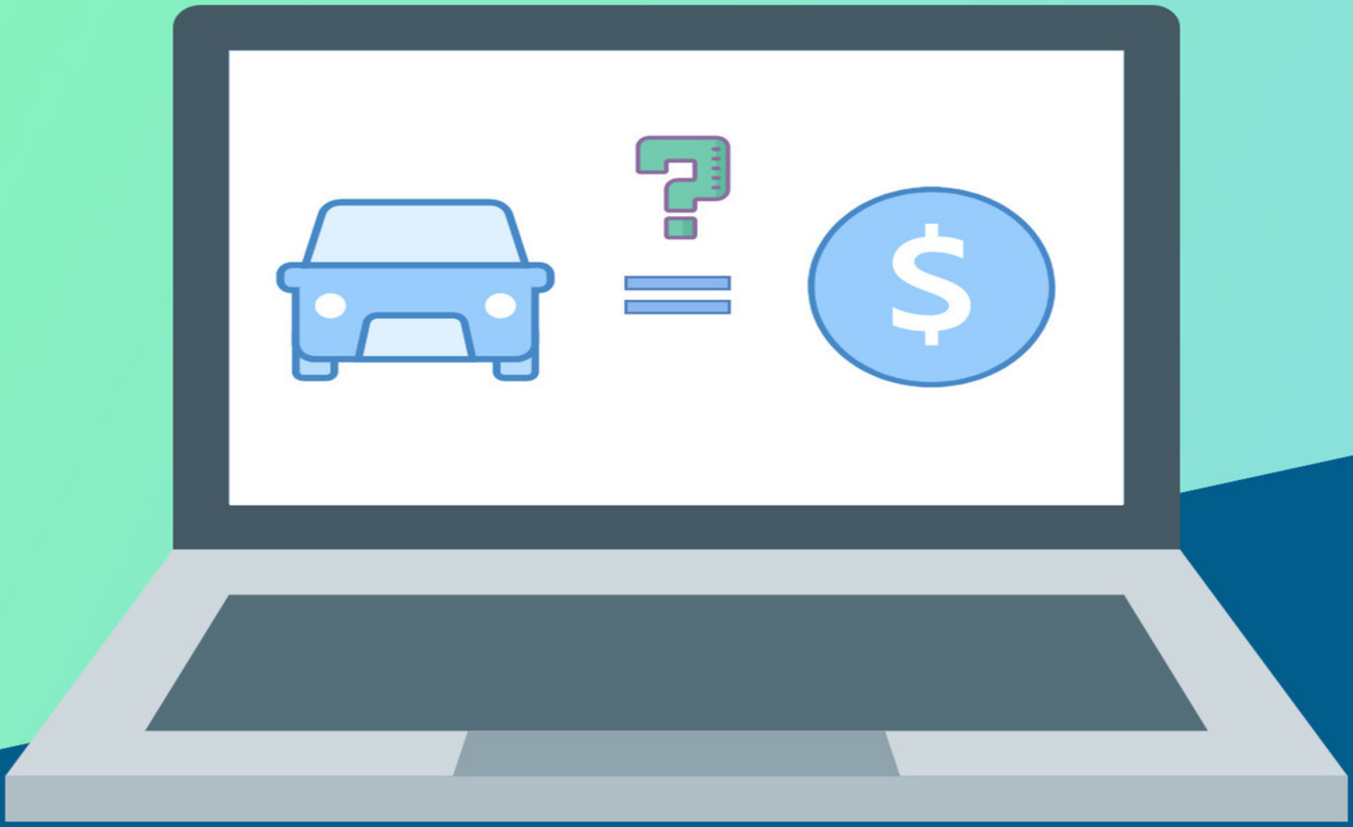


# PYTHON WEBAPP

LEARN HOW TO SERVE A MACHINE LEARNING MODEL PREDICTING CAR PRICES

BY NICOLAS MAIGNAN

# PYTHON WEBAPP

LEARN HOW TO SERVE A
MACHINE LEARNING MODEL
PREDICTING CAR PRICES

BY NICOLAS MAIGNAN

# Python WebApp

Nicolas Maignan

# The power of indecision

## A code book taking you through a small Python Full Stack journey

## 0. Introduction

The goal of this book is to take you through the journey that I, author of this book, followed. This journey, despite being initially guided by my indecision, led to a drastic improvement in my programming skills and an increase in my value as a developer. The point of this book is for you to be able to replicate the process on your own use case, which you are free to define according to your own centers of interest or to your latest concerns. If you go through the following pages, you will learn how to create a Car Price Prediction Web-Application serving the results of a Machine-Learning model using the following frameworks:

- Python:
  - Environment control: Conda, Pip
  - Quality control: Mypy, Flake8, Black
  - Web crawling/scraping: Selenium
  - General data science: Pandas, LightGBM, Scikit-learn
  - Web-Applications: Quart (similar to Sanic, Flask, Django, etc.)
- DevOps:
  - App containerization: Docker
  - Deployment: Docker-compose

In 2018, I was looking to buy a second hand car. My professional life had just started, my student loan was still to be paid back, and I had always heard that buying a brand-new car was the worst possible investment, as the vehicle loses 10 to 30% of its value from the first kilometer traveled. My quest for the best possible deal had already lasted several weeks, and where I started with a precise idea of the car I wanted to acquire, I was later standing completely crushed under my own indecision. Too many brands, too many models, to many prices, and way too many offers. I started to feel a lack of knowledge on my end. So many variables were to be considered that I was never able to asses whether a price was fair or not. The more lost I was, the more obsessed I became with the idea of making the best possible choice.

