Published in Towards Data Science

Zhou (Joe) Xu Follow

Nov 2, 2021 · 9 min read · Listen

Build REST API for Machine Learning Models using Python and Flask-RESTful

A minimal working example for data scientists to follow along

Photo by Marcin Jozwiak on Unsplash

Note: All the code and results mentioned in this post can be accessed from my GitHub page.

1. Introduction

As data scientists, we need to not only build models but also deploy them and make them useful in complete systems. In many cases, we need to collaborate with other developers, and it is almost always a good idea to keep the model in a separate location and use an API to serve the model to other applications.

An Application Programming Interface (API) is a web service that grants access to specific data and methods that other applications can access via standard HTTP protocols, while REpresentational State Transfer (REST) is one of the most popular architectural styles of APIs for web service. On the other hand, Python is the most favorable language among data scientists and there are two popular web frameworks: Django and Flask. Compared to Django, Flask is famous for its lightweight and rapid development. It also has many extensions to add specific functionalities to the vanilla Flask application, and Flask-RESTful is a perfect choice to make it really easy to build a fully functioning REST API.

Here I am going to walk you through how to build a minimum viable REST API using Flask-RESTful with an example and all codes you need to follow along. There are 3 main sections in this post:


- **Test the API**: Use the model to make predictions by calling the API

2. Environment Setup

Since it is a minimal example, it does not require many packages. You can either use `conda` or `pip`. For 2.1 to 2.3 below, choose any of the methods that fit your interest. After finishing any of these, you can jump directly to section 3.

2.1 Using conda

If you have conda installed, you can run:

```
conda install -c conda-forge numpy scikit-learn flask-restful requests
```

2.2 Using pip

if you don't have conda installed, `pip` can also do it by running:

```
pip install numpy scikit-learn flask-restful requests
```

2.3 Create a conda environment for it

Sometimes it is beneficial to keep different environments for different projects, so if you want to create another virtual environment, feel free to do it by running:

```
conda create --name flask_api -conda-forge numpy scikit-learn flask-restful requests
```

It will create another environment called `flask_api` with these packaged installed, and you can switch to the new environment by running:

```
conda activate flask_api
```

You can also create virtual environments using `pip`, but I will skip that as you can easily find tutorials anywhere online.

3. Train a minimal ML model


this section, which can be downloaded here as well. We name it `train_model.py` , and you can just run using `python train_model.py`

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
import pickle

# Load Iris data
iris = load_iris()

# Split into train and test sets
X_train, X_test, y_train, y_test = \
    train_test_split(iris['data'], iris['target'], random_state=12)

# Train the model
clf = RandomForestClassifier(random_state=12)
clf.fit(X_train, y_train)

# Make prediction on the test set
y_predict = clf.predict(X_test)
print(y_predict)

# Save model
with open('model.pickle', 'wb') as f:
    pickle.dump(clf, f)
```

restapi_train_model.py hosted with ❤ by **GitHub** **view raw**

If you are familiar with machine learning and have had some hands-on experience with `scikit-learn` , then the code should make sense to you.

Basically, we first load a toy dataset `iris` , as shown below:

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

150 rows × 5 columns

There are 150 records. Each of them has 4 features and is labeled in one of the 3 classes (setosa, versicolor, and virginica).

In this example, we directly train the model and skip the fine-tuning part, and then use the model to make predictions on the test set:

```
[0 2 0 1 2 2 2 0 2 0 1 0 0 0 1 2 2 1 0 1 0 1 2 1 0 2 2 1 0 0 0 1 2 0 2
0 1 1]
```

Where the numbers represent each of the 3 classes. By comparing with the true labels, we find the accuracy is 97% and the confusion matrix is shown below:

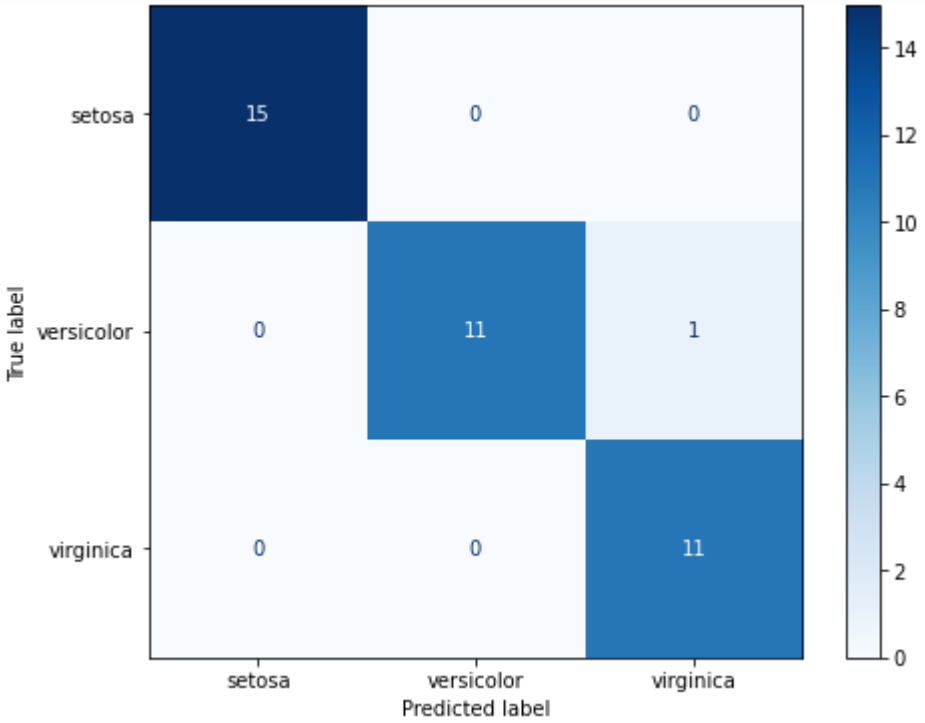


Figure 1: Confusion matrix of the model prediction on the test set (directly from the model object)

Our performance is good enough (probably too good in real life). We then save the trained model by serializing it in a pickle file.

4. Build the REST API

4.1 Understand the code

Now, with a simple model ready to be served, let's build a REST API. Again, below is the code and I will explain it step by step. You can also download it from my GitHub. We name this one `api.py`

```
from flask import Flask, jsonify
from flask_restful import Api, Resource, reqparse
import pickle
import numpy as np
import json

app = Flask(__name__)
```

```
13
14  # Define how the api will respond to the post requests
15  class IrisClassifier(Resource):
16      def post(self):
17          args = parser.parse_args()
18          X = np.array(json.loads(args['data']))
19          prediction = model.predict(X)
20          return jsonify(prediction.tolist())
21
22  api.add_resource(IrisClassifier, '/iris')
23
24  if __name__ == '__main__':
25      # Load model
26      with open('model.pickle', 'rb') as f:
27          model = pickle.load(f)
28
29      app.run(debug=True)
```

restapi_api.py hosted with ❤ by **GitHub** **view raw**

```
app = Flask(__name__)
api = Api(app)
```

First, we import modules, and the two lines above should almost always exist on the top of the code to initiate the flask app and API.

```
# Create parser for the payload data
parser = reqparse.RequestParser()
parser.add_argument('data')
```

The code chunk above creates a request parser to parse arguments that will be sent with the request. In our case, since we serve the ML model, we usually send the data in a JSON serialized format and name the key as `data` , so we ask the parser to look for the data contained in the request.


