13 Designing and Implementing Large-Scale, Robust ML Software

So far, we have learned how to develop ML models, how to work with data, and how to create and test the entire ML pipeline. What remains is to learn how we can integrate these elements into a **user interface** (**UI**) and how to deploy it so that they can be used without the need to program. To do so, we'll learn how to deploy the model complete with a UI and the data storage for the model.

In this chapter, we'll learn how to integrate the ML model with a graphical UI programmed in Gradio and storage in a database. We'll use two examples of ML pipelines – an example of the model for predicting defects from our previous chapters and a generative AI model to create pictures from a natural language prompt.

In this chapter, we're going to cover the following main topics:

- ML is not alone – elements of a deployed ML-based system
- The UI of an ML model
- Data storage
- Deploying an ML model for numerical data
- Deploying a generative ML model for images
- Deploying a code completion model as an extension to Visual Studio Code

ML is not alone

Chapter 2 introduced several elements of an ML system – storage, data collection, monitoring, and infrastructure, just to name a few of them. We need all of them to deploy a model for the users, but not all of them are important for the users directly. We need to remember that the users are interested in the results, but we need to pay attention to all details related to the development of such systems. These activities are often called AI engineering.

The UI is important as it provides the ability to access our models. Depending on the use of our software, the interface can be different. So far, we've focused on the models themselves and on the data that is used to train the models. We have not focused on the usability of models and how to integrate them into the tools.

By extension, as for the UI, we also need to talk about storing data in ML. We can use **comma-separated values** (**CSV**) files, but they quickly become difficult to handle. They are either too large to read into memory or too cumbersome for version control and exchanging data.

Therefore, in this chapter, we'll focus on making the ML system usable. We'll learn how to develop a UI, how to link the system to the database, and how to design a Visual Studio Code extension that can complete code in Python.

The UI of an ML model

A UI serves as the bridge between the intricate complexities of ML algorithms and the end users who interact with the system. It is the interactive canvas that allows users to input data, visualize results, control parameters, and gain insights from the ML model's outputs. A well-designed UI empowers users, regardless of their technical expertise, to harness the potential of ML for solving real-world problems.

Effective UIs for ML applications prioritize clarity, accessibility, and interactivity. Whether the application is aimed at business analysts, healthcare professionals, or researchers, the interface should be adaptable to the user's domain knowledge and objectives. Clear communication of the model's capabilities and limitations is vital, fostering trust in the technology and enabling users to make informed decisions based on its outputs. Hence my next best practice.

Best practice #66

Focus on the user task when designing the UI of the ML model.

We can use different types of UIs, but the majority of modern tools gravitate around two – web-based interfaces (which require thin clients) and extensions (which provide in-situ improvements). ChatGPT is an example of the web-based interface to the GPT-4 model, while GitHub CoPilot is an example of the extension interface to the same model.

In the first example, let's look at how easy it is to deploy an ML app using the Gradio framework. Once we have prepared a pipeline for our model, we just need a handful of lines of code to make the app. Here are the lines, based on the example of a model that exists at Hugging Face, for text classification:

```
import gradio as gr
from transformers import pipeline

pipe = pipeline("text-classification")

gr.Interface.from_pipeline(pipe).launch()
```

The first two lines import the necessary libraries– one for the UI (Gradio) and one for the pipeline. The second line imports the default text classification pipeline from Hugging Face and the last line creates the UI for the pipeline. The UI is in the form of a website with input and output buttons, as shown in *Figure 13.1*:

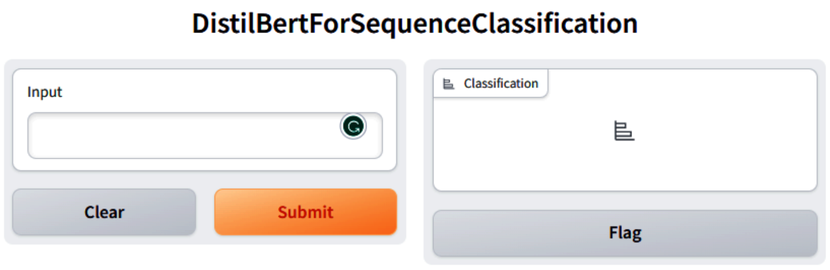


Figure 13.1 – UI for the default text classification pipeline

We can test it by inputting some example text. Normally, we would input this in a script and provide some sort of analysis, but this is done by the Gradio framework for us. We do not even need to link the parameters of the pipeline with the elements of the UI.

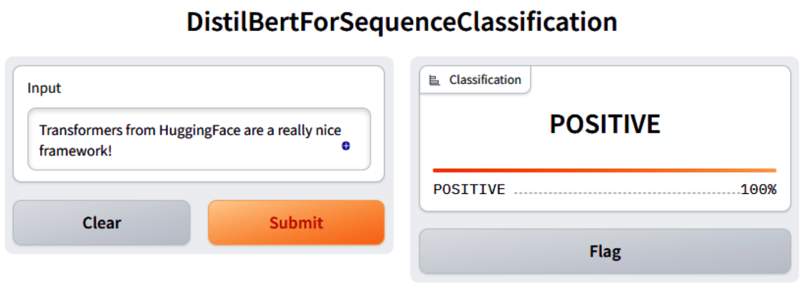
What happens behind the scenes can be explained by observing the output of the script in the console (edited for brevity):

```
No model was supplied, defaulted to distilbert-base-uncased-finetuned-
sst-2-english and revision af0f99b
Using a pipeline without specifying a model name and revision in
production is not recommended.
Downloading (…)lve/main/config.json: 100%|██████████████████| 629/629
[00:00<00:00, 64.6kB/s]
Downloading model.safetensors:
100%|████████████████| 268M/268M [00:04<00:00, 58.3MB/s]
Downloading (…)okenizer_config.json: 100%|████████████████| 48.0/48.0
[00:00<00:00, 20.7kB/s]
```

```
Downloading (…)solve/main/vocab.txt: 100%|████████████████| 232k/232k
[00:00<00:00, 6.09MB/s]
Running on local URL:  http://127.0.0.1:7860
```

The framework has downloaded the default model, its tokenizers, and the vocabulary file and then created the application on the local machine.