Luckily enough, I had already been introduced to the wonders of Data Science, Web Scraping, Machine Learning and DevOps. And it is faced with the idea of spending a new weekend of active but fruitless research that I decided to solve my problem in a slightly overkill way. In this book, I won't describe the exact development process I followed. Instead, I'll tell you how things should have been done in the first place to avoid some snags I encountered along the way. On this note, step zero should have been to clearly express the problem I was attempting to solve. From now on, this is the problematic we're are trying to solve:

**How to make sure the price of a second hand car is fair?**

## 1. Working with data

### 1.0 Scoping the need

Answering this question in a structured way requires a lot of data. But after several weeks of intensive research on countless websites, the basic variables needed to describe a car are quite clear.

\`brand\` : the brand of this car \`model\` : the model of this car

\`mileage\` : the distance already traveled by this car

\`model_year\` : the year in which this car was manufactured

\`circulation_date\` : the date at which this car was put to circulation

\`transmission\` : the kind of transmission of this car (automatic or manual)

\`number_of_doors\` : the number of doors of this vehicle `din_horsepower` : the “Deutsche Industrie Normen” horsepower of this car

`tax_horsepower` : the taxable horsepower of this car

`energy` : the type of fuel or energy used by this car

`first_hand` : whether this car is a first-hand one

`imported` : whether this car was imported (to France)

`price` : and of course the exhibition price of this car

And despite a large number of variables set aside, we still end up with the need to retrieve thirteen data fields in a structured way, for several thousand announcements.

Of course such an open dataset, representing the current state of the second-hand car market, doesn’t exist. It has to be created and to do so, we’re going to scrape a website on which those fields are structured enough that they can be easily retrieved. Finding the perfect website doesn’t take long, although many websites still allow people to sell second-hand goods in a completely unstructured way, meaning that the data available from an ad to another are very inconsistent.

## 1.1 Acquiring data

Let’s first define two terms, which are very close to one another:

- **Web scraping** is the fact of automatically downloading web-pages while extracting data from them.
- **Web crawling** is the fact of automatically downloading web-pages while extracting hyperlinks they contain and following those hyperlinks.

While web-scraping and crawling aren’t illegal by themselves, it’s highly probable that using those techniques on websites you don’t own will be illegal. Sadly, the legislation concerning scraping and crawling the web is full of gray areas, and whether the website’s Terms of Use are enforceable, whether you entered a contract with the website or whether fair use can be applied can only be determined case by case.

So before starting any data gathering based on crawling or scraping web-pages, seek legal advice, doubt anything you can read, and cover yourself. I don’t think I will ever stop gathering data this way for very simple reasons:

- I consider this data as being publicly available data.
- Google and all Search Engine Providers have constantly been crawling the web since its start, indeed, web-pages and their content have to be discovered and indexed if results are to be displayed in a search engine (although Google gives webmasters the right to decide whether they want to be indexed on their search engine or not, and most of them want to).

But none of those reasons are legal arguments.

The least that can be done to cover yourself is to use a **Virtual Private Network** with a **Kill-switch** , but I would recommend using **Proxy chains** , picking proxies in several and politically opposed countries, even though this would slow the crawling/scraping process.

Whatever the nature of a project, developing sustainably way means controlling your environment. And by controlling, I don’t mean applying all the possible best practices, but at least a set of rules to follow and which make your code more readable and maintainable over time. If your project becomes something greater than you initially forecasted, you, as well as the other possible contributors, will be grateful that this frame was set early on.

In this case, the programming language we’ll use is Python3, and we’ll use Anaconda to control our python environments. Anaconda is a free and open-source distribution of Python and R. It’s build to simplify package version and environment management.

Let’s build a research python environment which we’ll be using for scraping, cleaning, and modelling:

```
conda create -n research python=3.7.5 conda activate research
```

We'll need several python packages now, including:

- Pandas
- Numpy
- Matplotlib
- Seaborn
- Jupyter
- Selenium
- LightGBM

```
pip install pandas numpy matplotlib seaborn jupyter selenium lightgbm
```

Now that this environment is ready, it is time to create a working directory and to save a static environment file, which allows us to build environments that consistently support our code. To do so, the following command can be run:

```
conda env export > research_env.yml
```

We'll start by creating our crawling/scraping code in a Jupyter notebook which we'll call crawler.ipynb:

```
ProjectDir/ ├── research_env.yml ├── notebooks/ │   ├── crawler.ipynb ├── data/ │   ├── website/
```

The first step to write our crawler script is to map the architecture of the website. In this case, the car ads are listed on pages having the following URLs:

https://www.website.com/ads?page=<integer:1-N>

Going through the HTML code of the /ads?page=<integer:1-N> on a browser developer tools, it is easy to isolate the code related to the listing of the car selling ads. In this case, each page contains 20 HTML div elements of class carAds defined as follows:

```
<html lang= "fr" > > <body class= "pageListing" … > <div class= "carAd" > <div class= "ad"
    <a href= "/car-ad-0000000001.html" class= "adLink" > </div > </div > … </body > </html>
```