```
        args = parser.parse_args()
        X = np.array(json.loads(args['data']))
        prediction = model.predict(X)
        return jsonify(prediction.tolist())

api.add_resource(IrisClassifier, '/iris')
```

Then we create a class called `IrisClassifier` by inheriting from the `Resource` class we imported from `flask-restful` , which already defines various methods of handling different types of requests, which include `get` `post` `put` `delete` and so on. Our goal is to rewrite the `post` method and tell it how to use the model to make predictions on the given data. Since the purpose of our API is to serve ML models, we only need the `post` method and not to worry about the others.

Within the `post` method, we first call the parser just defined to get the arguments. The data (originally np.array) is serialized into string format within the JSON, so we use `json.loads` to deserialize it into a list and then back into a NumPy array. The prediction part stays the same as how `scikit-learn` models always do. In the final step, we need to make the predicted labels (np.array) back to list and call the `jsonify` function imported from `flask` to serialize it again so that they will be returned back to the application in a proper format.

After the class is defined, we add our `IrisClassifier` (essentially a modified `resource` class) into our API, along with the relative route to access it, in our case it's `/iris` . The whole URL for our ML model will then be something like `http://127.0.0.1:5000/iris` if we just run our flask locally.

```
if __name__ == '__main__':
    # Load model
    with open('model.pickle', 'rb') as f:
        model = pickle.load(f)

    app.run(debug=True)
```


In the final part of the code in `api.py`, we load the model saved from last section so that the app knows where to get the model if any prediction is requested. Then we run the flask app in a debug mode, where it just allows arbitrary code to be executed directly in the browser if any error happens.

4.2 Spin up the REST API

If you have the code saved as `api.py`, you can run it using:

```
python api.py
```

There will be messages showing like:

```
 * Serving Flask app "api" (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: on
  * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
  * Restarting with stat
  * Debugger is active!
  * Debugger PIN: 117-609-315
```

Then you are good to go!

5. Test the API

If you have followed along with previous sections, you should have the REST API running on your local machine. Now it's time to test and see if it works. Again, below is all the code (minimal) you need to copy, paste, and run. For more complete code and results, please see the Jupyter Notebook, or for the code below, you can access it here.

I will then explain the code step by step. Let's name it `test_api.py` and you can run it by calling `python test_api.py`


```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

# Load data
iris = load_iris()

# Split into train and test sets using the same random state
X_train, X_test, y_train, y_test = \
    train_test_split(iris['data'], iris['target'], random_state=12)

# Serialize the data into json and send the request to the model
payload = {'data': json.dumps(X_test.tolist())}
y_predict = requests.post('http://127.0.0.1:5000/iris', data=payload).json()

# Make array from the list
y_predict = np.array(y_predict)
print(y_predict)
```

restapi_test_api.py hosted with ❤ by **GitHub** **view raw**

```
# Load data
iris = load_iris()

