Unidad 8. Despliegue y convergencia tecnológica

“Despliegue de modelos mediante aplicaciones web: el framework Flask”

Contenido

En este documento veremos cómo podemos desplegar nuestros modelos utilizando aplicaciones web. De esta forma, configuraremos un servidor web para desarrollar una API que responda a peticiones de clientes que harán consultas sobre el modelo, y responderemos enviando la información en formato JSON, como hacen muchas de las webs de modelos pre-entrenados que hemos visto en otros documentos.

1. Frameworks disponibles

A la hora de desplegar nuestro modelo de IA sobre un servidor, existen distintas alternativas en Python que podemos emplear, como, por ejemplo:

- **Flask**: es un framework ligero, flexible y fácil de usar, con una integración sencilla con modelos de IA.
- **Django**: es otro framework más completo (y complejo), que permite desarrollar distintos tipos de aplicaciones web, aunque su curva de aprendizaje no es tan sencilla.
- **FastAPI**: otro framework ligero que permite construir APIs en Python

En esta unidad nos centraremos en configurar y utilizar **Flask**. Recuerda consultar su web oficial para ampliar referencias.

Para poder trabajar con Flask es necesario conocer el concepto de WSGI (Web Server Gateway Interface), que es una especificación de una interface simple y universal entre los servidores web y las aplicaciones web o frameworks desarrolladas con python.

2. Flask, características y el modelo MVC

Flask es un “micro” framework escrito en Python y concebido para facilitar el desarrollo de aplicaciones Web bajo el patrón MVC. A diferencia de otros más complejos como Django, Flask proporciona una estructura minimalista y flexible que permite construir aplicaciones web de manera rápida y controlada. Su filosofía se basa en el principio de "keep it simple", lo que lo convierte en una excelente opción para crear APIs RESTful, prototipos rápidos y microservicios.

Características clave de Flask

- **Ligereza y simplicidad**: No impone una estructura rígida. El desarrollador elige cómo organizar el proyecto.
- **Basado en Werkzeug y Jinja2**

- **Extensible**: aunque de base es mínimo, permite integrar fácilmente extensiones para autenticación, ORM, validación de formularios, migraciones, etc.
- **Ideal para APIs**: Muy usado en proyectos de machine learning e inteligencia artificial para servir modelos como APIs RESTful.

¿Por qué usar Flask en un entorno de IA?

- **Integración directa con Python**, el lenguaje más utilizado en inteligencia artificial.
- Permite *servir modelos entrenados* como servicios web consumibles por aplicaciones cliente.
- Facilita la creación de **interfaces ligeras de prueba o demo** para modelos y pipelines de datos.
- Compatible con librerías como TensorFlow, PyTorch, Scikit-learn o Pandas, sin capas adicionales.

Flask y el MVC

Flask es totalmente compatible con el patrón MVC. Aunque por defecto no tiene un ORM, podemos usar una extensión de Flask para definir el modelo de datos. Esta característica nos abstrae del uso del motor de Base de Datos y lo hace independiente.

Con Flask vamos a definir un controlador, que es capaz de determinar las rutas con las que accedemos a la aplicación, procesar la información necesaria y mostrar la información necesaria en cada momento.

Flask utiliza jinja2 como motor de plantillas, con lo que es muy fácil diseñar las vistas que vamos a mostrar a los usuarios en cada momento.

Extensiones útiles para Flask

El “minimalismo” de Flask requiere que se usen algunas extensiones. Veamos las más utilizadas:

- **Flask-Bootstrap** te ahorrará muchísimo tiempo si quieres una interfaz con buen diseño y responsiva rápidamente. Bootstrap es un framework front-end muy popular y esta extensión facilita su integración en tus plantillas.
- **Flask-WTF** es fundamental para trabajar con formularios de manera segura y eficiente. La validación y el manejo de formularios HTML pueden ser tediosos, y Flask-WTF lo simplifica enormemente.

- **Flask-SQLAlchemy** es, como mencionaste, un ORM potente que facilita la interacción con bases de datos relacionales. Es una de las extensiones más utilizadas en proyectos Flask de tamaño medio a grande.
- **Flask-Login** es imprescindible para cualquier aplicación que necesite autenticación de usuarios y gestión de sesiones. Proporciona las herramientas necesarias para implementar el inicio y cierre de sesión, recordar usuarios y proteger rutas.

