import pandas as pd

import numpy as np

# Load the dataset

url = "http://s3.amazonaws.com/assets.datacamp.com/course/Kaggle/train.csv"

df = pd.read\_csv(url)

include = ['Age', 'Sex', 'Embarked', 'Survived'] # Only four features

df\_ = df[include]

"Sex" and "Embarked" are categorical features with non-numeric values and that is why they require some numeric transformations.

“Age” feature has missing values. These values can be imputed with a summary statistic such as median or mean.

Scikit-learn treats the cell values which do not contain anything as NaNs. Here, you will merely replace NaNs with 0, and you will write a helper function for that.

categoricals = []

for col, col\_type in df\_.dtypes.iteritems():

     if col\_type == 'O':

          categoricals.append(col)

     else:

          df\_[col].fillna(0, inplace=True)

The above lines of code does the following:

* Iterates over all the columns in the dataframe df and appending the columns (with non-numeric values) to a list categorical.
* If the columns do not have non-numeric values (which is only Age in this case), then it checks if it has missing values or not and fills them with 0.

**Filling NaNs with a single value may have unintended consequences, especially if the amount that you’re replacing NaNs with is within the observed range for the numeric variable. Since zero is not an observed and legitimate age value you are not introducing bias, you would have if you used say 36!** - [Source](https://towardsdatascience.com/a-flask-api-for-serving-scikit-learn-models-c8bcdaa41daa)

Now that you handled the missing values and separated the non-numeric columns you are ready to convert them to numeric ones. You will do this by using [One Hot Encoding](http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html" \l "sklearn.preprocessing.OneHotEncoder). Pandas provides a simple method get\_dummies() for creating OHE variables for a given dataframe.

df\_ohe = pd.get\_dummies(df\_, columns=categoricals, dummy\_na=True)

When you use OHE, a new column is created for every column/value combination, in a column\_value format. For example - for the “Embarked” variable, OHE will produce “Embarked\_C”, “Embarked\_Q”, “Embarked\_S”, and “Embarked\_nan”.

Now that you’ve successfully preprocessed your dataset, you’re ready to train the machine learning model. You will use a *Logistic Regression classifier* for this.

from sklearn.linear\_model import LogisticRegression

dependent\_variable = 'Survived'

x = df\_ohe[df\_ohe.columns.difference([dependent\_variable])]

y = df\_ohe[dependent\_variable]

lr = LogisticRegression()

lr.fit(x, y)

LogisticRegression(C=1.0, class\_weight=None, dual=False, fit\_intercept=True,

          intercept\_scaling=1, max\_iter=100, multi\_class='ovr', n\_jobs=1,

          penalty='l2', random\_state=None, solver='liblinear', tol=0.0001,

          verbose=0, warm\_start=False)

You have built your machine learning model. You will now save this model. Technically speaking, you will serialize this model. In Python, you call this **Pickling**.

**Saving the model: Serialization and Deserialization**

You will use sklearn’s joblib for this.

import joblib

joblib.dump(lr, 'model.pkl')

['model.pkl']

The Logistic Regression model is now persisted. You can load this model into memory with a single line of code. Loading the model back into your workspace is known as Deserialization.

lr = joblib.load('model.pkl')

You’re now ready to use Flask to serve your persisted model. You have already seen how minimalistic Flask is to get started with.

**Creating an API from a machine learning model using Flask**

For serving your model with Flask, you will do the following two things:

* Load the already persisted model into memory when the application starts,
* Create an API endpoint that takes input variables, transforms them into the appropriate format, and returns predictions.

More specifically, your sample input to the API will look like the following:

[

{"Age": 85, "Sex": "male", "Embarked": "S"},

{"Age": 24, "Sex": '"female"', "Embarked": "C"},

{"Age": 3, "Sex": "male", "Embarked": "C"},

{"Age": 21, "Sex": "male", "Embarked": "S"}

]

(which is a JSON list of inputs)

and your API will output like the following:

{"prediction": [0, 1, 1, 0]}

The predictions denote the survival statuses where 0 represents No and 1 represents Yes.