Each div of class carAd element has a div of class ad which itself has the following sub-element a of class adLink. This a element, holds an href, having a fixed prefix https://www.website.com/car-ad- and allowing us to build the URL to each of the car ads as follows:

https://www.website.com/car-ad-<integer>.html

So what we want to do is to crawl each listing page and retrieve the href value of each ad. To do so, we'll be using Selenium , running a headless Chrome webdriver . Selenium Webdriver drives a browser natively, Chrome in our case, as a user would. The reason why Selenium Webdriver and its python bindings are often used to crawl websites is that websites can easily protect themselves from simple GET Http requests to their pages, while it is much harder to detect a crawler emulating a user navigating on its browser.

```
from typing import List from dataclasses import dataclass



from selenium.webdriver import Chrome, ChromeOptions  # type: ignore @dataclass class Crawler:
 webdriver_path:     chrome_options:     element_identifer:     listing_url:
str                 str                 str                     str


  def   crawl_listing_page(self   ,   url:   str)   -> ""
List[str]:                                            "

     Gathers the href of all the car ads present on a listing ""
page.                                                          "

      # Create our Chrome WebDriver with the possibility to run
it with

     # or without graphical user interface (headless or   if                self
not)                                                  .chrome_options:

    chrome_options             =       chrome_options.add_argument(self
ChromeOptions()                   .chrome_options)

        driver   =   Chrome(self   .webdriver_path,   options=  else
chrome_options)                                                :

     driver       =       Chrome(self    # Open  the  WebPage  we  want  to   driver.get(ur
.webdriver_path)                      crawl                              l)

    #   Find   the   HTML   elements   we're
interested in

      elems        =        driver.find_elements_by_class_name(self
.element_identifer)

    #  Return  the  href  attribute  of  those
elements

     return  [elem.get_attribute("href"  )  for  elem  in
elems]


   def  crawl_listing_pages(self   ,   start:  int,   end:  int)   ->  ""
List[str]:                                                         "

      Gathers the href of all the car ads on a range of listing ""
pages.                                                           "

   #  Instanciate  our  output  list  of   all_hrefs:  list  =
hrefs                                  []

     # Iterate through the range of listing pages we want to    for  page  in  range(start,
crawl                                                          end):

 try         #  Retrieve  all  the  hrefs  of
:     this page

            page_hrefs = self .crawl_listing_page(self .listing_url
+ str(page))

     #   Add   them   to   the      all_hrefs.extend(page_h    except           pas
output list                  refs)                       Exception :        s

  return
all_hrefs
```

Which we can use as follows:

```
crawler = Crawler( webdriver_path= "./chromedriver" , chrome_options= "--headless" ,
element_identifer= "adLink" , listing_url= "https://www.website.com/ads?page=" , )

ads_urls = crawler.crawl_listing_pages(1 , 2 )
```

This would return a list of car ads URLs:

```
[ 'https://www.website.com/car-ad-9105831530.html' ,
'https://www.website.com/car-ad-7101646302.html' ,
'https://www.website.com/car-ad-6101081682.html' ,
'https://www.website.com/car-ad-9105797990.html' ,
'https://www.website.com/car-ad-7101643178.html' , … ]
```

Now for each of those car ads, we want to retrieve the 13 fields defined earlier. Therefore we explore the ads pages html source code to isolate the html elements of each of those fields.

```
<html lang= "fr" > > <body class= "pageAd" … > <div class= "dataList"
<li class= "carBrand" > Brand here</li> <li class= "carMileage" > 100 000</li> … > </div …
</body > </html>
```

All the necessary fields are present inside a div element of class dataList. The text of each of the sub li elements we're interested in can be easily retrieved using the following script:

```
from typing import List from dataclasses import dataclass

from selenium.webdriver import Chrome, ChromeOptions  # type: ignore

import pandas as pd  # type: ignore @dataclass class Scraper: webdriver_path: str

fields_identifiers: dict   ads_urls: list   results_path: str   batch_size: int = 5

chrome_options: str = "" def scrape_ad_page(self , url: str) -> dict: """

  Extracts the data fields of a car ad. """   if self.chrome_options:

    chrome_options = ChromeOptions()   chrome_options.add_argument(self.chrome_options)
```

```
        driver   =   Chrome(self   .webdriver_path,   options=  else
chrome_options)                                                :


     driver       =       Chrome(self  driver.get(ur    data:  dict  =  {"url"  :
.webdriver_path)                     l)               url}

    for         field         in         self  try
.fields_identifiers.keys():                    :

        element_text                          =        self                    ).te
driver.find_element_by_class_name(                .fields_identifiers[field]      xt

    data[field]         =    except              data[field]   =  return
element_text              Exception :        None               data


  def scrape_ads_pages(self ) -> None  ""      Extracts the data fields of all the car
:                                      "   ads.

    Treats   the   ads   per   batch   of   size
self.batch_size.

    Saves the extracted data in one csv file per ""   ads_batches  =
batch.                                         "   [

     self   .ads_urls[i   :   i   +   self
.batch_size]

       for  i  in  range(len(self  .ads_urls))[::  self
.batch_size]                                             ]

    for        i,        batch        in
enumerate(ads_batches):

         results: List[dict] = [self .scrape_ad_page(page) for page
in batch]

      dataframe:         pd.Dataframe         =
pd.DataFrame(results)

          dataframe.to_csv(self .results_path + f" {i} .csv"  , sep= "," ,
index= False )
```

Which can be used as follows:

```
                    webdriver_path= "./chromedriver"  fields_identifiers=
scraper = Scraper( ,                                 {

  "brand" : "carBrand"    "mileage"              :   }
,                       "carMileage" ,             … ,

  ads_urls=  ads_urls,    #  the  output  of  the   results_path= "../data/website/"
crawler                                          ,

 batch_size=   5   chrome_options=  "--headless"
,               ,                              ) scraper.scrape_ads_pages()
```

Running those two pieces of code allows us to retrieve several hundred thousand car ads and this with high data consistency.