The result of using this app is presented in *Figure 13.2*. We input some simple text and almost instantly get its classification:



Transformers from HuggingFace are a really nice framework!

POSITIVE

POSITIVE 100%

Figure 13.2 – Data analyzed using the default text classification pipeline

This kind of integration is a great way to deploy models first and to make sure that they can be used without the need to open a Python environment or similar. With this, we've come to my next best practice.

Best practice #67

Prepare your models for web deployment.

Regardless of what kind of models you develop, try to prepare them for web deployment. Our models can be then packaged as Docker containers and provided as part of a larger system of microservices. Using Gradio is a great example of how such a web deployment can be achieved.

Data storage

So far, we've used CSV files and Excel files to store our data. It's an easy way to work with ML, but it is also a local one. However, when we want to scale our application and use it outside of just our machine, it is often much more convenient to use a real database engine. The database plays a crucial role in an ML pipeline by providing a structured and organized repository for storing, managing, and retrieving data. As ML applications increasingly rely on large volumes of data, integrating a database into the pipeline becomes essential for a few reasons.

Databases offer a systematic way to store vast amounts of data, making it easily accessible and retrievable. Raw data, cleaned datasets, feature vectors, and other relevant information can be efficiently stored in the database, enabling seamless access by various components of the ML pipeline.

In many ML projects, data preprocessing is a critical step that involves cleaning, transforming, and aggregating data before feeding it to the model. Databases allow you to store intermediate preprocessed data, reducing the need to repeat resource-intensive preprocessing steps each time the model is trained. This speeds up the overall pipeline and maintains data consistency.

ML pipelines often involve data from diverse sources such as sensors, APIs, files, and databases. Having a centralized database simplifies the process of integrating different data streams, ensuring that all relevant information is readily available for training and inference.

Even maintaining a record of different dataset versions is important for reproducibility and tracking changes. Databases can be used to store different versions of datasets, making it easier to roll back to previous versions if needed and facilitating collaboration among team members.

Finally, ML applications that handle large-scale data require efficient data management to scale effectively. Databases provide mechanisms for indexing, partitioning, and optimizing queries, which enhance performance and allow the pipeline to handle increasing data volumes.

So, let's create a database in SQLite that would contain the same numerical data that we used in our previous work:

```
# create the database
import sqlite3

conn = sqlite3.connect('ant13.db')

c = conn.cursor()
```

In the preceding code fragment, we use the `sqlite3` engine to create a database and connect to it (`sqlite3.connect`). Once we connect to a database, we need a cursor to move around in the database and execute our queries. The next step is to import our existing data into the database.

Now, we can open the Excel file and transfer the data to the database:

```
# read the excel file with the data
# and save the data to the database
import pandas as pd

# read the excel file
df = pd.read_excel('chapter_12.xlsx', sheet_name='ant_1_3')

# print the first 5 rows
print(df.head())

# create the engine that we use to connect to the database to
# save the data
engine = create_engine('sqlite:///ant13.db')
```

```
# save the dataframe to the database
df.to_sql('ant_1_3', engine, index=False, if_exists='replace')
```

The preceding code reads data from an Excel file, processes it using the pandas library, and then saves the processed data into an SQLite database. First, the code reads an Excel file called `'chapter_12.xlsx'` and extracts data from the `'ant_1_3'` sheet. The data is loaded into a pandas DataFrame, `df`. Then, the code uses the `create_engine` function from the `sqlalchemy` library to establish a connection to an SQLite database. It then creates a connection to a database file named `'ant13.db'`.

Then, it uses the built-in `to_sql` function to create a database table based on the DataFrame. In this example, the function has the following parameters:

- `'ant_1_3'` is the name of the table in the database where the data will be stored.
- `engine` is the connection to the SQLite database that was created earlier.
- `index=False` specifies that the DataFrame index should not be saved as a separate column in the database.
- `if_exists='replace'` indicates that if a table named `'ant_1_3'` already exists in the database, it should be replaced with the new data. Other options for `if_exists` include `append` (add data to the table if it exists) and `fail` (raise an error if the table already exists).

After this, we have our data in a database and can easily share the data across multiple ML pipelines. However, in our case, we'll only demonstrate how to extract such data into a DataFrame so that we can use it in a simple ML application:

```
# select all rows from that database
data = engine.execute('SELECT * FROM ant_1_3').fetchall()
# and now, let's create a dataframe from that data
df = pd.DataFrame(data)

# get the names of the columns from the SQL database
# and use them as the column names for the dataframe
df.columns = [x[0] for x in engine.description]

# print the head of the dataframe
df.head()
```

The `'SELECT * FROM ant_1_3'` query selects all columns from the `'ant_1_3'` table in the database. The `fetchall()` method retrieves all the rows returned by the query and stores them in the data variable. The data variable will be a list of tuples, where each tuple represents a row of data.

Then, it creates a pandas DataFrame, `df`, from the data list. Each tuple in the list corresponds to a row in the DataFrame, and the columns of the DataFrame will be numbered automatically. Finally, the code retrieves the names of the columns in the original database table. The `engine.description` attribute holds metadata about the result of the executed SQL query. Specifically, it provides information about the columns returned by the query. The code then extracts the first element of each tuple in `engine.description`, which is the column name, and assigns these names to the columns of the DataFrame, `df`.

From there, the workflow with the data is just as we know it – it uses a pandas DataFrame.

