5 Deploying machine learning models

This chapter covers

- Saving models with Pickle
- Serving models with Flask
- Managing dependencies with Pipenv
- Making the service self-contained with Docker
- Deploying it to the cloud using AWS Elastic Beanstalk

As we continue to work with machine learning techniques, we'll keep using the project we already started: churn prediction. In chapter 3, we used Scikit-learn to build a model for identifying churning customers. After that, in chapter 4, we evaluated the quality of this model and selected the best parameter `C` using cross-validation.

We already have a model that lives in our Jupyter Notebook. Now we need to put this model into production, so other services can use the model to make decisions based on the output of our model.

In this chapter, we cover *model deployment* : the process of putting models to use. In particular, we see how to package a model inside a web service, so other services can use it. We also see how to deploy the web service to a production-ready environment.

5.1 Churn-prediction model

To get started with deployment we use the model we trained previously. First, in this section, we review how we can use the model for making predictions, and then we see how to save it with Pickle.

5.1.1 Using the model

To make it easier, we can continue the same Jupyter Notebook we used for chapters 3 and 4.

Let's use this model to calculate the probability of churning for the following customer:

```
customer = {
    'customerid': '8879-zkjof',
    'gender': 'female',
    'seniorcitizen': 0,
    'partner': 'no',
    'dependents': 'no',
    'tenure': 41,
    'phoneservice': 'yes',
    'multiplelines': 'no',
    'internetservice': 'dsl',
    'onlinesecurity': 'yes',
    'onlinebackup': 'no',
    'deviceprotection': 'yes',
    'techsupport': 'yes',
    'streamingtv': 'yes',
    'streamingmovies': 'yes',
    'contract': 'one_year',
    'paperlessbilling': 'yes',
    'paymentmethod': 'bank_transfer_(automatic)',
    'monthlycharges': 79.85,
    'totalcharges': 3320.75,
}
```

To predict whether this customer is going to churn, we can use the `predict` function we wrote in the previous chapter:

```
df = pd.DataFrame([customer])
y_pred = predict(df, dv, model)
y_pred[0]
```

This function needs a dataframe, so first we create a dataframe with one row—our customer. Next, we put it into the `predict` function. The result is a NumPy array with a single element—the predicted probability of churn for this customer:

```
0.059605
```

This means that this customer has a 6% probability of churning.

Now let's take a look at the `predict` function, which we wrote previously for applying the model to the customers in the validation set:

```
def predict(df, dv, model):
    cat = df[categorical + numerical].to_dict(orient='rows')
    X = dv.transform(cat)
    y_pred = model.predict_proba(X)[:, 1]
    return y_pred
```

Using it for one customer seems inefficient and unnecessary: we create a dataframe from a single customer only to convert this dataframe back to a dictionary later inside `predict`.

To avoid doing this unnecessary conversion, we can create a separate function for predicting the probability of churn for a single customer only. Let's call this function `predict_single`:

```
def predict_single(customer, dv, model):     ❶
    X = dv.transform([customer])              ❷
    y_pred = model.predict_proba(X)[:, 1]     ❸
    return y_pred[0]                          ❹
```

❶ Instead of passing a dataframe, passes a single customer

❷ Vectorizes the customer: creates the matrix X

❸ Applies the model to this matrix

❹ Because we only have one customer, we only need the first element of the result.

Using it becomes simpler—we simply invoke it with our customer (a dictionary):

```
predict_single(customer, dv, model)
```

The result is the same: this customer has a 6% probability of churning.

We trained our model inside the Jupyter Notebook we started in chapter 3. This model lives there, and once we stop the Jupyter Notebook, the trained model will disappear. This means that now we can use it only inside the notebook and nowhere else. Next, we see how to address it.

5.1.2 Using Pickle to save and load the model

To be able to use it outside of our notebook, we need to save it, and then later, another process can load and use it (figure 5.1).

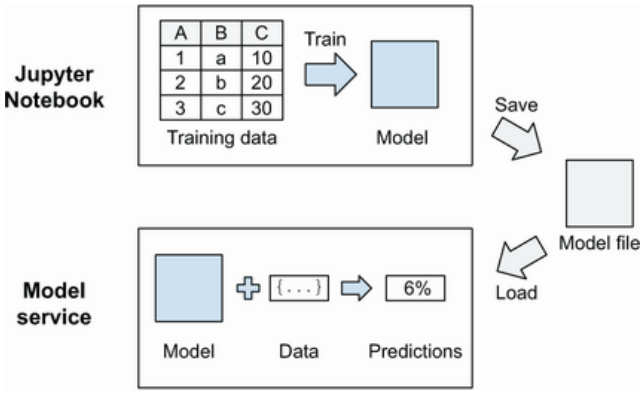

Figure 5.1 We train a model in a Jupyter Notebook. To use it, we first need to save it and then load it in a different process.

Pickle is a serialization/deserialization module that's already built into Python: using it, we can save an arbitrary Python object (with a few exceptions) to a file. Once we have a file, we can load the model from there in a different process.

NOTE "Pickle" can also be used as a verb: *pickling* an object in Python means saving it using the Pickle module.

SAVING THE MODEL

To save the model, we first import the Pickle module, and then use the `dump` function:

```
import pickle
```

```
with open('churn-model.bin', 'wb') as f_out:      ❶
    pickle.dump(model, f_out)                     ❷
```

❶ Specifies the file where we want to save

❷ Saves the model to file with Pickle

To save the model, we use the `open` function, which takes two arguments:

- The name of the file that we want to open. For us, it's churn-model.bin.
- The mode with which we open the file. For us, it's `wb`, which means we want to write to the file (`w`), and the file is binary (`b`) and not text —Pickle uses binary format for writing to files.

The `open` function returns `f_out` —the file descriptor we can use to write to the file.

