persistence is all you need. For example, if the end-user of your model is a data scientist with a full Python environment setup, they can launch up their own notebook, call `.load` on the model file, and start making predictions.

Model ***deployment*** goes beyond model persistence to allow your model to be used in many more contexts. Here are just a few examples:

- A mobile app that uses a model to power its game AI
- A website that uses a model to decide which ads to serve to a given viewer
- A CRM (customer relationship management) platform that uses a model to decide when to send out coupons

In order for these applications to work, your model needs to use a deployment strategy that is more complex than simply pickling a Python model.

Model Deployment with Cloud Functions

A cloud function is a very popular way to deploy a machine learning model. When you deploy a model as a cloud function, that means that you are setting up an ***HTTP API backend***. This is the same kind of API that you have previously queried using the `requests` library.

The advantage of a cloud function approach is that the cloud provider handles the actual web server maintenance underpinning the API, and you just have to provide a small amount of function code. If you need a lot of customization, you may need to write and deploy an actual web server, but for many use cases you can just use the cloud function defaults.

When a model is deployed as an API, that means that **it can be queried from any language** that can use HTTP APIs. This means that even though you wrote the model in Python, an app written in Objective C or a website written in JavaScript can use it!

Creating a Cloud Function

Let's go ahead and create one! We are going to use the format required by Google Cloud Functions (documentation here (https://cloud.google.com/functions)).

This is by no means the only platform for hosting a cloud function -- feel free to investigate AWS Lambda or Azure Functions or other cloud provider options! However Google Cloud Functions are a convenient option because their free tier allows up to 2 million API calls per day, and they promise not to charge for additional API calls unless given authorization. This is useful when you're learning how to deploy models and don't want to spend money on cloud services.

Components of a Cloud Function for Model Deployment

In order to deploy a model, you will need:

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

Our Pipeline

Let's say the model we have developed is a multi-class classifier trained on the iris dataset (https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_iris.html#sklearn.datasets.load_iris) from scikit-learn.

In [1]:
```
import pandas as pd
from sklearn.datasets import load_iris

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

Out[1]:

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	class
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0
...	...	...	...	...	...
145	6.7	3.0	5.2	2.3	2
146	6.3	2.5	5.0	1.9	2
147	6.5	3.0	5.2	2.0	2
148	6.2	3.4	5.4	2.3	2
149	5.9	3.0	5.1	1.8	2

150 rows × 5 columns

In [2]:
```
X.describe()
```

Out[2]:

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
count	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.057333	3.758000	1.199333
std	0.828066	0.435866	1.765298	0.762238
min	4.300000	2.000000	1.000000	0.100000
25%	5.100000	2.800000	1.600000	0.300000
50%	5.800000	3.000000	4.350000	1.300000
75%	6.400000	3.300000	5.100000	1.800000
max	7.900000	4.400000	6.900000	2.500000

And let's say we are using logistic regression, with default regularization applied.

As you can see, even with such a simple dataset, we need to scale the data before passing it into the classifier, since the different features do not currently have the same scale. Let's go ahead and use a pipeline for that:

In [3]:
```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression

pipe = Pipeline([
    ("scaler", StandardScaler()),
    ("model", LogisticRegression())
])