# Split into train and test sets using the same random state
X_train, X_test, y_train, y_test = \
    train_test_split(iris['data'], iris['target'], random_state=12)
```

After importing the relevant modules, the first thing to do is get a subset of data (features only) to test the predictions returned by the API. For easier comparison, we can just use the same test set that we split in section 3. We can load the iris dataset and use the same random state (12) to get the exact subset of data `X_test` .

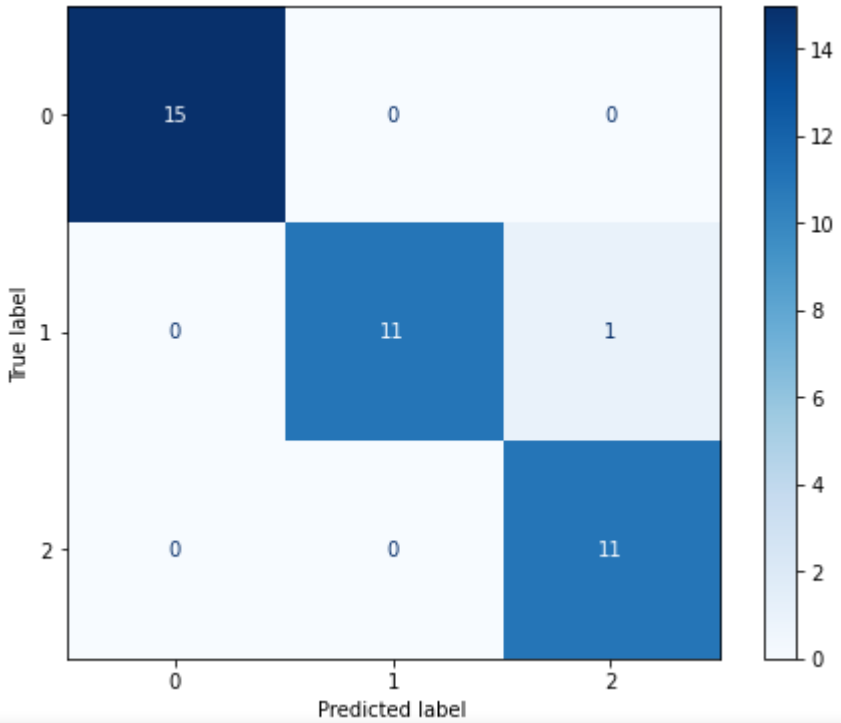
```
# Serialize the data into json and send the request to the model
payload = {'data': json.dumps(X_test.tolist())}
y_predict = requests.post('http://127.0.0.1:5000/iris',
data=payload).json()
```


Then, we create a `payload` variable to serialize the list of `X_test` into JSON format with key as `data` , just like how we defined in `api.py` , so that our API knows where to find the input data. Then we post the payload to the endpoint we defined and get the prediction results.

```
# Make array from the list
y_predict = np.array(y_predict)
print(y_predict)
```

The final step is optional, but we can make the data back to NumPy array and print out the predictions, which is exactly the same as we get before in section 3 when we just create the model. The confusion matrix is exactly the same as well:

```
[0 2 0 1 2 2 2 0 2 0 1 0 0 0 1 2 2 1 0 1 0 1 2 1 0 2 2 1 0 0 0 1 2 0 2
0 1 1]
```



On the REST API end, after the `post` method was used, we can see the message showing Response of 200, meaning the request has been processed successfully:

```
127.0.0.1 - - [01/Nov/2021 17:01:42] "POST /iris HTTP/1.1" 200 -
```

Therefore, we have tested the REST API that we just built and got exactly the same results from which were returned by the original model object. Our API works!

Conclusion

We just covered the minimal knowledge and steps needed to build a fully functioning REST API using Python and Flask-RESTFul. If we just look at the API building part (Section 4), it is indeed very simple — below 30 lines of code and most of them are just standard statements. It is extremely easy to modify the code as well: all you need to do to tailor it for your model is to replace the model you want to use, specify how the `post` method work, and give a proper route to access the API.

To further make use of the REST API we just built to serve our Machine Learning models to other applications, we can deploy it on a server, or using any of the popular cloud web hosting services: AWS, Google Cloud, Heroku, etc. Through proper configurations, anyone with permission will then be able to access our model from anywhere in the world.

Thank you for reading! If you like this article, please **follow my channel** (really appreciate it 🙏). I will keep writing to share my ideas and projects about data science. Feel free to contact me if you have any questions.

About Me

I am a data scientist at Sanofi. I embrace technology and learn new skills every day. You are welcome to reach me from Medium Blog, LinkedIn, or GitHub. My opinions are my own and not the views of my employer.


- Loan Default Prediction for Profit Maximization
- Loan Default Prediction with Berka Dataset
- Understanding Sigmoid, Logistic, Softmax Functions, and Cross-Entropy Loss (Log Loss) in Classification Problems

References

[1] How to Use the Python Requests Module with REST API: https://www.nylas.com/blog/use-python-requests-module-rest-apis/

[2] Python REST API Tutorial — Building a Flask REST API: https://www.youtube.com/watch?v=GMppyAPbLYk&t=3769s&ab_channel=TechWithTim