Next, we use the `dump` function from Pickle. It also takes two arguments:

- The object we want to save. For us, it's `model`.
- The file descriptor, pointing to the output file, which is `f_out` for us.

Finally, we use the `with` construction in this code. When we open a file with `open`, we need to close it after we finish writing. When using `with`, it happens automatically. Without `with`, our code would look like this:

```
f_out = open('churn-model.bin', 'wb')
pickle.dump(model, f_out)
f_out.close()
```

In our case, however, saving just the model is not enough: we also have a `DictVectorizer` that we "trained" together with the model. We need to save both.

The simplest way of doing this is to put both of them in a tuple when pickling:

```
with open('churn-model.bin', 'wb') as f_out:
    pickle.dump((dv, model), f_out)             ❶
```

❶ The object we save is a tuple with two elements.

LOADING THE MODEL

To load the model, we use the `load` function from Pickle. We can test it in the same Jupyter Notebook:

```
with open('churn-model.bin', 'rb') as f_in:     ❶
    dv, model = pickle.load(f_in)               ❷
```

❶ Opens the file in read mode

❷ Loads the tuple and unpacks it

We again use the `open` function, but this time, with a different mode: `rb`, which means we open it for reading (`r`), and the file is binary (`b`).

WARNING Be careful when specifying the mode. Accidentally specifying an incorrect mode may result in data loss: if you open an existing file with the `w` mode instead of `r`, it will overwrite the content.

Because we saved a tuple, we unpack it when loading, so we get both the vectorizer and the model at the same time.

WARNING Unpickling objects found on the internet is not secure: it can execute arbitrary code on your machine. Use it only for things you trust and things you saved yourself.

Let's create a simple Python script that loads the model and applies it to a customer.

We will call this file churn_serving.py. (In the book's GitHub repository, this file is called churn_serving_simple.py.) It contains

- The `predict_single` function that we wrote earlier
- The code for loading the model

- The code for applying the model to a customer

You can refer to appendix B to learn more about creating Python scripts.

First, we start with imports. For this script, we need to import Pickle and NumPy:

```
import pickle
import numpy as np
```

Next, let's put the `predict_single` function there:

```
def predict_single(customer, dv, model):
    X = dv.transform([customer])
    y_pred = model.predict_proba(X)[:, 1]
    return y_pred[0]
```

Now we can load our model:

```
with open('churn-model.bin', 'rb') as f_in:
    dv, model = pickle.load(f_in)
```

And apply it:

```
customer = {
    'customerid': '8879-zkjof',
    'gender': 'female',
    'seniorcitizen': 0,
    'partner': 'no',
    'dependents': 'no',
    'tenure': 41,
    'phoneservice': 'yes',
    'multiplelines': 'no',
    'internetservice': 'dsl',
    'onlinesecurity': 'yes',
    'onlinebackup': 'no',
    'deviceprotection': 'yes',
    'techsupport': 'yes',
    'streamingtv': 'yes',
    'streamingmovies': 'yes',
```

```
    'contract': 'one_year',
    'paperlessbilling': 'yes',
    'paymentmethod': 'bank_transfer_(automatic)',
    'monthlycharges': 79.85,
    'totalcharges': 3320.75,
}

prediction = predict_single(customer, dv, model)
```

Finally, let's display the results:

```
print('prediction: %.3f' % prediction)

if prediction >= 0.5:
    print('verdict: Churn')
else:
    print('verdict: Not churn')
```

After saving the file, we can run this script with Python:

```
python churn_serving.py
```

We should immediately see the results:

```
prediction: 0.059
verdict: Not churn
```

This way, we can load the model and apply it to the customer we specified in the script.

Of course, we aren't going to manually put the information about customers in the script. In the next section, we cover a more practical approach: putting the model into a web service.

5.2 Model serving

We already know how to load a trained model in a different process. Now we need to *serve* this model—make it available for others to use.

In practice, this usually means that a model is deployed as a web service, so other services can communicate with it, ask for predictions, and use the results to make their own decisions.

In this section, we see how to do it in Python with Flask—a Python framework for creating web services. First, we take a look at why we need to use a web service for it.

5.2.1 Web services

We already know how to use a model to make a prediction, but so far, we have simply hardcoded the features of a customer as a Python dictionary. Let's try to imagine how our model will be used in practice.

Suppose we have a service for running marketing campaigns. For each customer, it needs to determine the probability of churn, and if it's high enough, it will send a promotional email with discounts. Of course, this service needs to use our model to decide whether it should send an email.

One possible way of achieving this is to modify the code of the campaign service: load the model, and score the customers right in the service. This approach is good, but the campaign service needs to be in Python, and we need to have full control over its code.

Unfortunately, this situation is not always the case: it may be written in some other language, or a different team might be in charge of this project, which means we won't have the control we need.

The typical solution for this problem is putting a model inside a web service—a small service (a *microservice*) that only takes care of scoring customers.

So, we need to create a churn service—a service in Python that will serve the churn model. Given the features of a customer, it will respond with the probability of churn for this customer. For each customer, the campaign service will ask the churn service for the probability of churn, and if it's high enough, then we send a promotional email (figure 5.2).

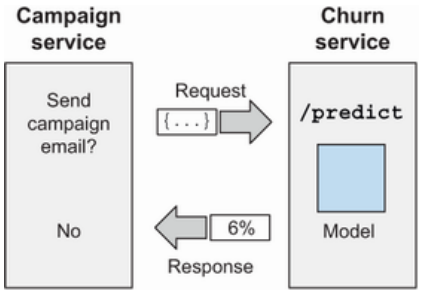

Figure 5.2 The churn service takes care of serving the churn-prediction model, making it possible for other services to use it.