3. Primeros pasos con Flask

Para poder utilizar Flask debemos instalarlo como un paquete del sistema:

```
pip install Flask
```

Esto instalará además otras dependencias adicionales, como *werkzeug*, que se utiliza para establecer una interfaz de comunicación entre el servidor y los clientes que se conecten a él.

Un primer ejemplo sencillo de prueba podría ser éste:

```
from flask import Flask

app = Flask(__name__)

@app.route('/', methods=['GET', 'POST'])
def home():
    return "¡Hola, Flask!"

if __name__ == '__main__':
    app.run(debug=True)
```

Expliquemos algunas de las líneas de este código:

1. El objeto `app` de la clase Flask es nuestra aplicación WSGI, que nos permitirá posteriormente desplegar nuestra aplicación en un servidor Web. Se le pasa como parámetro el módulo actual (`__name__`).
2. El decorador `route` nos permite filtrar la petición HTTP recibida, de tal forma que si la petición se realiza a la URL `/` se ejecutará la función vista `hello_word`.
3. La función “home” que se ejecuta devuelve una respuesta HTTP. En este caso devuelve una cadena de caracteres que se será los datos de la respuesta, pero podría ser la renderización de un documento HTML.
4. Finalmente, si ejecutamos este módulo se pone en marcha el método `run` que ejecuta un servidor web para que podamos probar la aplicación.

Puesta en marcha del servidor

Suponiendo que hayamos llamado `app.py` a nuestro servidor, para ponerlo en marcha podemos ejecutar este comando desde terminal (o ejecutar Run desde el entorno de desarrollo):

```
flask --app app run
```

Después podemos acceder a *http://localhost:5000* para ver el mensaje (por defecto Flask escucha en dicho puerto 5000 al iniciar). Una forma de modificar el puerto de escucha ponerlo en marcha es con un código como este:

```
if __name__ == '__main__':
    app.run(port=80)
```

Nótese que el método run admite un parámetro port para poder establecer el puerto por el que escuchar. Bastaría entonces con ejecutar un programa como éste con el correspondiente comando python:

```
python app.py
```

4. Enrutamiento

El objeto Flask `app` nos proporciona un decorador `router` que es capaz de filtrar la función que se va ejecutar analizando la petición HTTP, fundamentalmente por la ruta y el método que se hace la petición (parámetro “methods”).

En Flask, una ruta define cómo responde la aplicación a una solicitud (request) realizada por un cliente (normalmente un navegador o una aplicación cliente) a una URL concreta. Las rutas están asociadas a funciones de Python, llamadas vistas (view functions), que devuelven una respuesta.

4.1. Definición de rutas con @app.route

Flask utiliza decoradores para mapear URLs a funciones. El decorador `@app.route()` se utiliza para asociar una URL con una función que será ejecutada cuando se acceda a esa ruta.

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def home():
    return 'Bienvenido a la aplicación Flask'

@app.route('/saludo')
```

```
def saludo():
    return 'Hola desde Flask'
```

Notas importantes:

- La función puede llamarse como quieras, pero debe devolver una respuesta (generalmente un `str`, `dict`, `Response`, o resultado de `render_template`).
- La ruta `'/'` representa la raíz del sitio.
- Las rutas por defecto usan el método GET (ver más adelante).

4.2. Métodos HTTP: GET, POST, PUT, DELETE

Flask permite especificar qué métodos HTTP se aceptan en cada ruta mediante el parámetro methods del decorador. **Ejemplo con métodos GET y POST:**

```
from flask import request

@app.route('/formulario', methods=['GET', 'POST'])
def formulario():
    if request.method == 'POST':
        nombre = request.form['nombre']
        return f'Hola, {nombre}'
    return '''
        <form method="post">
            Nombre: <input type="text" name="nombre">
            <input type="submit" value="Enviar">
        </form>
    '''
```

Podemos definir rutas para múltiples métodos:

```
@app.route('/usuario', methods=['GET', 'POST', 'PUT', 'DELETE'])
def usuario():
    if request.method == 'GET':
        return "Obteniendo usuario"
    elif request.method == 'POST':
        return "Creando usuario"
    elif request.method == 'PUT':
        return "Actualizando usuario"
    elif request.method == 'DELETE':
        return "Eliminando usuario"
```

4.3. Procesar peticiones: parámetros en URL y formularios

Para recoger datos que nos lleguen en las peticiones de los clientes (campos de formularios, o incluso ficheros), necesitamos utilizar el elemento `request` de Flask, así que lo importaremos también al inicio:

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

Parámetros en la URL (routing dinámico): podemos definir rutas dinámicas con `<nombre>`:

```
@app.route('/saludo/<nombre>')
def saludo_personalizado(nombre):
    return f'Hola, {nombre}'

@app.route("/articulos/<int:id>")
def mostrar_ariculo(id):
    return f'Articulo con ID: {id}'
```

Tipos de conversión en las rutas:

- `<int:id>` — fuerza a que el valor sea entero.
- `<float:precio>` — número con decimales.
- `<path:ruta>` — permite `/` dentro del parámetro.
- `<string:texto>` — valor por defecto (texto sin `/`).

Parámetros en la query string (GET): accesibles con `request.args`:

```
@app.route('/buscar')
def buscar():
    query = request.args.get('q', '')
    return f'Buscando: {query}'
```

TIP: Accede a `/buscar?q=flask` para probarlo. El segundo parámetro del método get permite indicar un valor por defecto en caso de que no se encuentre el parámetro buscado.

Parámetros en formularios (POST):

```
@app.route('/login', methods=['POST'])
def login():
    usuario = request.form.get('usuario')
    clave = request.form.get('clave')
    return f'Usuario: {usuario}, Clave: {clave}'
```

El envío de datos por POST no se hace en la propia línea de la URL, sino en el cuerpo de la petición. En este caso podemos utilizar el mapa `request.form` para acceder a los campos.

Si estamos subiendo algún fichero podemos acceder a él por su nombre de campo de formulario en el mapa `request.files`. De forma opcional también podemos guardarlo en la ubicación que queramos, usando el método `save` del propio objeto. Podemos utilizar el nombre original del fichero con su propiedad `filename`.

```
fichero = request.files['nombre_fichero']
fichero.save('/home/usuario/ficheros/' + fichero.filename)
```

4.4. Uso de request, redirect y abort

- `request`: objeto que representa la solicitud HTTP, incluyendo datos del formulario, cabeceras, método, etc.
- `redirect`: permite redirigir a otra ruta:
- `abort`: interrumpe el flujo devolviendo un código de error. Es posible personalizar los errores con manejadores de error (@app.errorhandler).

```
from flask import request, abort

@app.route('/info')
def info():
    user_agent = request.headers.get('User-Agent')
    return f'Usas: {user_agent}'

@app.route('/inicio')
def inicio():
    return redirect('/bienvenida')

@app.route('/admin')
def admin():
    abort(403)  # Forbidden
```

5. Gestión de vistas y plantillas

En aplicaciones web, necesitamos generar contenido HTML dinámico. Flask utiliza **Jinja2** como su motor de plantillas por defecto del lado servidor, una poderosa herramienta que permite combinar lógica sencilla de presentación con HTML. Las plantillas permiten separar la **lógica de la aplicación** (Python) de la **interfaz de usuario** (HTML), siguiendo el patrón **MVC** (Modelo-Vista-Controlador) o al menos su espíritu. El uso de plantillas es fundamental en Flask

cuando queremos presentar datos en páginas web, generar formularios, mostrar resultados de modelos IA o visualizar respuestas dinámicas al usuario.

Jinja2 es un motor de plantillas para Python inspirado en Django, que permite incrustar código Python-like dentro de archivos HTML. Flask lo integra de forma nativa mediante la función `render_template()`.

Estructura de un proyecto con plantillas

Para que Flask encuentre y procese las plantillas, deben guardarse en una carpeta especial llamada `templates`, situada en el mismo directorio que el script principal (`app.py` o similar).

```
/mi_app/
│
├── app.py
└── templates/
    ├── base.html
    ├── index.html
    └── saludo.html
```

Uso de `render_template()`: esta función permite cargar una plantilla y pasarle datos desde Python. Internamente busca el archivo en la carpeta `templates`.