## 1.2 Cleaning

Consistency doesn't mean the data is ready to be used. With a new set of data, an important thing is to make sure we understand what kind of variables we're looking at. Here are the two general types of variables we most commonly observe:

- **Categorical**
  - **Ordinal** : represent discrete and ordered units, example: “poor, fair, good, very good, excellent”, note that the difference between “poor” and “fair”, is not necessarily the same as between “fair”, and “good”, hence why ordinal variables are used to measure non-numeric concepts like feelings.
  - **Nominal** : represent discrete units, example: “Dog, Cat, Bird”
  - **Binary** : represent nominal variables with only two states, examples: “0, 1”, “False, True”, “Yes, No”
- **Numerical**
  - **Discrete** : represent distinct and separate values, which can’t be measured, but can be counted, example: the number of persons in a room (there can’t be, hopefully, half a person in the room)
  - **Continuous** : represent measurements, meaning variables which can take any value within a range, and therefore which can be measured but not counted, example: the weight of a person

Each type of variable comes with its understanding and cleaning challenges.

All the code defined in this section will be stored in a cleaner.ipynb Jupyter notebook as follows:

```
ProjectDir/ ├── notebooks/ │   ├── crawler.ipynb │   ├── cleaner.ipynb ├── data/

│   ├── website/ │   │   ├── 0.csv │   │   ├── 1.csv │   │   ├── ...
```

Let’s load the CSVs with Pandas to take a look at them. Pandas is an open-source library providing simple data structures and analysis tools for Python. A nice alternative to using Pandas is to use Dask , which scales python data science capabilities to clusters with high parallelization and asynchronous programming.

We’ll load some imports:

```
import glob import pandas as pd import re import matplotlib.pyplot as plt import seaborn as sns

from uuid import uuid1
```

And load the files:

```
# Load all our CSVs in one single dataframe

df = pd.concat([pd.read_csv(f) for f in glob.glob("../data/website/*.csv" )], ignore_index= True
)

# Sort the columns alphabetically df = df.reindex(sorted(df.columns), axis= 1 )
```

The df DataFrame would resemble such a table:

| **brand** | **circulation date** | **din_horsepower** | **energy** | **first_hand** | **mileage** | **model** | **model_year** | **number_of_doors** | **imported** | **I** |
|---|---|---|---|---|---|---|---|---|---|---|
| PEUGEOT | 25/02/2015 | 150hp | Diesel | yes | 62 040 km | 508 | 2015 | 5 | yes | 1 € |
| Citroen | 03/01/2013 | 181hp | DIESEL | NaN | 74 084km | DS5 | 2013 | 5 | NaN | 2 € |

From this glimpse at the data, we can immediately spot some issues such as:

- missing values represented by NaN
- numerical variables encoded as strings as 74 084km or 23400 €
- inconsistent formats, see the spaces around units (74 084km versus 62 040 km, words being inconsistently capitalized, etc.)

Classifying the variables encountered will help us decide on strategies to clean our dataset.

Let’s start with our categorical variables. Before we look at any metrics, like the cardinality of our categorical variables, we’ll apply two cleaning steps:

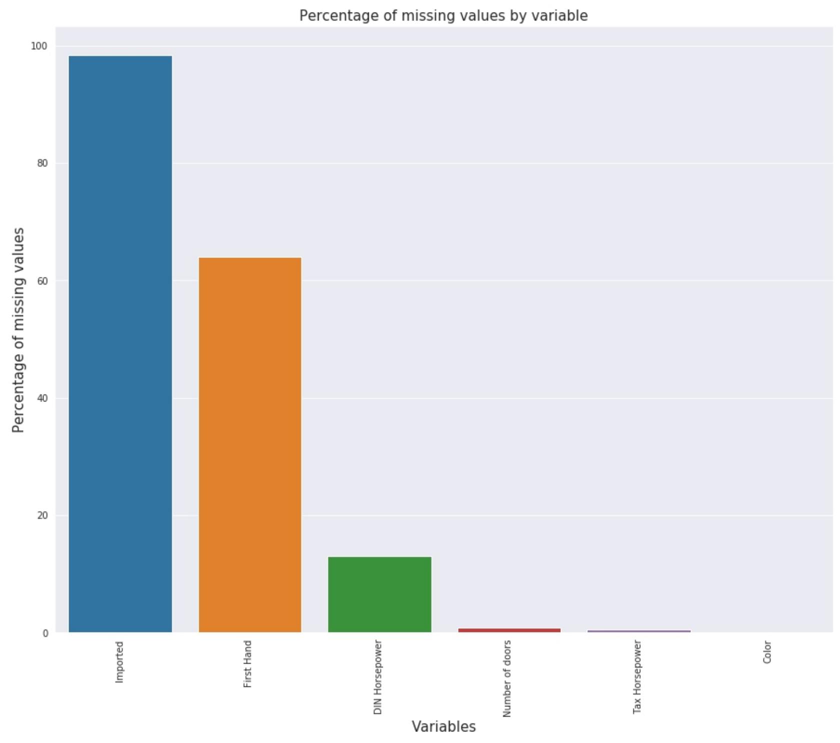
- lower all the string variables, this way two different strings like PEUGEOT and Peugeot become the same again.
- strip all the string variables, so we remove useless spacings and can again recover from peugeot to peugeot

```
# Treat categorical variables categorical_variables = [ "brand" , "model" , "finish"
, "transmission" , "energy" , "first_hand" , "imported" ] for column in categorical_variables:

  # Overwrite the Series with its lowered version df[column] = df[column].str.lower()

  # Overwrite the Series with its stripped version df[column] = df[column].str.strip()
```