This gives us another advantage: separation of concerns. If the model is created by data scientists, then they can take ownership of the service and maintain it, while the other team takes care of the campaign service.

One of the most popular frameworks for creating web services in Python is Flask, which we cover next.

5.2.2 Flask

The easiest way to implement a web service in Python is to use Flask. It's quite lightweight, requires little code to get started, and hides most of the complexity of dealing with HTTP requests and responses.

Before we put our model inside a web service, let's cover the basics of using Flask. For that, we'll create a simple function and make it available as a web service. After covering the basics, we'll take care of the model.

Suppose we have a simple Python function called `ping`:

```
def ping():
    return 'PONG'
```

It doesn't do much: when invoked, it simply responds with PONG. Let's use Flask to turn this function into a web service.

Anaconda comes with Flask preinstalled, but if you use a different Python distribution, you'll need to install it:

```
pip install flask
```

We will put this code in a Python file and will call it flask_test.py.

To be able to use Flask, we first need to import it:

```
from flask import Flask
```

Now we create a Flask app—the central object for registering functions that need to be exposed in the web service. We'll call our app test:

```
app = Flask('test')
```

Next, we need to specify how to reach the function by assigning it to an address, or a *route* in Flask terms. In our case, we want to use the `/ping` address:

```
@app.route('/ping', methods=['GET'])        ❶
def ping():
    return 'PONG'
```

❶ Registers the /ping route, and assigns it to the ping function

This code uses decorators—an advanced Python feature that we don't cover in this book. We don't need to understand how it works in detail; it's enough to know that by putting `@app.route` on top of the function definition, we assign the `/ping` address of the web service to the `ping` function.

To run it, we only need one last bit:

```
if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0', port=9696)
```

The `run` method of `app` starts the service. We specify three parameters:

- `debug=True`. Restarts our application automatically when there are changes in the code.
- `host='0.0.0.0'`. Makes the web service public; otherwise, it won't be possible to reach it when it's hosted on a remote machine (e.g., in AWS).
- `port=9696`. The port that we use to access the application.

We're ready to start our service now. Let's do it:

```
python flask_test.py
```

When we run it, we should see the following:

```
 * Serving Flask app "test" (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://0.0.0.0:9696/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 162-129-136
```

This means that our Flask app is now running and ready to get requests. To test it, we can use our browser: open it and type localhost:9696/ping in the address bar. If you run it on a remote server, you should replace localhost with the address of the server. (For AWS EC2, use the public DNS hostname. Make sure that the port 9696 is open in the security group of your EC2 instance: go to the security group, and add a custom TCP rule with the port 9696 and the source 0.0.0.0/0.) The browser should respond with PONG (figure 5.3).

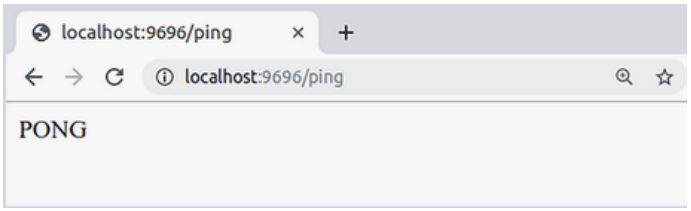

Figure 5.3 The easiest way to check if our application works is to use a web browser.

Flask logs all the requests it receives, so we should see a line indicating that there was a GET request on the `/ping` route:

```
127.0.0.1 - - [02/Apr/2020 21:59:09] "GET /ping HTTP/1.1" 200 -
```

As we can see, Flask is quite simple: with fewer than 10 lines of code, we created a web service.

Next, we'll see how to adjust our script for churn prediction and also turn it into a web service.

5.2.3 Serving churn model with Flask

We've learned a bit of Flask, so now we can come back to our script and convert it to a Flask application.

To score a customer, our model needs to get the features, which means that we need a way of transferring some data from one service (the campaign service) to another (the churn service).

As a data exchange format, web services typically use JSON (Javascript Object Notation). It's similar to the way we define dictionaries in Python:

```
{
    "customerid": "8879-zkjof",
    "gender": "female",
    "seniorcitizen": 0,
    "partner": "no",
    "dependents": "no",
```

```
    ...
}
```

To send data, we use POST requests, not GET: POST requests can include the data in the request, whereas GET cannot.

Thus, to make it possible for the campaign service to get predictions from the churn service, we need to create a `/predict` route that accepts POST requests. The churn service will parse JSON data about a customer and respond in JSON as well (figure 5.4).

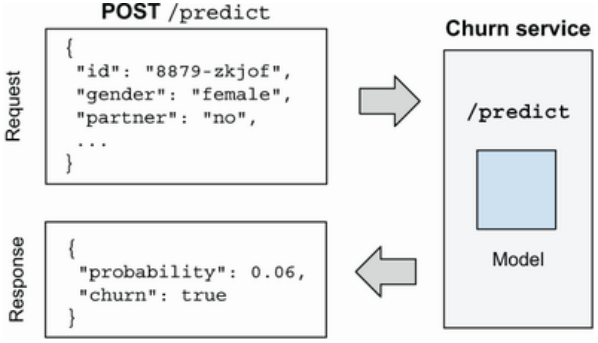

Figure 5.4 To get predictions, we POST the data about a customer in JSON to the `/predict` route and get the probability of churn in response.

Now we know what we want to do, so let's start modifying the churn_serving.py file.

