1. Create model and sabe as iris.mdl
2. Create web api app.py
3. Create program to call web api (app.py) with parameters

When making predictions, we will have four input parameters:

* sepal length,
* sepal width,
* petal length,
* and finally, petal width.

Those will help to decide which type of iris flower the input is.

I used the scikit-learn implementation of a simple KNN (K-nearest neighbor) algorithm to predict the type of iris:

# model.py

from sklearn import datasets

from sklearn.model\_selection import train\_test\_split

from sklearn.neighbors import KNeighborsClassifier

from sklearn.metrics import accuracy\_score

from sklearn.externals import joblib

import numpy as np

def train(X,y):

# train test split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3)

knn = KNeighborsClassifier(n\_neighbors=1)

# fit the model

knn.fit(X\_train, y\_train)

preds = knn.predict(X\_test)

acc = accuracy\_score(y\_test, preds)

print(f'Successfully trained model accuracy of {acc:.2f}')

return knn

if \_\_name\_\_ == '\_\_main\_\_':

iris\_data = datasets.load\_iris()

X = iris\_data['data']

y = iris\_data['target']

labels = {0 : 'iris-setosa',

1 : 'iris-versicolor',

2 : 'iris-virginica'}

# rename integer labels to actual flower names

y = np.vectorize(labels.\_\_getitem\_\_)(y)

modelo\_knn = train(X,y)

# serialize model

joblib.dump(modelo\_knn, 'iris.mdl')

As you can see, I trained the model with 70% of the data and then validated with 30% out of sample test data.

After the model training has taken place, I serialize the model with the *joblib* library.

Joblib is an alternative to pickle, which preserves the persistence of *scikit* estimators, which include a large number of *numpy* arrays (such as the KNN model, which contains all the training data).

After the file is saved as a *joblib* file it can be loaded again later in our application.

**The API with Python and Flask**

This class will be a child class of the *Flask-RESTful* class Resource. This lets our class inherit the respective class methods and allows *Flask* to do the work behind your API without needing to implement everything.

To build an API from our trained model with package *Flask* and *Flask-RESTful*.

Further, we import joblib to load our model and numpy to handle the input and output data.

A new script, namely app.py

-Import flask

-Train model in the same folder as the script

# app.py

from flask import Flask

from flask\_restful import Api, Resource, reqparse

from sklearn.externals import joblib

import numpy as np

APP = Flask(\_\_name\_\_)

API = Api(APP)

#modelo previamente salvado

IRIS\_MODEL = joblib.load('iris.mdl')

class Predict(Resource):

@staticmethod

def post():

parser = reqparse.RequestParser()

parser.add\_argument('petal\_length')

parser.add\_argument('petal\_width')

parser.add\_argument('sepal\_length')

parser.add\_argument('sepal\_width')

args = parser.parse\_args() # creates dict

X\_new = np.fromiter(args.values(), dtype=float)

# convert input to array

out = {'Prediction': IRIS\_MODEL.predict([X\_new])[0]}

return out, 200

API.add\_resource(Predict, '/predict')

if \_\_name\_\_ == '\_\_main\_\_':

APP.run(debug=True, port='1080')

Notes:

1)post

The post method allows to send a body (in JSON) along with the default API parameters.

2)body

The body is not delivered directly in the URL, but as a text, it is parsed with RequestParser.add\_argument() method

3) Convert the input parsed into array

4) return the prediction of our model as JSON.

5)endpoint

API.add\_resource(Predict, '/predict')

The '/predict' , is the API endpoint. localhost/predict

4)return 200

return out, 200

It means that the request has been received sucessfully.

**Run the API**

To run the app, simply open a terminal in the same directory as your app.py script and run this command.

python run app.py

You should now get a notification, that the API runs on your localhost in the port you defined. There are several ways of accessing the API once it is deployed.

For debugging and testing purposes, I usually use tools like *[Postman](https://www.postman.com/" \t "_blank)*. We can also access the API from within a Python application, just like another user might want to do to use your model in their code.

We use the *[requests](https://requests.readthedocs.io/en/master/" \t "_blank)* module, by first defining the URL to access and the body to send along with our HTTP request:

**INVOKE WEB API**

import requests

url = 'http://127.0.0.1:1080/predict'

# localhost + port / endpoint

body = {

"petal\_length": 2,

"sepal\_length": 2,

"petal\_width": 0.5,

"sepal\_width": 3

}

response = requests.post(url, data=body)

response.json()

The output should look something like this:

Out[1]: {'Prediction': 'iris-versicolor'}

For the next step, maybe try [securing your APIs](https://www.statworx.com/en/content-hub/blog/two-patterns-to-secure-rest-apis/" \t "_blank)? If you are interested in learning how to build an API with R, you should check out [this post](https://www.statworx.com/en/content-hub/blog/how-to-create-rest-apis-with-r-plumber/" \t "_blank).