JSON stands for JavaScript Object Notation, and it is one of the most widely used data interchange formats. If you need a quick introduction to it, please follow [these tutorials](https://www.w3schools.com/js/js_json_intro.asp).

Let's write a function predict() which will do:

* Load the persisted model into memory when the application starts,
* Create an API endpoint that takes input variables, transforms them into the appropriate format, and returns predictions.

You have already seen how to load a persisted model. Now, you will focus on how you can use it for predicting the survival status upon receiving inputs.

from flask import Flask, jsonify

app = Flask(\_\_name\_\_)

@app.route('/predict', methods=['POST'])

def predict():

     json\_ = request.json

     query\_df = pd.DataFrame(json\_)

     query = pd.get\_dummies(query\_df)

     prediction = lr.predict(query)

     return jsonify({'prediction': list(prediction)})

Fantastic! But you have got a little problem here.

The function that you wrote would only work under conditions where the incoming request contains all possible values for the categorical variables which may or may not be the case in real-time. If the incoming request does not include all possible values of the categorical variables then as per the current method definition of predict(), get\_dummies() would generate a dataframe that has fewer columns than the classifier excepts, which would result in a runtime error.

To solve this problem, you will persist the list of columns during model training as well. You can serialize any Python object into a .pkl file. You will use joblib in the same way as previously.

(Keep that in mind, as discussed earlier it is always better to do all the server level coding in a text editor and then run it from a terminal)

model\_columns = list(x.columns)

joblib.dump(model\_columns, 'model\_columns.pkl')

['model\_columns.pkl']

As you have persisted the list of columns already, you can just handle the missing values at the time of prediction. You will have to load model columns when the application starts.

@app.route('/predict', methods=['POST']) # Your API endpoint URL would consist /predict

def predict():

    if lr:

        try:

            json\_ = request.json

            query = pd.get\_dummies(pd.DataFrame(json\_))

            query = query.reindex(columns=model\_columns, fill\_value=0)

            prediction = list(lr.predict(query))

            return jsonify({'prediction': prediction})

        except:

            return jsonify({'trace': traceback.format\_exc()})

    else:

        print ('Train the model first')

        return ('No model here to use')

You included all the required elements in the "/predict" API, and now you just need to write the main class.

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

try:

port = int(sys.argv[1]) # This is for a command-line argument

except:

port = 12345 # If you don't provide any port then the port will be set to 12345

lr = joblib.load(model\_file\_name) # Load "model.pkl"

print ('Model loaded')

model\_columns = joblib.load(model\_columns\_file\_name) # Load "model\_columns.pkl"

print ('Model columns loaded')

app.run(port=port, debug=True)

Your API now ready to be hosted. But before going any further, let's recap all that you did till this point:

**Putting it all together:**

* You loaded Titanic dataset and selected the four features.
* You did the necessary data preprocessing.
* You built a Logistic Regression classifier and serialized it.
* You also serialized all the columns from training as a solution to the less than expected number of columns is to persist the list of columns from training.
* You then wrote a simple API using Flask that would predict if a person had survived in the shipwreck given there age, sex and embarked information.

Let's put all the code in one place so that you don't miss out on anything. Also, it is a good programming practice if you separate your Logistic Regression model code and your Flask API code into separate .py files.

So your model.py should look like the following:

import pandas as pd

import numpy as np

# Load the dataset

url = "http://s3.amazonaws.com/assets.datacamp.com/course/Kaggle/train.csv"

df = pd.read\_csv(url)

include = ['Age', 'Sex', 'Embarked', 'Survived'] # Only four features

df\_ = df[include]

# Data Preprocessing

categoricals = []

for col, col\_type in df\_.dtypes.iteritems():

     if col\_type == 'O':

          categoricals.append(col)

     else:

          df\_[col].fillna(0, inplace=True)

df\_ohe = pd.get\_dummies(df\_, columns=categoricals, dummy\_na=True)

# Logistic Regression classifier

from sklearn.linear\_model import LogisticRegression

dependent\_variable = 'Survived'

x = df\_ohe[df\_ohe.columns.difference([dependent\_variable])]

y = df\_ohe[dependent\_variable]

lr = LogisticRegression()

lr.fit(x, y)

# Save your model

#from sklearn.externals

import joblib

joblib.dump(lr, 'model.pkl')

print("Model dumped!")