In this example, the entire DataFrame fits in the database and the entire database can fit into one frame. However, this is not the case for most ML datasets. The pandas library has limitations in terms of its size, so when training models such as GPT models, we need more data than a DataFrame can hold. For that, we can use either the Dataset library from Hugging Face, or we can use databases. We can only fetch a limited amount of data, train a neural network on it, validate on another data, then fetch a new set of rows, train the neural network a bit more, and so on.

In addition to making the database on files, which can be a bit slow, the SQLite library allows us to create databases in memory, which is much faster, but they do not get serialized to our permanent storage – we need to take care of that ourselves.

To create an in-memory database, we can simply change the name of the database to `:memory:` in the first script, like this:

```
conn = sqlite3.connect(':memory:')

c = conn.cursor()
```

We can use it later on in a similar way, like so:

```
# create the enginve that we use to connect to the database to
# save the data
engine = create_engine('sqlite:///:memory:')

# save the dataframe to the database
df.to_sql('ant_1_3', engine, index=False, if_exists='replace')
```

In the end, we need to remember to serialize the database to a file; otherwise, it will disappear the moment our system closes:

```
# serialize to disk
c.execute("vacuum main into 'saved.db'")
```

Using databases together with ML is quite simple if we know how to work with DataFrames. The added value, however, is quite large. We can serialize data to files, read them into memory, manipulate them, and serialize them again. We can also scale up our applications beyond one system and use these systems online. However, for that, we need a UI.

With that, we've come to my next best practice.

Best practice #68

Try to work with in-memory databases and dump them to disk often.

Libraries such as pandas have limitations on how much data they can contain. Databases do not. Using an in-memory database provides a combination of the benefits of both without these limitations. Storing the data in memory enables fast access, and using the database engine does not limit the size of the data. We just need to remember to save (dump) the database from the memory to the disk once in a while to prevent the loss of data in case of exceptions, errors, defects, or equipment failures.

Deploying an ML model for numerical data

Before we create the UI, we need to define a function that will take care of making predictions using a model that we trained in the previous chapter. This function takes the parameters as a user would see them and then makes a prediction. The following code fragment contains this function:

```
import gradio as gr
import pandas as pd
import joblib

def predict_defects(cbo,
                    dcc,
                    exportCoupling,
                    importCoupling,
                    nom,
                    wmc):

    # we need to convert the input parameters to floats to use them in
the prediction
    cbo = float(cbo)
    dcc = float(dcc)
    exportCoupling = float(exportCoupling)
    importCoupling = float(importCoupling)
    nom = float(nom)
    wmc = float(wmc)

    # now, we need to make a data frame out of the input parameters
    # this is necessary because the model expects a data frame
    # we create a dictionary with the column names as keys
    # and the input parameters as values
    # please note that the names of the features must be the same as
in the model
```

```
    data = {
        'CBO': [cbo],
        'DCC': [dcc],
        'ExportCoupling': [exportCoupling],
        'ImportCoupling': [importCoupling],
        'NOM': [nom],
        'WMC': [wmc]
    }

    # we create a data frame from the dictionary
    df = pd.DataFrame(data)

    # load the model
    model = joblib.load('./chapter_12_decision_tree_model.joblib')

    # predict the number of defects
    result = model.predict(df)[0]

    # return the number of defects
    return result
```

This fragment starts by importing three libraries that are important for the UI and the modeling. We already know about the pandas library, but the other two are as follows:

- `gradio`: This library is used to create simple UIs for interactive ML model testing. The library makes it very easy to create the UI and connect it to the model.
- `joblib`: This library is used for saving and loading Python objects, particularly ML models. Thanks to this library, we do not need to train the model every time the user wants to open the software (UI).

The `predict_defects` function is where we use the model. It is important to note that the naming of the parameters is used automatically by the UI to name the input boxes (as we'll see a bit later). It takes six input parameters: `cbo`, `dcc`, `exportCoupling`, `importCoupling`, `nom`, and `wmc`. These parameters are the same software metrics that we used to train the model. As these parameters are inputted as text or numbers, it is important to convert them into floats, as this was the input value of our model. Once they have been converted, we need to turn these loose parameters into a single DataFrame that we can use as input to the model. First, we must convert it into a dictionary and then use that dictionary to create a DataFrame.

Once the data is ready, we can load the model using the `model = joblib.load('./chapter_12_decision_tree_model.joblib')` command. The last thing we must do is make a prediction using that model. We can do this by writing `result = model.predict(df)[0]`. The function ends by returning the result of the predictions.

There are a few items that are important to note. First, we need a separate function to handle the entire workflow since the UI is based on that. This function must have the same number of parameters as the number of input elements we have on our UI. Second, it is important to note that the names of the columns in the DataFrame should be the same as the names of the columns in the training data (the names are case-sensitive).

So, the actual UI is handled completely by the Gradio library. This is exemplified in the following code fragment:

```
# This is where we integrate the function above with the user
interface
# for this, we need to create an input box for each of the following
parameters:
# CBO, DCC, ExportCoupling,  ImportCoupling,  NOM,  WMC

demo = gr.Interface(fn=predict_defects,
                    inputs = ['number', 'number', 'number', 'number',
'number', 'number'],
                    outputs = gr.Textbox(label='Will contain
defects?',
                                         value= 'N/A'))

# and here we start the actual user interface
# in a browser window
demo.launch()
```

This code fragment demonstrates the integration of the previously defined `predict_defects` function with a UI. Gradio is used to create a simple UI that takes input from the user, processes it using the provided function, and displays the result. The code consists of two statements:

1. Creating the interface using the `gr.Interface` function with the following parameters:

   - `fn=predict_defects`: This argument specifies the function that will be used to process the user input and produce the output. In this case, it's the `predict_defects` function that was defined previously. Please note that the arguments of the function are not provided, and the library takes care of extracting them (and their names) automatically.
   - `inputs`: This argument specifies the types of inputs the interface should expect. In this case, it lists six input parameters, each of the `'number'` type. These correspond to the `cbo`, `dcc`, `exportCoupling`, `importCoupling`, `nom`, and `wmc` parameters in the `predict_defects` function.
   - `outputs`: This argument specifies the output format that the interface should display to the user. In this case, it's a text box labeled "Will contain defects?" with an initial value of `'N/A'`. Since our model is binary, we only use 1 and 0 as the output. To mark the fact that the model has not been used yet, we start with the `'N/A'` label.