```python
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/')
def inicio():
    return render_template('index.html', nombre='Josemi')
```

```html
<!-- templates/index.html -->
<!DOCTYPE html>
<html>
<head><title>Página de inicio</title></head>
<body>
    <h1>Bienvenido, {{ nombre }}</h1>
</body>
</html>
```

5.1. Sintaxis básica de Jinja2

La sintaxis de Jinja2 es el conjunto de reglas especiales que te permiten escribir archivos de texto (las plantillas) que contienen una mezcla de:

- HTML estático: El código HTML normal que define la estructura básica de tu página web (los encabezados, los párrafos, las listas, etc.).
- Instrucciones dinámicas: Marcadores especiales que le dicen a Jinja2 cómo insertar datos dinámicos, ejecutar lógica de programación (como mostrar diferentes cosas según ciertas condiciones o repetir elementos), y manipular la información antes de que se envíe al navegador del usuario.

En esencia, la sintaxis de Jinja2 actúa como un puente entre el código Python (donde reside la lógica de la aplicación y los datos) y los archivos HTML (donde se muestra esa información de manera atractiva).

a) Inserción de variables: `{{ ... }}`

Permite mostrar valores desde Flask:

```
<p>Tu nombre es: {{ nombre }}</p>
```

Internamente, Jinja2 convierte esto a una llamada `str(nombre)` escapando HTML por seguridad (evita XSS). Podemos acceder a los atributos de los objetos utilizando la notación de punto (.) o mediante corchetes ([]) para acceder a elementos de diccionarios o listas.

```
# En código Python
contexto = {
    'user': {'name': 'Ana', 'age': 30},
    'items': ['manzana', 'plátano', 'naranja']
}

# En Jinja2
<p>Nombre: {{ user.name }}</p>
<p>Edad: {{ user['age'] }}</p>
<ul>
{% for fruta in items %}
    <li>{{ fruta }}</li>
{% endfor %}
</ul>
```

b) Estructuras de control: `{% ... %}`

Permiten condicionales y bucles en la plantilla.

Condicionales:

```
{% if usuario %}
  <p>Hola {{ usuario }}</p>
```

```
{% else %}
  <p>Invitado</p>
{% endif %}
```

Bucles: en los bucles, Jinja2 permite variables útiles como `loop.index`, `loop.first`, etc.

```
<ul>
{% for item in lista %}
  <li>{{ item }}</li>
{% endfor %}
</ul>
```

c) Filtros

Jinja2 incluye filtros para transformar datos: `{{ variable|filtro }}`

Ejemplos:

```
<p>{{ nombre|upper }}</p>        <!-- MAYÚSCULAS -->
<p>{{ fecha|date('d/m/Y') }}</p> <!-- Formato de fecha -->
<p>{{ comentario|safe }}</p>     <!-- No escapa HTML -->
```

Listado de filtros comunes:

- `upper`, `lower`, `capitalize`
- `length`
- `safe`, `escape`
- `join`, `replace`, `truncate`

5.2. Inclusión de archivos estáticos (CSS, JS, imágenes)

Flask espera que los archivos estáticos (como hojas de estilo, scripts JS o imágenes) se almacenen en una carpeta llamada `static`.

```
/mi_app/
│
├── app.py
├── templates/
│   └── index.html
└── static/
    ├── css/
    │   └── estilos.css
    └── img/
        └── logo.png
```

Para referenciarlos desde las plantillas se usa `url_for('static', filename='ruta')`.

Ejemplo:

```
<!-- HTML -->
<link rel="stylesheet" href="{{ url_for('static',
filename='css/estilos.css') }}">
<img src="{{ url_for('static', filename='img/logo.png') }}"
alt="Logo">
```

6. Gestión de formularios en Flask

El manejo de formularios es uno de los pilares fundamentales en cualquier aplicación web. Flask ofrece varias formas de trabajar con ellos, desde la **gestión manual** hasta el uso de bibliotecas como **Flask-WTF** que automatizan y facilitan tareas como la validación o la protección contra ataques.

6.1. Recepción de datos desde formularios HTML

Cuando un usuario envía un formulario, el navegador genera una **solicitud HTTP**, normalmente de tipo **POST** (aunque también puede ser GET). En Flask, podemos acceder a los datos enviados mediante el objeto `request`, importado desde `flask`.

Formulario HTML:

```html
<!-- templates/formulario.html -->
<form method="post" action="/procesar">
  <label>Nombre: <input type="text" name="nombre"></label><br>
  <label>Email: <input type="email" name="email"></label><br>
  <input type="submit" value="Enviar">
</form>
```

Código Flask:

```python
from flask import Flask, request, render_template

app = Flask(__name__)

@app.route('/formulario')
def formulario():
    return render_template('formulario.html')