Then we’ll look at the missing values of our dataframe using Pandas, Matplotlib and Seaborn:

```
df_na = (df.isnull().sum() / len(df)) * 100

df_na = df_na.drop(df_na[df_na == 0 ].index).sort_values(ascending= False )[:30 ]

f, ax = plt.subplots(figsize= (15 , 12 )) plt.xticks(rotation= '90' )

sns.barplot(x= df_na.index, y= df_na) plt.xlabel('Variables' , fontsize= 15 )

plt.ylabel('Percentage of missing values' , fontsize= 15 )

plt.title('Percentage of missing values by variable' , fontsize= 15 )
```

Which outputs the following graph:



The cardinality, or number of unique values each categorical value can take, is easily retrieved using either:

```
len(df["Column" ].unique())
```

Or directly using Pandas describe function which returns a table containing statistical insights (count, unique, top, etc.).

```
df.describe(include = "all" )
```

At this point we have enough information to determine the type of categorical variable we're dealing with:

| Variable | Cardinality | Data Type | Example |
|---|---|---|---|
| brand | 117 | nominal | ["peugeot", "citroen", …] |
| model | 1126 | nominal | ["508", "ds5", …] |
| transmission | 2 | binary | "mecanical" or "automatic" |
| energy | 8 | nominal | ["diesel", "gasoline", …] |
| first_hand | 1 | binary | "yes" or NaN |
| imported | 1 | binary | "yes" or NaN |

Binary variables can now be processed. The transmission variable can be simplified as a boolean called automatic, while first_hand and imported can also be turned into boolean, True for yes and False otherwise. To do so we'll just use direct string comparison. Doing so we'll also solve our issue of missing values for the first_hand and imported variables.

```
# Create the Automatic column and drop the original

df["automatic" ] = df["transmission" ] == "automatic"

df.drop(["transmission" ], axis= 1 , inplace= True )

# Turn the First Hand and Imported columns into booleans

df["first_hand" ] = df["first_hand" ] == "yes" df["imported" ] = df["imported" ] == "yes"
```

Concerning nominal variables, an easy way to simplify the problem is to reduce the number of categories we want to deal with. In this particular case, selecting a subset of brands of interest will immediately reduce the cardinality of the brand and model variables. Here we'll select a small subset of brands that seem more relevant to my situation. Modeling would also be improved considering how different the price of a regular car evolves compared to the price of a premium car.

```
# Filter on brands brands_of_interest = [ "audi" , "bmw" , "citroen" , "ds" , "ford" ,
"mercedes" , "nissan" , "opel" , "peugeot" , "renault" , "seat" , "toyota" ]

df = df[df["brand" ].isin(brands_of_interest)]
```

Here are the updated cardinalities on our dataframe, which has lost less than 20% of its size, meaning that a tenth of the brands represents 80% of the second-hand car selling ads on the crawled website:

| Variable | Cardinality |
|---|---|
| brand | 12 |
| model | 455 |

| Variable | Cardinality |
|---|---|
| automatic | 2 |
| energy | 8 |
| first_hand | 2 |
| imported | 2 |

Now that we don’t have columns containing large amounts of NaN values, we can use a simple Pandas function to remove rows with missing fields. Potentially, if this operation was to remove too many rows from a dataset, which is not the case here, we could implement strategies to replace missing values with placeholders determined using statistics (for example replacing missing horsepowers with the average observed horsepower for the considered car model).

```
# Removing rows with missing values df.dropna(inplace= True )
```

Next are the numerical variables, and those come with several issues:

- model_year and number_of_doors are correctly encoded as integers
- mileage, din_horsepower, tax_horsepower and price are encoded as strings with inconsistent formats
- circulation_date has to be turned into a datetime objects

We’ll be using some regex for the first problem, and Pandas to_datetime function for the second one:

```
# Treat string like numerical variables

str_numerical_variables = ["mileage" , "din_horsepower" , "tax_horsepower" , "price" ]

                                          # Use regex to extract integer values from the
for column in str_numerical_variables: strings

 df[column]  =      float(""   .join(re.findall(r"\d+"   ,   str(s))))   for   s   in
[            df[column].tolist()                                                  ]
```

```
# Format Circulation date

df["circulation_date" ] = pd.to_datetime(df["circulation_date" ], infer_datetime_format= True )
```