2. Launching the interface (`demo.launch()`): This line of code starts the UI in a web browser window, allowing users to interact with it.

The UI that was created using Gradio has input fields where the user can provide values for the software metrics (`cbo`, `dcc`, `exportCoupling`, `importCoupling`, `nom`, `wmc`). Once the user provides these values and submits the form, the `predict_defects` function will be called with the provided input values. The predicted result (whether defects will be present or not) will be displayed in the text box labeled "Will contain defects?".

We can start this application by typing the following in the command prompt:

```
>python app.py
```

This starts a local web server and provides us with the address of it. Once we open the page with the app, we'll see the following UI:

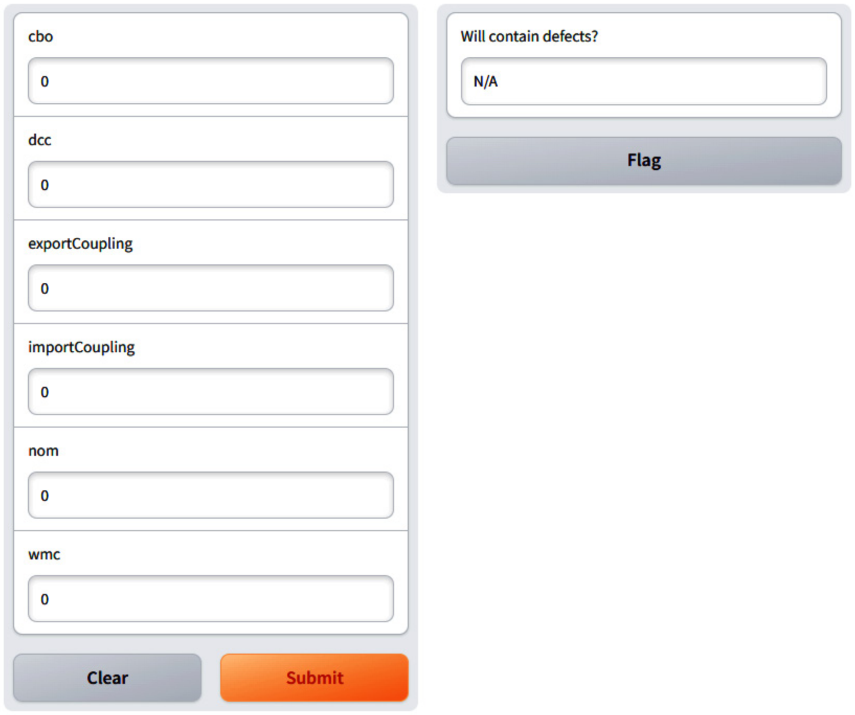
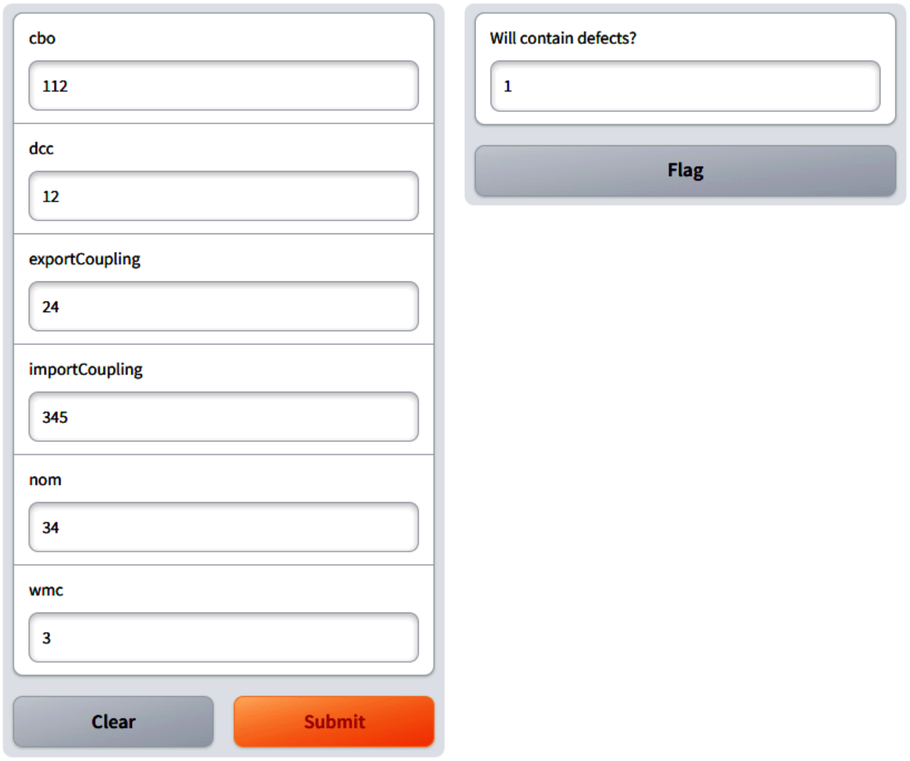

cbo
0
dcc
0
exportCoupling
0
importCoupling
0
nom
0
wmc
0
Will contain defects?
N/A

Figure 13.3 – UI for the defect prediction model created using Gradio

The UI is structured into two columns – the right-hand column with the result and the left-hand column with the input data. At the moment, the input data is the default, and therefore the prediction value is N/A, as per our design.