pipe.fit(X, y)
```

Out[3]:
```
Pipeline(steps=[('scaler', StandardScaler()), ('model', LogisticRegression())])
```

Now the pipeline is ready to make predictions on new data!

In the example below, we are sending in X values from a record in the training data. We know that the classification *should* be 0, so this is a quick check to make sure that the model works and we are getting the results we expect:

In [4]:
```
example = [[5.1, 3.5, 1.4, 0.2]]
pipe.predict(example)[0]
```

Out[4]:
```
0
```

It worked!

(Note that when you call `.predict` on a pipeline, it is actually transforming the input data then making the prediction. You do not need to apply a separate `.transform` step.)

Pickling Our Pipeline

In this case, because raw data needs to be preprocessed before our model can use it, we'll pickle the entire pipeline, not just the model.

Sometimes you will see something referred to as a "pickled model" when it's actually a pickled pipeline, so we'll use that language interchangeably.

First, let's import the `joblib` library:

In [5]:
```
import joblib
```

Then we can serialize the pipeline into a file called `model.pkl` .

(Recall that `.pkl` is a conventional file ending for a pickled scikit-learn model. This blog post (https://towardsdatascience.com/guide-to-file-formats-for-machine-learning-columnar-training-inferencing-and-the-feature-store-2e0c3d18d4f9) covers many other file formats that you might see that use different libraries.)

In [6]:
```
with open("model.pkl", "wb") as f:
    joblib.dump(pipe, f)
```

That's it! Sometimes you will also see approaches that pickle the model along with its current metrics and possibly some test data (in order to confirm that the model performance is the same when un-pickled) but for now we'll stick to the most simple approach.

Creating Our Function

The serialized model is not sufficient on its own for the HTTP API server to know what to do. You also need to write the actual Python function for it to execute.

Let's write a function and test it out. In this case, let's say the function:

- takes in 4 arguments representing the 4 features
- returns a dictionary with the format `{"predicted_class": <value>}` where `<value>` is 0, 1, or 2

In [7]:
```
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 = predictions[0]

    return {"predicted_class": prediction}
```

Now let's test it out!

In [8]:
```
iris_prediction(5.1, 3.5, 1.4, 0.2)
```

Out[8]:
```
{'predicted_class': 0}
```

The specific next steps needed to incorporate this function into a cloud function platform will vary. For Google Cloud Functions specifically, it looks like this. All of this code would need to be written in a file called `main.py` :

In [9]:
```
import json
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}

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)
```

(Unusually for a `.py` file, we don't need to include an `if __name__ == "__main__":` statement in the file. This is due to the particular configuration of Google Cloud Functions, which operations a web server behind the scenes. Instead, if you want to deploy the function, you'll need to configure it in the console so that the `predict` function will be invoked.)

Creating Our Requirements File

One last thing we need before we can upload our cloud function is a requirements file. As we have developed this model, we have likely been using some massive environment like `learn-env` that includes lots of packages that we don't actually need.

Let's make a file that specifies only the packages we need. Which ones are those?

scikit-learn

We used scikit-learn to build our pickled model. (Technically, a model pipeline.) Let's figure out what exact version it was:

In [10]:
```
import sklearn
sklearn.__version__
```

Out[10]:
```
'0.23.2'
```

joblib

We also used joblib to serialize the model. We'll repeat the same step:

In [11]:
```
joblib.__version__
```

Out[11]:
```
'0.17.0'
```

Creating the File

At the time of this writing, we are using scikit-learn 0.23.2 and joblib 0.17.0. (If you see different numbers there, that means we have updated the environment, so use those numbers instead!) We create a file called `requirements.txt` containing these lines, with pip-style versions (documentation here (https://pip.pypa.io/en/stable/reference/requirements-file-format/#requirements-file-format)):

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

Putting It All Together

Now we have:

1. `model.pkl` (a pickled model file)
2. `main.py` (a Python file defining the function)
3. `requirements.txt` (a requirements file)

Copies of each of these files are available in this repository.

If you want to deploy these on Google Cloud Functions, you'll want to combine them all into a single zipped archive. For example, to do this with the `zip` utility on Mac or Linux, run this command in the terminal:

```
zip archive model.pkl main.py requirements.txt
```

That will create an archive called `archive.zip` which can be uploaded following these instructions (https://cloud.google.com/functions/docs/deploying/console). An already-created `archive.zip` is available in this repository if you just want to practice following the Google Cloud Function instructions.

You will want to specify an executed function of `predict` and also select the checkbox for "Allow unauthenticated invocations" if you want to make a public API.

Then the code to test out your deployed function would be something like this, where you replace the `url` value with your actual URL.

```python
import requests
response = requests.post(
    url="https://<name here>.cloudfunctions.net/function-1",
    json={"sepal_length": 5.1, "sepal_width": 3.5, "petal_length": 1.4, "petal_width": 0.2}
)
response
```

Summary

In this lesson, we discussed the purpose of model deployment, and cloud functions in particular. Then we walked through the process of pickling a scikit-learn pipeline and using it in a deployed cloud function.