First, we add a few more imports at the top of the file:

```
from flask import Flask, request, jsonify
```

Although previously we imported only `Flask`, now we need to import two more things:

- `request`: To get the content of a POST request
- `jsonsify`: To respond with JSON

Next, create the Flask app. Let's call it churn:

```
app = Flask('churn')
```

Now we need to create a function that

- Gets the customer data in a request
- Invokes `predict_simple` to score the customer
- Responds with the probability of churn in JSON

We'll call this function `predict` and assign it to the `/predict` route:

```
@app.route('/predict', methods=['POST'])                ❶
def predict():
    customer = request.get_json()                       ❷

    prediction = predict_single(customer, dv, model)    ❸
    churn = prediction >= 0.5                           ❹

    result = {                                          ❹
        'churn_probability': float(prediction),         ❹
        'churn': bool(churn),                           ❹
    }                                                   ❹

    return jsonify(result)                              ❺
```

❶ Assigns the /predict route to the predict function

❷ Gets the content of the request in JSON

❸ Scores the customer

❹ Prepares the response

❺ Converts the response to JSON

To assign the route to the function, we use the `@app.route` decorator, where we also tell Flask to expect POST requests only.

The core content of the `predict` function is similar to what we did in the script previously: it takes a customer, passes it to `predict_single`, and does some work with the result.

Finally, let's add the last two lines for running the Flask app:

```
if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0', port=9696)
```

We're ready to run it:

```
python churn_serving.py
```

After running it, we should see a message saying that the app started and is now waiting for incoming requests:

```
 * Serving Flask app "churn" (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://0.0.0.0:9696/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
```

Testing this code is a bit more difficult than previously: this time, we need to use POST requests and include the customer we want to score in the body of the request.

The simplest way of doing this is to use the requests library in Python. It also comes preinstalled in Anaconda, but if you use a different distribution, you can install it with `pip`:

```
pip install requests
```

We can open the same Jupyter Notebook that we used previously and test the web service from there.

First, import requests:

```
import requests
```

Now, make a POST request to our service

```
url = 'http://localhost:9696/predict'              ❶
response = requests.post(url, json=customer)       ❷
result = response.json()                           ❸
```

❶ The URL where the service lives

❷ Sends the customer (as JSON) in the POST request

❸ Parses the response as JSON

The `results` variable contains the response from the churn service:

```
{'churn': False, 'churn_probability': 0.05960590758316391}
```

This is the same information we previously saw in the terminal, but now we got it as a response from a web service.

NOTE Some tools, like Postman (**https://www.postman.com/**), make it easier to test web services. We don't cover Postman in this book, but you're free to give it a try.

If the campaign service used Python, this is exactly how it could communicate with the churn service and decide who should get promotional emails.

With just a few lines of code, we created a working web service that runs on our laptop. In the next section, we'll see how to manage dependencies in our service and prepare it for deployment.

5.3 Managing dependencies

For local development, Anaconda is a perfect tool: it has almost all the libraries we may ever need. This, however, also has a downside: it takes up 4 GB when unpacked, which is too large. For production, we prefer to have only the libraries we actually need.

Additionally, different services have different requirements. Often, these requirements conflict, so we cannot use the same environment for running multiple services at the same time.

In this section, we see how to manage dependencies of our application in an isolated way that doesn't interfere with other services. We cover two tools for this: Pipenv, for managing Python libraries, and Docker, for managing the system dependencies such as the operating system and the system libraries.

5.3.1 Pipenv

To serve the churn model, we only need a few libraries: NumPy, Scikit-learn, and Flask. So, instead of bringing in the entire Anaconda distribution with all its libraries, we can get a fresh Python installation and install only the libraries we need with `pip`:

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

Before we do that, let's think for a moment about what happens when we use `pip` to install a library:

- We run `pip install library` to install a Python package called Library (let's suppose it exists).
- Pip goes to PyPI.org (the Python package index—a repository with Python packages), and gets and installs the latest version of this library. Let's say, it's version 1.0.0.

After installing it, we develop and test our service using this particular version. Everything works great. Later, our colleagues want to help us with the project, so they also run `pip install` to set up everything on their machine—except this time, the latest version turns out to be 1.3.1.

If we're unlucky, versions 1.0.0 and 1.3.1 might not be compatible with each other, meaning that the code we wrote for version 1.0.0 won't work for version 1.3.1.

It's possible to solve this problem by specifying the exact version of the library when installing it with `pip`:

```
pip install library==1.0.0
```

Unfortunately, a different problem may appear: what if some of our colleagues already have version 1.3.1 installed, and they already used it for some other services? In this case, they cannot go back to using version 1.0.0: it could cause their code to stop working.

We can solve these problems by creating a *virtual environment* for each project—a separate Python distribution with nothing else but libraries required for this particular project.

Pipenv is a tool that makes managing virtual environments easier. We can install it with pip:

```
pip install pipenv
```

After that, we use `pipenv` instead of `pip` for installing dependencies:

```
pipenv install numpy scikit-learn flask
```

When running it, we'll see that first, it configures the virtual environment, and then it installs the libraries:

```
Running virtualenv with interpreter .../bin/python3
✔ Successfully created virtual environment!
Virtualenv location: ...
Creating a Pipfile for this project...
Installing numpy...
Adding numpy to Pipfile's [packages]...
✔ Installation Succeeded
Installing scikit-learn...
Adding scikit-learn to Pipfile's [packages]...
✔ Installation Succeeded
Installing flask...
Adding flask to Pipfile's [packages]...
✔ Installation Succeeded
Pipfile.lock not found, creating...
Locking [dev-packages] dependencies...
```

```
Locking [packages] dependencies...
⣾ Locking...
```

After finishing the installation, it creates two files: Pipenv and Pipenv.lock.

The Pipenv file looks pretty simple:

```
[[source]]
name = "pypi"
url = "https://pypi.org/simple"
verify_ssl = true

[dev-packages]

[packages]
numpy = "*"
scikit-learn = "*"
flask = "*"

[requires]
python_version = "3.7"
```

We see that this file contains a list of libraries as well as the version of Python we use.