We can fill in the data and press the **Submit** button to obtain the values of the prediction. This is shown in *Figure 13.4*:



cbo	112
dcc	12
exportCoupling	24
importCoupling	345
nom	34
wmc	3
Will contain defects?	1

Figure 13.4 – UI with the prediction outcome

Once we fill in data to make predictions, we can submit it; at this point, our outcome shows that the module with these characteristics would contain defects. It's also quite logical – for any module that has 345 inputs, we could almost guarantee that there would be some defects. It's just too complex.

This model, and the UI, are only available locally on our computer. We can, however, share it with others and even embed it in websites, if we change just one line. Instead of `demo.launch()` without parameters, we can supply one parameter – `demo.launch(share=True)`.

Although we've used Gradio as an example of the UI, it illustrates that it is rather easy to link an existing model to a UI. We can input the data manually and get a prediction from the model. Whether the UI is programmed in Gradio or any other framework becomes less important. The difficulty may differ – for example, we may need to program the link between the input text boxes and model parameters manually – but the essence is the same.

Deploying a generative ML model for images

The Gradio framework is very flexible and allows for quickly deploying models such as generative AI stable diffusion models – image generators that work similarly to the DALL-E model. The deployment of such a model is very similar to the deployment of the numerical model we covered previously.

First, we need to create a function that will generate images based on one of the models from Hugging Face. The following code fragment shows this function:

```
import gradio as gr
import pandas as pd
from diffusers import StableDiffusionPipeline
import torch

def generate_images(prompt):
    '''
    This function uses the prompt to generate an image
    using the anything 4.0 model from Hugging Face
    '''

    # importing the model from Hugging Face
    model_id = "xyn-ai/anything-v4.0"
    pipe = StableDiffusionPipeline.from_pretrained(model_id,
                                                   torch_dtype=torch.
float16,
                                                   safety_
checker=None)

    # send the pipeline to the GPU for faster processing
    pipe = pipe.to("cuda")

    # create the image here
    image = pipe(prompt).images[0]

    # return the number of defects
    return image
```

This code fragment starts by importing the necessary libraries. Here, we'll notice that there is another library – `diffusers` – which is an interface to image generation networks. The function imports a pre-trained model from the Hugging Face hub. The model is `"xyn-ai/anything-v4.0"`. It is a variant of the Anything 4.0 model, cloned by one of the users. The `StableDiffusionPipeline.from_pretrained()` function is used to load the model as a pipeline for image generation. The `torch_dtype` parameter is set to `torch.float16`, which indicates the data type to be used for computations (lower precision for faster processing).

The image is generated using the pipeline bypassing the prompt as an argument to the `pipe()` function. The generated images are accessed using the `images[0]` attribute. The `prompt` parameter is provided through the parameter of the function, which is supplied by the UI.

The function returns the image, which is then captured by the UI and displayed.

The code for the UI is also quite straightforward once we know the code from the previous example:

```
demo = gr.Interface(fn=generate_images,
                    inputs = 'text',
                    outputs = 'image')

# and here we start the actual user interface
# in a browser window
demo.launch()
```

Compared to the previous example, this code contains only one input parameter, which is the prompt that's used to generate the image. It also has one output, which is the image itself. We use the `'image'` class to indicate that it is an image and should be displayed as such. The output of this model is presented in *Figure 13.5*:

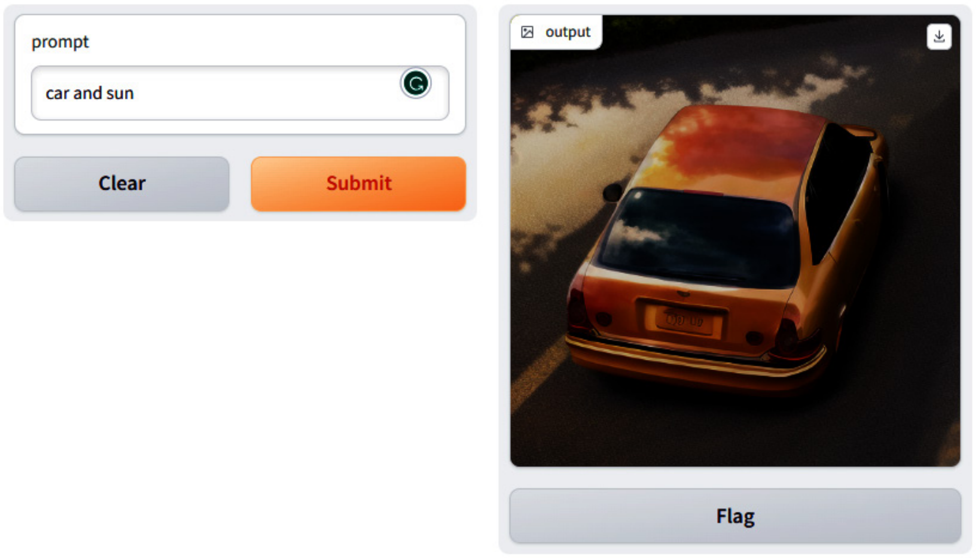


Figure 13.5 – Image generated from the Anything 4.0 model using the "car and sun" prompt

Please note that the model is not perfect as the generated car has distortion artifacts – for example, the right-hand side taillight is not generated perfectly.

Deploying a code completion model as an extension

So far, we've learned how to deploy models online and on the Hugging Face hub. These are good methods and provide us with the ability to create a UI for our models. However, these are standalone tools that require manual input and provide an output that we need to use manually – for example, paste into another tool or save to disk.

In software engineering, many tasks are automated and many modern tools provide an ecosystem of extensions and add-ins. GitHub Copilot is such an add-in to Visual Studio 2022 and an extension to Visual Studio Code – among other tools. ChatGPT is both a standalone web tool and an add-in to Microsoft's Bing search engine.