@app.route('/procesar', methods=['POST'])
def procesar():
    nombre = request.form.get('nombre')
    email = request.form.get('email')
    return f"Nombre recibido: {nombre}, Email: {email}"
```

Detalles importantes:

- `request.form` es un **diccionario inmutable** que contiene los datos del formulario enviados por POST.

- Usar `request.form.get('clave')` es preferible a `request.form['clave']` porque el primero devuelve `None` si la clave no existe (y no lanza excepción).
- En formularios GET (no recomendados para datos sensibles), se usaría `request.args`.

6.2. Validación manual de formularios

Validar los datos del formulario es esencial para:

- Garantizar que los campos obligatorios están presentes.
- Verificar que los valores tienen el formato adecuado (email, números, etc.).
- Proteger contra ataques (por ejemplo, inyección o XSS).

Validación básica en Flask (sin librerías externas)

```
@app.route('/procesar', methods=['POST'])
def procesar():
    nombre = request.form.get('nombre', '').strip()
    email = request.form.get('email', '').strip()

    errores = []

    if not nombre:
        errores.append("El nombre es obligatorio.")
    if not email or '@' not in email:
        errores.append("El email debe ser válido.")

    if errores:
        return render_template('formulario.html',
errores=errores, nombre=nombre, email=email)

    return f"Datos recibidos correctamente: {nombre} - {email}"
```

Plantilla con errores:

```
<!-- templates/formulario.html -->
{% if errores %}
  <ul style="color: red;">
    {% for error in errores %}
      <li>{{ error }}</li>
    {% endfor %}
  </ul>
{% endif %}

<form method="post" action="/procesar">
```

```
  <label>Nombre: <input type="text" name="nombre" value="{{
nombre }}"></label><br>
  <label>Email: <input type="email" name="email" value="{{ email
}}"></label><br>
  <input type="submit" value="Enviar">
</form>
```

6.3. Introducción a Flask-WTF (ampliación)

Flask-WTF es una extensión de Flask basada en **WTForms**, que simplifica la creación y validación de formularios en Python. Proporciona:

- Declaración de formularios como clases Python.
- Validación automática y extensible.
- Soporte integrado para CSRF.
- Widgets HTML y renderizado de campos.

Instalación:

```
pip install flask-wtf
```

Configuración básica:

```
from flask import Flask
from flask_wtf import FlaskForm
from wtforms import StringField, EmailField, SubmitField
from wtforms.validators import DataRequired, Email

app = Flask(__name__)
app.secret_key = 'clave_secreta'  # Requisito para CSRF

class ContactoForm(FlaskForm):
    nombre = StringField('Nombre', validators=[DataRequired()])
    email = EmailField('Email', validators=[DataRequired(),
Email()])
    enviar = SubmitField('Enviar')
```

Vista y plantilla asociadas:

```
from flask import render_template

@app.route('/contacto', methods=['GET', 'POST'])
def contacto():
    form = ContactoForm()
    if form.validate_on_submit():
```

```
        return f"Recibido: {form.nombre.data} - {form.email.data}"
    return render_template('contacto.html', form=form)
```

Plantilla HTML:

```
<!-- templates/contacto.html -->
<form method="post">
  {{ form.hidden_tag() }}  <!-- Necesario para el token CSRF -->
  <p>{{ form.nombre.label }} {{ form.nombre }}</p>
  <p>{{ form.email.label }} {{ form.email }}</p>
  <p>{{ form.enviar() }}</p>
</form>

{% if form.errors %}
  <ul style="color: red;">
    {% for campo, errores in form.errors.items() %}
      {% for error in errores %}
        <li>{{ campo }}: {{ error }}</li>
      {% endfor %}
    {% endfor %}
  </ul>
{% endif %}
```

6.4. Seguridad en formularios: CSRF tokens

CSRF (*Cross-Site Request Forgery*) es un ataque que aprovecha sesiones activas para ejecutar acciones maliciosas. Por ejemplo, un usuario logueado puede ser inducido a hacer clic en un enlace que realice una acción sin su consentimiento.

¿Cómo protege Flask contra CSRF?

- Si usas formularios **HTML manuales**, deberías generar e incluir un token CSRF por ti mismo.
- Si usas **Flask-WTF**, la protección CSRF viene activada por defecto.

Requisitos:

1. Tener configurado `secret_key` en la aplicación.
2. Incluir `form.hidden_tag()` en la plantilla para que se inserte el token oculto.
3. Asegurarte de que las rutas POST verifiquen el token.

Ejemplo con Flask-WTF:

```
from flask_wtf.csrf import CSRFProtect

csrf = CSRFProtect(app)
```

Esto aplica CSRF a todas las rutas que aceptan POST automáticamente.

¿Y si no se usa Flask-WTF?

Deberías implementar manualmente la generación y comprobación del token. Sin embargo, no es recomendable, dado que Flask-WTF lo gestiona de forma sencilla, segura y transparente.

Buenas prácticas en la gestión formularios

- Usa POST para modificar datos; GET solo para consultas.
- Nunca confíes en los datos del cliente: siempre valida del lado del servidor.
- Mantén separados el contenido y la lógica del formulario.
- Protege los formularios con CSRF, especialmente si manejan datos sensibles.
- Escapa los valores de entrada si los muestras posteriormente (aunque Flask lo hace por defecto).
- En formularios complejos, considera Flask-WTF para mayor robustez y claridad.

7. Integración de modelos de IA en Flask

Una de las aplicaciones más comunes de Flask en el contexto de la Inteligencia Artificial es como **servidor web que encapsula un modelo previamente entrenado** y lo expone a través de formularios o interfaces web para realizar predicciones.

7.1. Cargar modelos entrenados (Pickle, scikit-learn, Keras...)

Una vez entrenado un modelo de ML o DL en Python (por ejemplo con **scikit-learn**, **TensorFlow/Keras** o **XGBoost**), se puede **guardar en disco** y luego **cargarlo desde Flask** para su uso en predicción.

a) Usando `pickle` con scikit-learn

```
import pickle
with open('modelo_clasificador.pkl', 'rb') as f:
    modelo = pickle.load(f)
```

ATENCIÓN: Cargar modelos con pickle puede ser inseguro si no confías en la fuente del archivo. No abras modelos no verificados.

b) Usando `joblib` (recomendado por scikit-learn)

```
import joblib
modelo = joblib.load('modelo_clasificador.joblib')
```

c) Usando `Keras` (redes neuronales)

```
from tensorflow.keras.models import load_model
modelo = load_model('modelo_clasificador.h5')
```

Los modelos de Keras se cargan como objetos funcionales o secuenciales, y requieren procesamiento adecuado de entrada (arrays NumPy con las dimensiones esperadas).

7.2. Renderizar predicciones desde servidor Flask (no API)

En lugar de usar Flask como API REST, en este enfoque la predicción se hace directamente en una **función de ruta (`@app.route`)** y se devuelve como parte de una página HTML renderizada (con Jinja2).

Ejemplo de flujo:

1. Usuario accede a un formulario HTML.
2. Rellena campos con datos de entrada.

3. Flask recibe la solicitud POST.
4. Flask procesa los datos, ejecuta la predicción.
5. El resultado se muestra en una nueva página o en la misma.

7.3. Procesamiento de datos de entrada

Antes de usar los datos en el modelo, es necesario replicar el **pipeline de preprocesamiento** usado en entrenamiento. Esto incluye:

a) Conversión de tipos

Los datos llegan como cadenas (str). Debes convertirlos a float, int, etc.

```
edad = float(request.form.get('edad'))
```

b) Codificación de variables categóricas: si el modelo usó OneHotEncoder, LabelEncoder u otra técnica, hay que reproducirla exactamente.

c) Escalado de variables: si se usó StandardScaler, MinMaxScaler, etc., se debe cargar el objeto transformador y aplicarlo antes de la predicción.

```
scaler = joblib.load('scaler.pkl')
X_scaled = scaler.transform([[edad, ingresos]])
```

Es recomendable **almacenar y reutilizar** los objetos de transformación junto con el modelo.

d) Validación básica: antes de pasar al modelo, se pueden añadir controles como:

```
if edad < 0 or edad > 120:
    return "Edad fuera de rango"
```

Esto evita errores en predicción y protege al modelo.

7.4. Ejemplo completo: servir un modelo de clasificación

Supongamos que tenemos un modelo de scikit-learn para predecir si una persona aprueba o no un examen, en función de horas de estudio y horas de sueño.

Estructura:

```
proyecto/
├── app.py
├── modelo_clasificador.joblib
├── scaler.joblib
└── templates/
    ├── formulario.html
    └── resultado.html
```

app.py

```python
from flask import Flask, render_template, request
import joblib
import numpy as np

app = Flask(__name__)