The other file—Pipenv.lock—contains the specific versions of the libraries that we used for the project. The file is too large to show in its entirety here, but let's take a look at one of the entries in the file:

```
"flask": {
    "hashes": [
        "sha256:4efa1ae2d7c9865af48986de8aeb8504...",
        "sha256:8a4fdd8936eba2512e9c85df320a37e6..."
    ],
    "index": "pypi",
    "version": "==1.1.2"
}
```

As we can see, it records the exact version of the library that was used during installation. To make sure the library doesn't change, it also saves the hashes—the checksums that can be used to validate that in the future we download the exact same version of the library. This way, we "lock" the dependencies to specific versions. By doing this, we make sure that in the future we will not have surprises with two incompatible versions of the same library.

If somebody needs to work on our project, they simply need to run the `install` command:

```
pipenv install
```

This step will first create a virtual environment and then install all the required libraries from Pipenv.lock.

IMPORTANT Locking the version of a library is important for reproducibility in the future and helps us avoid having unpleasant surprises with code incompatibility.

After all the libraries are installed, we need to activate the virtual environment—this way, our application will use the correct versions of the libraries. We do it by running the `shell` command:

```
pipenv shell
```

It tells us that it's running in a virtual environment:

```
Launching subshell in virtual environment...
```

Now we can run our script for serving:

```
python churn_serving.py
```

Alternatively, instead of first explicitly entering the virtual environment and then running the script, we can perform these two steps with just one command:

```
pipenv run python churn_serving.py
```

The `run` command in Pipenv simply runs the specified program in the virtual environment.

Regardless of the way we run it, we should see exactly the same output as previously:

```
 * Serving Flask app "churn" (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://0.0.0.0:9696/ (Press CTRL+C to quit)
```

When we test it with requests, we see the same output:

```
{'churn': False, 'churn_probability': 0.05960590758316391}
```

You most likely also noticed the following warning in the console:

```
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
```

The built-in Flask web server is indeed for development only: it's very easy to use for testing our application, but it won't work reliably under load. We should use a proper WSGI server instead, as the warning suggests.

WSGI stands for *web server gateway interface*, which is a specification describing how Python applications should handle HTTP requests. The details of WSGI are not important for the purposes of this book, so we won't cover it in detail.

We will, however, address the warning by installing a production WSGI server. We have multiple possible options in Python, and we'll use Gunicorn.

NOTE Gunicorn doesn't work on Windows: it relies on features specific to Linux and Unix (which includes MacOS). A good alternative that also works on Windows is Waitress. Later, we will use Docker, which will solve this problem—it runs Linux inside a container.

Let's install it with Pipenv:

```
pipenv install gunicorn
```

This command installs the library and includes it as a dependency in the project by adding it to the Pipenv and Pipenv.lock files.

Let's run our application with Gunicorn:

```
pipenv run gunicorn --bind 0.0.0.0:9696 churn_serving:app
```

If everything goes well, we should see the following messages in the terminal:

```
[2020-04-13 22:58:44 +0200] [15705] [INFO] Starting gunicorn 20.0.4
[2020-04-13 22:58:44 +0200] [15705] [INFO] Listening at: http://0.0.0.0:9696 (15705)
[2020-04-13 22:58:44 +0200] [15705] [INFO] Using worker: sync
[2020-04-13 22:58:44 +0200] [16541] [INFO] Booting worker with pid: 16541
```

Unlike the Flask built-in web server, Gunicorn is ready for production, so it will not have any problems under load when we start using it.

If we test it with the same code as previously, we see the same answer:

```
{'churn': False, 'churn_probability': 0.05960590758316391}
```

Pipenv is a great tool for managing dependencies: it isolates the required libraries into a separate environment, thus helping us avoid conflicts between different versions of the same package.

In the next section, we look at Docker, which allows us to isolate our application even further and ensure it runs smoothly anywhere.

5.3.2 Docker

We have learned how to manage Python dependencies with Pipenv. However, some of the dependencies live outside of Python. Most importantly, these dependencies include the operating system (OS) as well as the system libraries.

For example, we might use Ubuntu version 16.04 for developing our service, but if some of our colleagues use Ubuntu version 20.04, they may run into trouble when trying to execute the service on their laptop.

Docker solves this "but it works on my machine" problem by also packaging the OS and the system libraries into a *Docker container*—a self-contained environment that works anywhere where Docker is installed (figure 5.5).

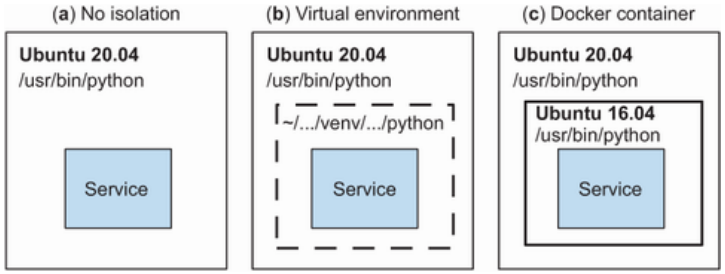

Figure 5.5 In case of no isolation (a), the service runs with system Python. In virtual environments (b), we isolate the dependencies of our service inside the environment. In Docker containers (c), we isolate the entire environment of the service, including the OS and system libraries.

Once the service is packaged into a Docker container, we can run it on the *host machine*—our laptop (regardless of the OS) or any public cloud provider.

Let's see how to use it for our project. We assume you already have Docker installed. Please refer to appendix A for details on how to install it.

First, we need to create a *Docker image*—the description of our service that includes all the settings and dependencies. Docker will later use the

image to create a container. To do it, we need a Dockerfile—a file with instructions on how the image should be created (figure 5.6).