Therefore, in the last part of this chapter, we'll package our models as an extension to a programming environment. In this section, we learn how to create an extension to complete code, just like GitHub CoPilot. Naturally, we won't use the CodeX model from CoPilot, but Codeparrot's model for the Python programming language. We've seen this model before, so let's dive deeper into the actual extension.

We need a few tools to develop the extension. Naturally, we need Visual Studio Code itself and the Python programming environment. We also need the Node.js toolkit to create the extension. We installed it from nodejs.org. Once we have installed it, we can use Node.js's package manager to install Yeoman and the framework to develop the extension. We can do that by using the following command in the command prompt:

```
npm install -g yo generator-code
```

Once the packages have been installed, we need to create the skeleton code for our extension by typing the following:

```
yo code
```

This will bring up the menu that we need to fill in:

```
     _-----_     ╭──────────────────────────╮
    |       |    │   Welcome to the Visual  │
    |--(o)--|    │   Studio Code Extension  │
   `---------´   │        generator!        │
    ( _´U`_ )    ╰──────────────────────────╯
    /___A___\   /
     |  ~  |
   __'.___.'__
 ´   `  |° ´ Y `

? What type of extension do you want to create? (Use arrow keys)
> New Extension (TypeScript)
  New Extension (JavaScript)
```

```
    New Color Theme
    New Language Support
    New Code Snippets
    New Keymap
    New Extension Pack
    New Language Pack (Localization)
    New Web Extension (TypeScript)
    New Notebook Renderer (TypeScript)
```

We need to choose the first option, which is a new extension that uses Typescript. It is the easiest way to start writing the extension. We could develop a very powerful extension using the language pack and language protocol, but for this first extension, simplicity beats power.

We need to make a few decisions about the setup of our extension, so let's do that now:

```
? What type of extension do you want to create? New Extension
(TypeScript)
? What's the name of your extension? mscopilot
? What's the identifier of your extension? mscopilot
? What's the description of your extension? Code generation using
Parrot
? Initialize a git repository? No
? Bundle the source code with webpack? No
? Which package manager to use? (Use arrow keys)
> npm
  yarn
  pnpm
```

We call our extension `mscopilot` and do not create much additional code – no git repository and no webpack. Again, simplicity is the key for this example. Once the folder has been created, we need one more package from Node.js to interact with Python:

```
npm install python-shell
```

After we click on the last entry, we get a new folder named `mscopilot`; we can enter it with the `code .` command. It opens Visual Studio Code, where we can fill the template with the code for our new extension. Once the environment opens, we need to navigate to the `package.json` file and change a few things. In that file, we need to find the `contributes` section and make a few changes, as shown here:

```
"contributes": {
    "commands": [
      {
        "command": "mscopilot.logSelectedText",
        "title": "MS Suggest code"
      }
    ],
    "keybindings": [
      {
        "command": "mscopilot.logSelectedText",
        "key": "ctrl+shift+l",
        "mac": "cmd+shift+l"
      }
    ]
  },
```

In the preceding code fragment, we added some information stating that our extension has one new function – `logSelectedText` – and that it will be available via the *Ctrl* + *Shift* + *l* key combination on Windows (and a similar one on Mac). We need to remember that the command name includes the name of our extension so that the extension manager knows that this command belongs to our extension. Now, we need to go to the `extension.ts` file and add the code for our command. The code following fragment contains the first part of the code – the setup for the extension and its activation:

```
import * as vscode from 'vscode';

// This method is called when your extension is activated
export function activate(context: vscode.ExtensionContext) {
  // Use the console to output diagnostic information (console.log)
and errors (console.error)
  // This line of code will only be executed once when your extension
is activated
  console.log('Congratulations, your extension "mscopilot" is now
active!');
```

This function just logs that our extension has been activated. Since the extension is rather invisible to the user (and it should be), it is a good practice to use the log file to store the information that has been instantiated.

Now, we add the code that will get the selected text, instantiate the Parrot model, and add the suggestion to the editor:

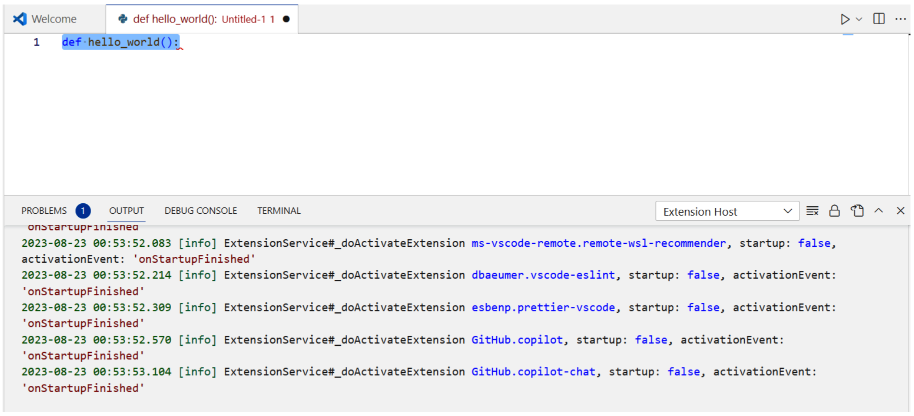
```
// Define a command to check which code is selected.
vscode.commands.registerCommand('mscopilot.logSelectedText', () => {
  // libraries needed to execute python scripts
  const python = require('python-shell');
  const path = require('path');

  // set up the path to the right python interpreter
  // in case we have a virtual environment
  python.PythonShell.defaultOptions = { pythonPath: 'C:/Python311/
python.exe' };
  // Get the active text editor
  const editor = vscode.window.activeTextEditor;

  // Get the selected text
     const selectedText = editor.document.getText(editor.selection);

  // prompt is the same as the selected text
  let prompt:string = selectedText;

  // this is the script in Python that we execute to
  // get the code generated by the Parrot model
  //
  // please note the strange formatting,
  // which is necessary as python is sensitive to indentation
  let scriptText = `
from transformers import pipeline

pipe = pipeline("text-generation", model="codeparrot/codeparrot-
small")
outputs = pipe("${prompt}", max_new_tokens=30, do_sample=False)
print(outputs[0]['generated_text'])`;

  // Let the user know what we start the code generation
  vscode.window.showInformationMessage(`Starting code generation for
prompt: ${prompt}`);

  // run the script and get the message back
  python.PythonShell.runString(scriptText, null).then(messages=>{
  console.log(messages);

  // get the active editor to paste the code there
  let activeEditor = vscode.window.activeTextEditor;

  // paste the generated code snippet
  activeEditor.edit((selectedText) => {

  // when we get the response, we need to format it
  // as one string, not an array of strings
  let snippet = messages.join('\n');

  // and replace the selected text with the output
  selectedText.replace(activeEditor.selection, snippet)  });
  }).then(()=>{
    vscode.window.showInformationMessage(`Code generation
finished!`);});
   });
context.subscriptions.push(disposable);
}
```