One additional operation we’ll conduct is removing ads for car models that don’t appear more than a thousand times in our set. This is to make sure we have enough data points on each car model to understand how its price evolves.

```
thresh = 1000 df = df.groupby("model" ).filter(lambda x: len(x) >= thresh)
```

The cleaning being done, we can finally look at some statistics and determine the type of data:

| Variable | Mean | Std | Min | Max | Data Type |
|---|---|---|---|---|---|
| mileage | 61738.48 | 60870.53 | 1.00 | 709293.00 | continuous |
| number_of_doors | 4.68 | 0.75 | 2.00 | 6.00 | ordinal |
| din_horsepower | 134.91 | 60.97 | 5.00 | 670.00 | continuous |
| tax_horsepower | 7.35 | 4.59 | 1.00 | 57.00 | continuous |
| price | 22599.99 | 16064.15 | 100.00 | 434.00 | continuous |
| model_year | 2014.68 | 4.03 | 1924.00 | 2019.00 | continuous |

We can save the data in our data folder:

```
# Create a uuid for each ad to identify them

df.to_csv("../data/cleaned_data.csv" , index= False )
```

## 2. Modeling

In order to answer the question stated previously:

**How to make sure the price of a second-hand car is fair?**

We could take several approaches such as:

- building a classifier, which goal would be to output a boolean representing whether or not a price is fair.
- building a regressor, which goal would be to infer the exact market price of a vehicle.

The notion of fairness is tricky though. Indeed, nothing tells us that any of the ads from which we gathered data comes with a fair price.

In the scenario of a classifier for example, we'd need to build a training target telling us whether the car price is fair or not. And the only way to do so would be to set subjective ranges around prices observed for similar vehicles, which considering the number of variables to take into account, wouldn't be possible.

Therefore we'll go for the regressor approach and we'll let the fairness evaluation to the human being. Indeed, in this case, the model will output a price representative of what it has observed in the dataset, and whether the price is fair or not depends on how fair all the ads were on the website we crawled.

So the entire exercise is based on a strong assumption that a majority of the car ads are fair, which we can easily accept considering the popularity of the website, and the simple fact that most of those car ads turn into deals.

Another question we want to ask ourselves is:

Shall we go for one single model capable of predicting the price of any car model within our dataset, or shall we cut the problem down?

The only way to answer this question was to try both approaches. From my tests, I decided to go for a general regressor rather than car-model-wise regressors. Despite how different prices behave depending on the brand or even on the model, modern Machine Learning algorithms are capable of generalizing a price behavior and to understand those differences. So provided the correct features, a model can understand that the price of a Mercedes doesn't decrease as much with the mileage as the price of a Renault. Keeping one model also simplifies the deployment stack, as we'll need to load one single model whatever the prediction we want to make.

### 2.0 Feature engineering

Feature engineering is the keystone of Machine Learning.

First, it translates the data into understandable inputs for our models. Secondly, it also allows data scientists to simplify how the model understands the problem. For example, if we were to predict the biological gender of an animal using its height, providing the model with a feature built from a comparison with the average height of a given gender can be more insightful than directly providing the model with the height (is_taller_than_average_female: True or False).

For this part, we'll be working in a new jupyter notebook called modeling.ipynb for visualizations and in a new app folder for production code. Please note that we're foreseeing production code at modeling time, as it is very easy to create a model that can be very hard to put in production.

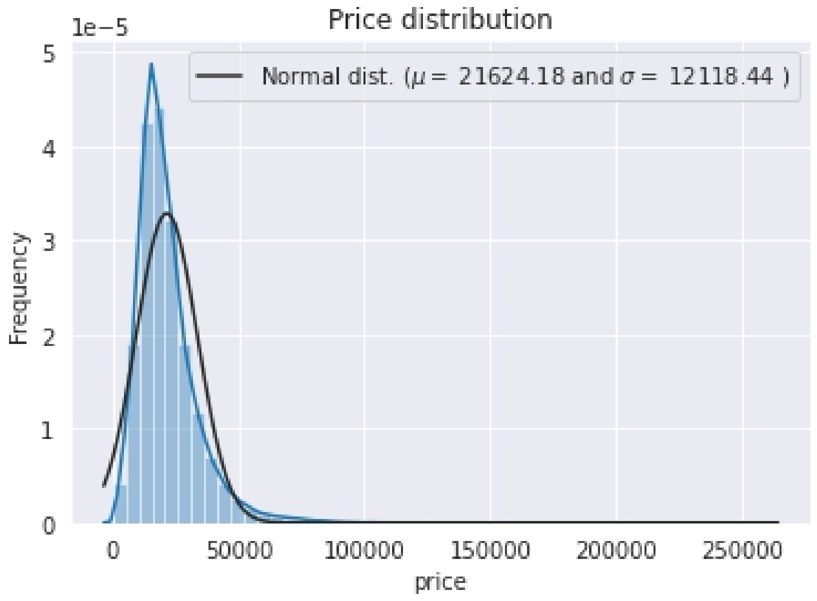
```
ProjectDir/ ├── notebooks/ │   ├── modeling.ipynb ├── app/ │   ├── data/
│   │   ├── cleaned_data.csv │   ├── utils/ │   │   ├── __init__.py │   │   ├── preprocessor.py
│   │   ├── pipeline.py │   ├── __init__.py │   ├── __main__.py
```