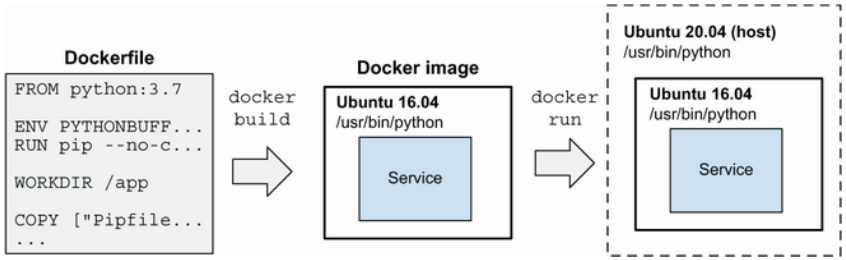

Figure 5.6 We build an image using instructions from Dockerfile. Then we can run this image on a host machine.

Let's create a file with name Dockerfile and the following content (note that the file shouldn't include the annotations):

```
FROM python:3.7.5-slim                                          ❶

ENV PYTHONUNBUFFERED=TRUE                                       ❷

RUN pip --no-cache-dir install pipenv                           ❸

WORKDIR /app                                                    ❹

COPY ["Pipfile", "Pipfile.lock", "./"]                          ❺

RUN pipenv install --deploy --system && \                       ❻
    rm -rf /root/.cache                                         ❻

COPY ["*.py", "churn-model.bin", "./"]                          ❼

EXPOSE 9696                                                     ❽

ENTRYPOINT ["gunicorn", "--bind", "0.0.0.0:9696", "churn_serving:app"] ❾
```

❶ Specifies the base image

❷ Sets a special Python settings for being able to see logs

❸ Installs Pipenv

❹ Sets the working directory to /app

❺ Copies the Pipenv files

❻ Installs the dependencies from the Pipenv files

❼ Copies our code as well as the model

❽ Opens the port that our web service uses

❾ Specifies how the service should be started

That's a lot of information to unpack, especially if you have never seen Dockerfiles previously. Let's go line by line.

First, we specify the base Docker image:

```
FROM python:3.7.5-slim
```

We use this image as the starting point and build our own image on top of that. Typically, the base image already contains the OS and the system libraries like Python itself, so we need to install only the dependencies of our project. In our case, we use `python:3.7.5-slim`, which is based on Debian 10.2 and contains Python version 3.7.5 and `pip`. You can read more about the Python base image in Docker hub (**https://hub.docker.com/_/python**)—the service for sharing Docker images.

All Dockerfiles should start with the `FROM` statement.

Next, we set the `PYTHONUNBUFFERED` environmental variable to `TRUE`:

```
ENV PYTHONUNBUFFERED=TRUE
```

Without this setting, we won't be able to see the logs when running Python scripts inside Docker.

Then, we use `pip` to install Pipenv:

```
RUN pip --no-cache-dir install pipenv
```

The `RUN` instruction in Docker simply runs a shell command. By default, `pip` saves the libraries to a cache, so later they can be installed faster. We don't need that in a Docker container, so we use the `--no-cache-dir` setting.

Then, we specify the working directory:

```
WORKDIR /app
```

This is roughly equivalent to the `cd` command in Linux (change directory), so everything we will run after that will be executed in the /app folder.

Then, we copy the Pipenv files to the current working directory (i.e., /app):

```
COPY ["Pipfile", "Pipfile.lock", "./"]
```

We use these files for installing the dependencies with Pipenv:

```
RUN pipenv install --deploy --system && \
    rm -rf /root/.cache
```

Previously, we simply used `pipenv install` for doing this. Here, we include two extra parameters: `--deploy` and `--system`. Inside Docker, we don't need to create a virtual environment—our Docker container is already isolated from the rest of the system. Setting these parameters allows us to skip creating a virtual environment and use the system Python for installing all the dependencies.

After installing the libraries, we clean the cache to make sure our Docker image doesn't grow too big.

Then, we copy our project files as well as the pickled model:

```
COPY ["*.py", "churn-model.bin", "./"]
```

Next, we specify which port our application will use. In our case, it's 9696:

```
EXPOSE 9696
```

Finally, we tell Docker how our application should be started:

```
ENTRYPOINT ["gunicorn", "--bind", "0.0.0.0:9696", "churn_serving:app"]
```

This is the same command we used previously when running Gunicorn locally.

Let's build the image. We do it by running the `build` command in Docker:

```
docker build -t churn-prediction .
```

The `-t` flag lets us set the tag name for the image, and the final parameter—the dot—specifies the directory with the Dockerfile. In our case, it means that we use the current directory.

When we run it, the first thing Docker does is download the base image:

```
Sending build context to Docker daemon  51.71kB
Step 1/11 : FROM python:3.7.5-slim
3.7.5-slim: Pulling from library/python
000eee12ec04: Downloading  24.84MB/27.09MB
ddc2d83f8229: Download complete
735b0bee82a3: Downloading  19.56MB/28.02MB
8c69dcedfc84: Download complete
495e1cccc7f9: Download complete
```

Then it executes each line of the Dockerfile one by one:

```
Step 2/9 : ENV PYTHONUNBUFFERED=TRUE
 ---> Running in d263b412618b
```

```
Removing intermediate container d263b412618b
 ---> 7987e3cf611f
Step 3/9 : RUN pip --no-cache-dir install pipenv
 ---> Running in e8e9d329ed07
Collecting pipenv
 ...
```

At the end, Docker tells us that it successfully built an image and tagged it as churn-prediction:latest:

```
Successfully built d9c50e4619a1
Successfully tagged churn-prediction:latest
```

We're ready to use this image to start a Docker container. Use the `run` command for that:

```
docker run -it -p 9696:9696 churn-prediction:latest
```

We specify a few parameters here:

- The `-it` flag tells Docker that we run it from our terminal and we need to see the results.
- The `-p` parameter specifies the port mapping. `9696:9696` means to map the port 9696 on the container to the port 9696 on the host machine.
- Finally, we need the image name and tag, which in our case is `churn-prediction :latest`.