This code registers our `'mscopilot.logSelectedText'` command. We made this visible to the extension manager in the previous file – `package.json`. When this command is executed, it performs the following steps. The important part is the interaction between the code in TypeScript and the code in Python. Since we're using the Hugging Face model, the easiest way is to use the same scripts that we've used so far in this book. However, since the extensions are written in TypeScript (or JavaScript), we need to embed the Python code in TypeScript, add a variable to it, and capture the outcome:

1. First, imports the required libraries – `python-shell` and `path` – which are needed to execute Python scripts from within a Node.js environment.
2. Next, it sets up the Python interpreter via `C:/Python311/python.exe`, which will be used to run Python scripts. This is important for ensuring that the correct Python environment is used, even when using a virtual environment. If we do not specify it, we need to find it in the script, which is a bit tricky in the user environment.
3. After, it sets the active text editor and the selected text. We need this selection so that we can send the prompt to the model. In our case, we'll simply send the selection to the model and get the suggested code.
4. Then, it prepares the prompt, which means that it creates a string variable that we use in the code of the Python script.
5. Next, it defines the Python script, where the connection to the ML model is established. Our Python script is defined as a multi-line string (`scriptText`) using a template literal. This script utilizes the Hugging Face Transformers library's `pipeline` function to perform text generation using the `codeparrot-small` model. The Python code is in boldface, and we can see that the string is complemented with the prompt, which is the selected text in the active editor.
6. Then, it displays short information to the user since the model requires some time to load and make an inference. It may take up to a minute (for the first execution) to get the inference as the model needs to be downloaded and set up. Therefore, it is important to display a message that we're starting the inference. A message is displayed to the user using `vscode.window.showInformationMessage`, indicating that the code generation process is about to start.
7. After, it runs the Python script (`scriptText`) using `python.PythonShell.runString`. The script's output is captured in the `messages` array. We lost control over the execution for a while since we waited for the Python script to finish; it provided us with a suggestion for code completion.

8. Next, it pastes the generated code from the response (`messages`) array into a single string (`snippet`). The snippet is then pasted into the active text editor at the position of the selected text, effectively replacing the selected text with the generated code. Since the first element of the response from the model is the prompt, we can simply just replace the selection with the snippet.
9. Finally, it displays the completion message after the code generation process.

Now, we have an extension that we can test. We can execute it by pressing the *F5* key. This brings up a new instance of Visual Studio, where we can type a piece of code to be completed, as shown in *Figure 13.6*:

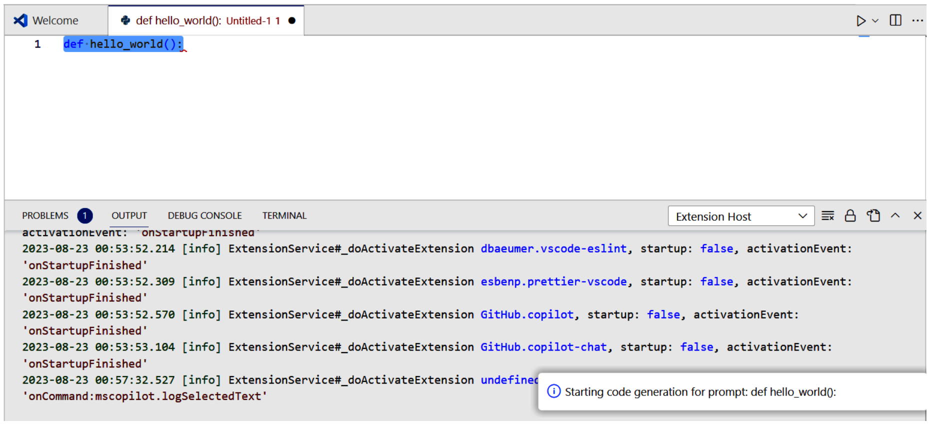


```
def hello_world():
```

```
onStartupFinished
2023-08-23 00:53:52.083 [info] ExtensionService#_doActivateExtension ms-vscode-remote.remote-wsl-recommender, startup: false,
activationEvent: 'onStartupFinished'
2023-08-23 00:53:52.214 [info] ExtensionService#_doActivateExtension dbaeumer.vscode-eslint, startup: false, activationEvent:
'onStartupFinished'
2023-08-23 00:53:52.309 [info] ExtensionService#_doActivateExtension esbenp.prettier-vscode, startup: false, activationEvent:
'onStartupFinished'
2023-08-23 00:53:52.570 [info] ExtensionService#_doActivateExtension GitHub.copilot, startup: false, activationEvent:
'onStartupFinished'
2023-08-23 00:53:53.104 [info] ExtensionService#_doActivateExtension GitHub.copilot-chat, startup: false, activationEvent:
'onStartupFinished'
```