# Load the model that you just saved

lr = joblib.load('model.pkl')

# Saving the data columns from training

model\_columns = list(x.columns)

joblib.dump(model\_columns, 'model\_columns.pkl')

print("Models columns dumped!")

Your api.py should look like the following:

from flask import Flask, request, jsonify

#from sklearn.externals

import joblib

import traceback

import pandas as pd

import numpy as np

# Your API definition

app = Flask(\_\_name\_\_)

@app.route('/predict', methods=['POST'])

def predict():

    if lr:

        try:

            json\_ = request.json

            print(json\_)

            query = pd.get\_dummies(pd.DataFrame(json\_))

            query = query.reindex(columns=model\_columns, fill\_value=0)

            prediction = list(lr.predict(query))

            return jsonify({'prediction': str(prediction)})

        except:

            return jsonify({'trace': traceback.format\_exc()})

    else:

        print ('Train the model first')

        return ('No model here to use')

if \_\_name\_\_ == '\_\_main\_\_':

    try:

        port = int(sys.argv[1]) # This is for a command-line input

    except:

        port = 12345 # If you don't provide any port the port will be set to 12345

    lr = joblib.load("model.pkl") # Load "model.pkl"

    print ('Model loaded')

    model\_columns = joblib.load("model\_columns.pkl")

    # Load "model\_columns.pkl"

    print ('Model columns loaded')

    app.run(port=port, debug=True)

Now you will test this API in an API client called [Postman](https://www.getpostman.com/).

Just make sure that model.py and api.py are in the same directory and also make sure that you have compiled them both before testing. Refer to the following snapshot of the terminal which is taken after both the .py files were compiled successfully.

![Text

Description automatically generated]()

If all of your files were compiled successfully, the following should be the directory structure:  
![A picture containing table

Description automatically generated]()

**Note**: The IPYNB file is optional though.

**Testing your API in Postman**

In order to test your API, you will need some kind of API client. Postman is undoubtedly one of the best ones out there. You can easily download Postman from the above link.

The Postman interface looks like the following if you downloaded the latest one:

![Graphical user interface, text, application

Description automatically generated]()

After you have started the Flask server successfully, you then need to enter the right URL with the correct port number in Postman. It should look similar to the following:

![Graphical user interface, text, application, email

Description automatically generated]()

Congratulations! You just built your first ever machine learning API.

Your API can predict if a passenger survived the Titanic shipwreck given there age, sex and embarked information. Now, your friend may call it from there front-end code and process the output of the API into something fascinating.

**Taking it further:**

In this tutorial, you covered one of the most vital industry demanding skills of a full-stack Data Scientist, i.e. building an API from a machine learning model. Although the API is straightforward, it is always better to start with the simplest of things so that you know the know-how in the details.

You can do a lot more in order to improve this. Possible options you might want to consider:

* Write a "/train" API which would train a Logistic Regression classifier with the data.
* Code a Neural Network model using keras and build an API out of it.
* Host your API on Cloud so that it can be consumed.
* For taking things to more advanced levels, you might refer to [this Machine Learning Mastery blog](https://machinelearningmastery.com/deploy-machine-learning-model-to-production/) which discusses several industry graded approaches.

The possibilities and opportunities are enormous here. You just need to carefully select the ones which are the most suitable for you.

If you would like to learn more about Machine Learning in Python, take DataCamp's [Preprocessing for Machine Learning in Python](https://www.datacamp.com/courses/preprocessing-for-machine-learning-in-python) course.

**References:**

The following are some references that were taken while writing this blog:

* Flask: Building Python Web Services
* [A Quora thread on the topic](https://www.quora.com/How-do-I-deploy-Machine-Learning-Models-as-an-API)