Now our service is running inside a Docker container, and we can connect to it using port 9696 (figure 5.7). This is the same port we used for our application previously.

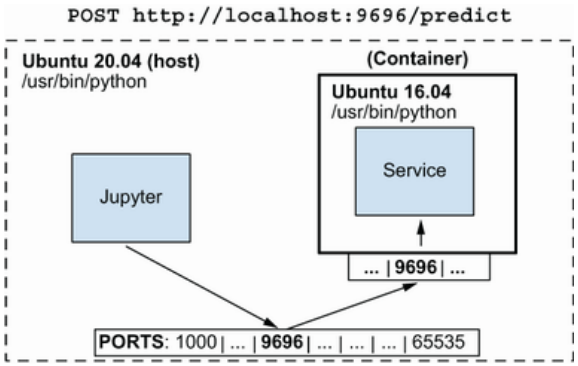

Figure 5.7 The 9696 port on the host machine is mapped to the 9696 port of the container, so when we send a request to `localhost:9696`, it's handled by our service in Docker.

Let's test it using the same code. When we run it, we'll see the same response:

```
{'churn': False, 'churn_probability': 0.05960590758316391}
```

Docker makes it easy to run services in a reproducible way. With Docker, the environment inside the container always stays the same. This means that if we can run our service on a laptop, it will work anywhere else.

We already tested our application on our laptop, so now let's see how to run it on a public cloud and deploy it to AWS.

5.4 Deployment

We don't run production services on our laptops; we need special servers for that.

In this section, we'll cover one possible option for that: Amazon Web Services, or AWS. We decided to choose AWS for its popularity—we're not affiliated with Amazon or AWS.

Other popular public clouds include Google Cloud, Microsoft Azure, and Digital Ocean. We don't cover them in this book, but you should be able to find similar instructions online and deploy a model to your favourite cloud provider.

This section is optional, so you can safely skip it. To follow the instructions in this section, you need to have an AWS account and configure the AWS command-line tool (CLI). Please refer to appendix A to see how to set it up.

5.4.1 AWS Elastic Beanstalk

AWS provides a lot of services, and we have many possible ways of deploying a web service there. For example, you can rent an EC2 machine (a server in AWS) and manually set up a service on it, use a "serverless" approach with AWS Lambda, or use a range of other services.

In this section, we'll use AWS Elastic Beanstalk, which is one of the simplest ways of deploying a model to AWS. Additionally, our service is simple enough, so it's possible to stay within the free-tier limits. In other words, we can use it for free for the first year.

Elastic Beanstalk automatically takes care of many things that we typically need in production, including

- Deploying our service to EC2 instances
- Scaling up: adding more instances to handle the load during peak hours
- Scaling down: removing these instances when the load goes away
- Restarting the service if it crashes for any reason
- Balancing the load between instances

We'll also need a special utility—Elastic Beanstalk command-line interface (CLI)—to use Elastic Beanstalk. The CLI is written in Python, so we can install it with `pip`, like any other Python tool.

However, because we use Pipenv, we can add it as a development dependency. This way, we'll install it only for our project and not systemwide.

```
pipenv install awsebcli --dev
```

NOTE Development dependencies are the tools and libraries that we use for developing our application. Usually, we need them only locally and don't need them in the actual package deployed to production.

After installing Elastic Beanstalk, we can enter the virtual environment of our project:

```
pipenv shell
```

Now the CLI should be available. Let's check it:

```
eb --version
```

It should print the version:

```
EB CLI 3.18.0 (Python 3.7.7)
```

Next, we run the initialization command:

```
eb init -p docker churn-serving
```

Note that we use `-p docker` : this way, we specify that this is a Docker-based project.

If everything is fine, it creates a couple of files, including a config.yml file in .elasticbeanstalk folder.

Now we can test our application locally by using `local run` command:

```
eb local run --port 9696
```

This should work in the same way as in the previous section with Docker: it'll first build an image and then run the container.

To test it, we can use the same code as previously and get the same answer:

```
{'churn': False, 'churn_probability': 0.05960590758316391}
```

After verifying that it works well locally, we're ready to deploy it to AWS. We can do that with one command:

```
eb create churn-serving-env
```

This simple command takes care of setting up everything we need, from the EC2 instances to auto-scaling rules:

```
Creating application version archive "app-200418_120347".
Uploading churn-serving/app-200418_120347.zip to S3. This may take a while.
Upload Complete.
Environment details for: churn-serving-env
  Application name: churn-serving
  Region: us-west-2
  Deployed Version: app-200418_120347
  Environment ID: e-3xkqdzdjbq
  Platform: arn:aws:elasticbeanstalk:us-west-2::platform/Docker running on 64bit Amazon Linux 2/3.0
  Tier: WebServer-Standard-1.0
  CNAME: UNKNOWN
  Updated: 2020-04-18 10:03:52.276000+00:00
Printing Status:
2020-04-18 10:03:51    INFO    createEnvironment is starting.
 -- Events -- (safe to Ctrl+C)
```

It'll take a few minutes to create everything. We can monitor the process and see what it's doing in the terminal.

When it's ready, we should see the following information:

```
2020-04-18 10:06:53    INFO    Application available at churn-serving-env.5w9pp7bkmj.us-west-2.elas
2020-04-18 10:06:53    INFO    Successfully launched environment: churn-serving-env
```

The URL (churn-serving-env.5w9pp7bkmj.us-west-2.elasticbeanstalk.com) in the logs is important: this is how we reach our application. Now we can use this URL to make predictions (figure 5.8).