Figure 13.6 – A test instance of some Visual Studio code with our extension activated. The selected text is used as the prompt for the model

Once we press *Ctrl* + *Shift* + *l*, as we defined in the `package.json` file, our command is activated. This is indicated by the messages in the lower right-hand corner of the environment in *Figure 13.7*:



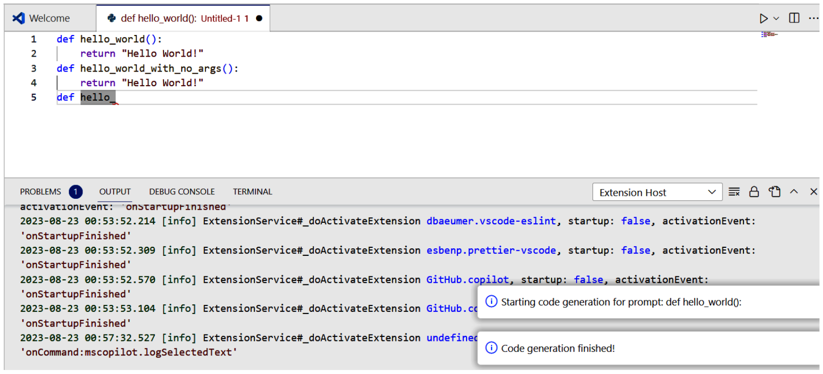
```
def hello_world():
```

```
activationEvent: 'onStartupFinished'
2023-08-23 00:53:52.214 [info] ExtensionService#_doActivateExtension dbaeumer.vscode-eslint, startup: false, activationEvent:
'onStartupFinished'
2023-08-23 00:53:52.309 [info] ExtensionService#_doActivateExtension esbenp.prettier-vscode, startup: false, activationEvent:
'onStartupFinished'
2023-08-23 00:53:52.570 [info] ExtensionService#_doActivateExtension GitHub.copilot, startup: false, activationEvent:
'onStartupFinished'
2023-08-23 00:53:53.104 [info] ExtensionService#_doActivateExtension GitHub.copilot-chat, startup: false, activationEvent:
'onStartupFinished'
2023-08-23 00:57:32.527 [info] ExtensionService#_doActivateExtension undefined
'onCommand:mscopilot.logSelectedText'
```

Starting code generation for prompt: def hello_world():

Figure 13.7 – Starting to generate the code. The message box and the log information indicate that our command is working

After a few seconds, we get a suggestion from the Parrot model, which we must then paste into the editor, as shown in *Figure 13.8*:



```
def hello_world():
    return "Hello World!"
def hello_world_with_no_args():
    return "Hello World!"
def hello_
```

```
activationEvent: 'onStartupFinished'
2023-08-23 00:53:52.214 [info] ExtensionService#_doActivateExtension dbaeumer.vscode-eslint, startup: false, activationEvent:
'onStartupFinished'
2023-08-23 00:53:52.309 [info] ExtensionService#_doActivateExtension esbenp.prettier-vscode, startup: false, activationEvent:
'onStartupFinished'
2023-08-23 00:53:52.570 [info] ExtensionService#_doActivateExtension GitHub.copilot, startup: false, activationEvent:
'onStartupFinished'
2023-08-23 00:53:53.104 [info] ExtensionService#_doActivateExtension GitHub.cc
'onStartupFinished'
2023-08-23 00:57:32.527 [info] ExtensionService#_doActivateExtension undefinec
'onCommand:mscopilot.logSelectedText'
```

Starting code generation for prompt: def hello_world():

Code generation finished!

Figure 13.8 – Code suggestion from the Parrot model pasted into the editor

Here, we can see that our extension is rather simple and gets all the suggestions from the model. So, in addition to the proper code (`return "Hello World!"`), it included more than we needed. We could write more interesting logic, parse the code, and clean it – the sky is the limit. I leave it to you to continue this work and to make it better. My job was to illustrate that writing a GitHub CoPilot-like tool is not as difficult as it may seem.

With this, we've come to my final best practice for this chapter.

Best practice #69

If your model/software aims to help in the daily tasks of your users, make sure that you develop it as an add-in.

Although we could use the Codeparrot model from the Gradio interface, it would not be appreciated. Programmers would have to copy their code to a web browser, click a button, wait for the suggestion, and paste it back into their environment. By providing an extension to Visual Studio Code, we can tap into the workflow of software developers. The only extra task is to select the text to complete and press *Ctrl* + *Shift* + *l*; I'm sure that this could be simplified even more, just like GitHub Copilot does.

Summary

This chapter concludes the third part of this book. It also concludes the most technical part of our journey through the best practices. We've learned how to develop ML systems and how to deploy them. These activities are often called AI engineering, which is the term that places the focus on the development of software systems rather than the models themselves. This term also indicates that testing, deploying, and using ML is much more than training, validating, and testing the models.

Naturally, there is even more to this. Just developing and deploying AI software is not enough. We, as software engineers or AI engineers, need to consider the implications of our actions. Therefore, in the next part of this book, we'll explore the concepts of bias, ethics, and the sustainable use of the fruits of our work – AI software systems.

References

To learn more about the topics that were covered in this chapter, take a look at the following resources:

- *Rana, R., et al. A framework for adoption of machine learning in industry for software defect prediction. In 2014 9th International Conference on Software Engineering and Applications (ICSOFT-EA). 2014. IEEE.*
- *Bosch, J., H.H. Olsson, and I. Crnkovic, Engineering ai systems: A research agenda. Artificial Intelligence Paradigms for Smart Cyber-Physical Systems, 2021: p. 1-19.*
- *Giray, G., A software engineering perspective on engineering machine learning systems: State of the art and challenges. Journal of Systems and Software, 2021. 180: p. 111031.*