# Cargar modelo y escalador
modelo = joblib.load('modelo_clasificador.joblib')
escalador = joblib.load('scaler.joblib')

@app.route('/')
def formulario():
    return render_template('formulario.html')

@app.route('/resultado', methods=['POST'])
def resultado():
    try:
        estudio = float(request.form['estudio'])
        sueno = float(request.form['sueno'])

        datos = np.array([[estudio, sueno]])
        datos_escalados = escalador.transform(datos)
        prediccion = modelo.predict(datos_escalados)

        resultado = "Aprobado" if prediccion[0] == 1 else "Suspendido"

        return render_template('resultado.html', resultado=resultado)

    except Exception as e:
        return f"Error en la predicción: {str(e)}"
```

formulario.html

```
<h2>Predicción de aprobado</h2>
<form method="post" action="/resultado">
  <label>Horas de estudio: <input type="number" name="estudio" step="0.1" required></label><br>
  <label>Horas de sueño: <input type="number" name="sueno" step="0.1" required></label><br>
  <input type="submit" value="Predecir">
</form>
```

resultado.html

```
<h2>Resultado de la predicción</h2>
<p>{{ resultado }}</p>
<a href="/">Volver</a>
```

7.5. Buenas prácticas: control de errores, logging, timeouts

a) Control de errores

- Usa try/except alrededor de la lógica de predicción.
- Gestiona errores de conversión de datos, fallos del modelo, errores en el formulario.
- Muestra mensajes amigables al usuario.

b) Logging

Utiliza el módulo logging para registrar errores, predicciones, tiempos de ejecución:

```
import logging
logging.basicConfig(level=logging.INFO)

@app.route('/resultado', methods=['POST'])
def resultado():
    try:
        ...
            logging.info(f"Predicción recibida: estudio={estudio}, sueno={sueno}, predicción={resultado}")
        ...
    except Exception as e:
        logging.error(f"Error: {str(e)}")
        ...
```

c) Timeouts y control de recursos

Aunque Flask no gestiona timeouts por sí solo, conviene:

- Usar servidores WSGI como **Gunicorn** para producción, donde puedes definir --timeout.
- Evitar cargar modelos pesados o volver a cargarlos en cada petición.
- Predecir de forma **síncrona y rápida** para evitar bloqueos.

8. Cuestiones adicionales

En este apartado veremos algunas cuestiones adicionales que nos pueden venir bien a la hora de trabajar con Flask, como la gestión de orígenes cruzados (*cross-origins*), o la definición de plantillas y vistas.

8.1. Puesta en producción

La forma de iniciar el servidor que hemos visto en un apartado anterior nos va a servir para el período de pruebas y desarrollo. De hecho, podremos ver un mensaje al arrancar el servidor que nos indica que esa no es la forma recomendada de poner en marcha el servidor en producción.

Para el despliegue en producción tenemos varias alternativas, y algunas de ellas las podemos consultar en la documentación oficial. Por ejemplo, podemos instalar un servidor WSGI (*Web Server Gateway Interface*) que haga de pasarela con Flask, como puede ser `waitress`:

```
pip install waitress
```

Después de eso podemos lanzar nuestra aplicación en dicho servidor de este modo:

```
from flask import Flask, request

...

if __name__ == '__main__':
    from waitress import serve
    # El parámetro 0.0.0.0 indica que la aplicación estará disponible para
    # cualquier dirección IP en el servidor (útil si queremos acceder desde
    # fuera del servidor local)
    serve(app, host='0.0.0.0', port=80)
```

8.2. Respuesta en JSON

Aunque Flask puede utilizarse para enviar distintos tipos de respuesta, e incluso renderizar páginas HTML o vistas jerarquizadas en plantillas, en este apartado explicaremos cómo enviar contenido en formato JSON, que pueda ser utilizado por cualquier cliente que acceda al servidor para obtener respuestas del modelo que estemos utilizando.

Para poder trabajar con datos en formato JSON podemos incorporar el módulo `jsonify` de Flask. Una vez hecho esto podemos transformar un diccionario Python en cadena JSON fácilmente:

```
def funcion():
    datos = {
        'usuario': 'nacho',
        'edad': 45
    }

    return jsonify(datos)
```

Sin embargo, Flask es capaz de convertir a formato JSON directamente un diccionario, así que podemos devolver directamente el diccionario (`return datos` en el ejemplo anterior).