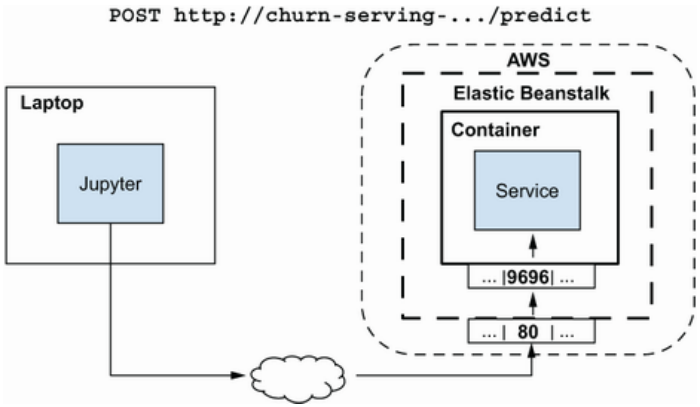

Figure 5.8 Our service is deployed inside a container on AWS Elastic Beanstalk. To reach it, we use its public URL.

Let's test it:

```
host = 'churn-serving-env.5w9pp7bkmj.us-west-2.elasticbeanstalk.com'
url = 'http://%s/predict' % host
response = requests.post(url, json=customer)
result = response.json()
result
```

As previously, we should see the same response:

```
{'churn': False, 'churn_probability': 0.05960590758316393}
```

That's all! We have a running service.

WARNING This is a toy example, and the service we created is accessible by anyone in the world. If you do it inside an organization, the access should be restricted as much as possible. It's not difficult to extend this example to be secure, but it's outside the scope of this book. Consult the security department at your company before doing it at work.

We can do everything from the terminal using the CLI, but it's also possible to manage it from the AWS Console. To do so, we find Elastic Beanstalk there and select the environment we just created (figure 5.9).

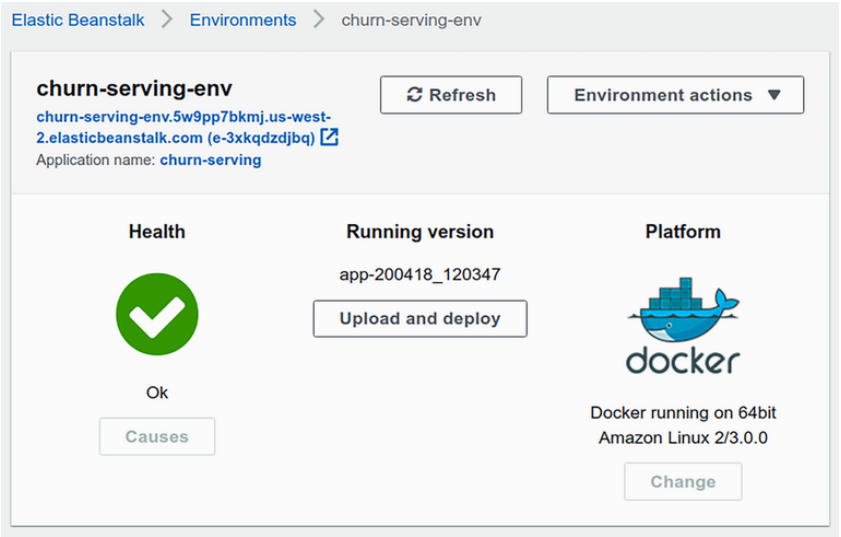

Figure 5.9 We can manage the Elastic Beanstalk environment in the AWS Console.

To turn it off, choose Terminate deployment in the Environment actions menu using the AWS Console.

WARNING Even though Elastic Beanstalk is free-tier eligible, we should always be careful and turn it off as soon as we no longer need it.

Alternatively, we use the CLI to do it:

```
eb terminate churn-serving-env
```

After a few minutes, the deployment will be removed from AWS and the URL will no longer be accessible.

AWS Elastic Beanstalk is a great tool for getting started with serving machine learning models. More advanced ways of doing it involve container orchestration systems like AWS ECS or Kubernetes or “serverless” with AWS Lambda. We will come back to this topic in chapters 8 and 9 when covering the deployment of deep learning models.

5.5 Next steps

We've learned about Pipenv and Docker and deployed our model to AWS Elastic Beanstalk. Try these other things to expand your skills on your own.

5.5.1 Exercises

Try the following exercises to further explore the topics of model deployment:

- If you don't use AWS, try to repeat the steps from section 5.4 on any other cloud provider. For example, you could try Google Cloud, Microsoft Azure, Heroku, or Digital Ocean.
- Flask is not the only way of creating web services in Python. You can try alternative frameworks like FastAPI (**https://fastapi.tiangolo.com/**), Bottle (**https://github.com/bottlepy/bottle**), or Falcon (**https://github.com/falconry/falcon**).

5.5.2 Other projects

You can continue other projects from the previous chapters and make them available as a web service as well. For example:

- The car-price prediction model we created in chapter 2
- The self-study projects from chapter 3: the lead scoring project and the default prediction project.

Summary

- Pickle is a serialization/deserialization library that comes built into Python. We can use it to save a model we trained in Jupyter Notebook and load it in a Python script.
- The simplest way of making a model available for others is wrapping it into a Flask web service.
- Pipenv is a tool for managing Python dependencies by creating virtual environments, so dependencies of one Python project don't interfere with dependencies of another Python project.

- Docker makes it possible to isolate the service completely from other services by packaging into a Docker container not only the Python dependencies but also the system dependencies, as well as the operational system itself.
- AWS Elastic Beanstalk is a simple way to deploy a web service. It takes care of managing EC2 instances, scaling the service up and down, and restarting if something fails.

In the next chapter, we continue learning about classification but with a different type of model—decision trees.