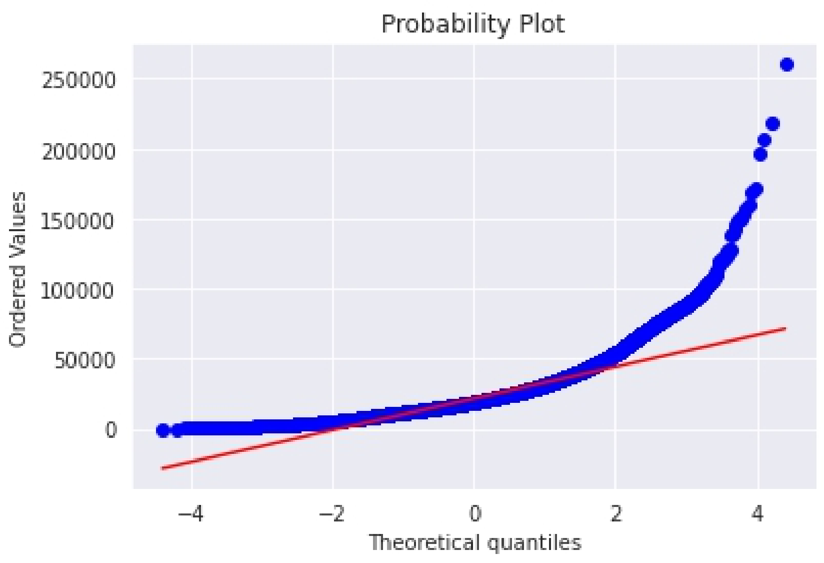
**Target engineering**

I like to start my feature engineering work by preparing the target of my problems. In this particular case, the target of the regression is of course the price column of our Pandas dataframe.

My first reflex is to look at the distribution of the data. We can do so using Seaborn in our modeling notebook:

```
import pandas as pd from scipy import stats import seaborn as sns

import matplotlib.pyplot as plt color = sns.color_palette() sns.set_style('darkgrid' )


df = pd.read_csv("../data/cleaned_data.csv" , parse_dates= ["circulation_date" ])


# Plot our Price data sns.distplot(df["price" ], fit= norm); # Fit a normal distribution to it


(mu, sigma) = norm.fit(df["price" ]) # Plot the distribution

plt.legend(['Normal dist. ($\mu=$ {:.2f} and $\sigma=$ {:.2f} )' .format(mu, sigma)],

  loc= 'best'
)             plt.ylabel('Frequency' ) plt.title('Price distribution' ) # Create a QQ-plot

fig = plt.figure() res = stats.probplot(df["price" ], plot= plt) plt.show()
```



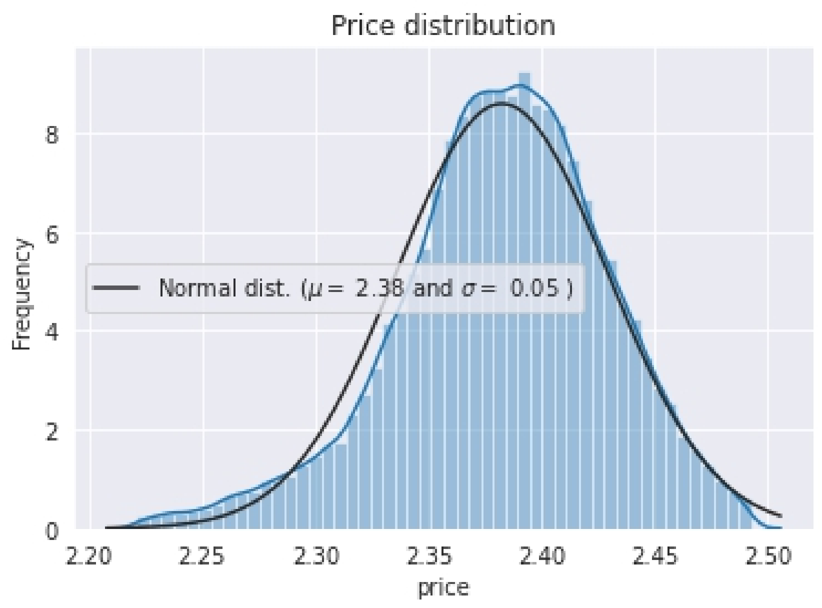


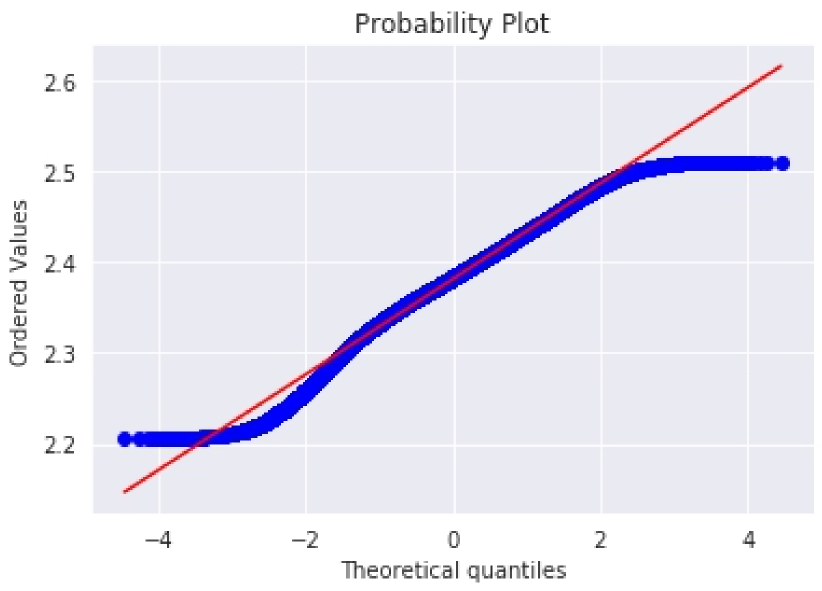
A first observation reveals that our data has outliers. This means data points extremely far from our normal distribution.

We can deal with those either by capping the data or by removing them. Here we'll choose to remove the data, as we aren't interested in predicting the values of super ars, collection cars, or simple ruins. We'll do so using filtering based on percentile.

A second observation, which the QQ plot allows us to make, is that our data is skewed. We, therefore, apply a logarithm transformation to the price column to make it more normally distributed, which will make any model more robust as it normalizes the magnitude differences.

```
# We apply log1p to our Price column df["price" ] = np.log1p(df["price" ])


# Plot the above graphs again sns.distplot(df["price" ] , fit= norm);

(mu, sigma) = norm.fit(df["price" ])

plt.legend(['Normal dist. ($\mu=$ {:.2f} and $\sigma=$ {:.2f} )' .format(mu, sigma)],

  loc= 'best'
)              plt.ylabel('Frequency' ) plt.title('Price distribution' ) fig = plt.figure()

res = stats.probplot(df["price" ], plot= plt) plt.show()
```





We can see that we've improved the distribution of our data by making it more approximate to normal. The QQ plot also shows a better linear behavior, even though we still observe skewness at both extremes.

To tackle those two issues, we'll create a function in our script app/utils/preprocessor.py:

```
from datetime import datetime import pandas as pd import numpy as np


                                                                ""
def process_target(df: pd.DataFrame) -> pd.DataFrame: "

  Removes outliers and applies log transform to the  ""
target.                                                  "

  upper_lim = df["price" ].quantile(0.99   lower_lim = df["price" ].quantile(0.01
)                                         )
```

```
   df  =  df[(df["price"  ]  <  upper_lim)  &  (df["price"  ]  >
lower_lim)]

  df["price"  ]  =  np.log1p(df["price"  return
])                                    df
```

**Creating some features**

We'll create some basic features from the dates variable, the model age, the horsepowers, etc. Those features will help the model understand the meaning of the dates related to a vehicle, and put this in relationship with how much the car traveled. To do so, we'll add a function to the app/utils/preprocessor.py script:

```
                                                        ""   Creates          ""
def create_features(df: pd.DataFrame) -> pd.DataFrame: "   features.         "

  #  Number  of  days  since  circulation
date

   df["circulation_days"  ]  =  (datetime.now()  -  df["circulation_date"
]).dt.days

  #  Number  of  years  since  model   df["model_age" ] = 2019 - df["model_year"
creation                           ]

  # Average milesage the car drove per
day

   df["average_mileage_per_day"    ]    =    df["mileage"    ]    /
df["circulation_days" ]

   #  Tax  horse  per  Din  horse,  "How  much  bang  for  the
bucks"

   df["tax_/_din"    ]    =    df["tax_horsepower"    ]    /  return
df["din_horsepower" ]                                       df
```

**Creating a Model pipeline**

The next step is to create a pipeline allowing us to:

- create reversible feature transforms so that we can translate features for the model
- generate and train a model
- make price predictions

Talking about a model, it's time to choose one, and for our problem, the best solution is certainly a Gradient Boosting Model. We have to decide which API we go for and the three main choices we have are:

- XGBoost (Mar 2014)
- LightGBM (Jan 2017)
- CatBoost (Apr 2017)

Those three solutions are extremely similar and easily inter-changeable. If you're interested in their differences, you'll find a lot of comparisons online. In this book, we'll be using LightGBM.

Then we still need to answer two questions:

**First, how do we transform categorical features for our model?**

For categorical features, we can use a processing step called One-Hot encoding, which replaces a categorical feature with as many binary features as it has labels. So a feature like brand="renault" or "bmw" turns into two features renault=True or False and bmw=True or False. I recommend that you also investigate Label (or Ordinal) encoding, which simply means converting string labels to integer values. Generally speaking, One-hot encoding is a better representation of categorical features, but you need to beware for feature explosion if the cardinality of your features is high, in which case you might prefer simpler Label encoding.

**Second, how do we transform numerical features for our model?**

For numerical features, generally used solutions are MinMaxScaler, RobustScaler, StandardScaler, and many more. We'll go for the classic StandardScaler which transforms our feature distribution so that it's centered on 0 and has a standard deviation of 1. Therefore 68% of the values will lie between -1 and 1. The reason why we apply such transform is simply that machine learning algorithms perform better and/or converge faster when features are on a relatively similar scale and/or close to normally distributed.

Let's create a function in the script api/utils/pipeline.py to create this pipeline. We'll make use of the famous Scikit-learn Pipelines to do so:

```
from typing import List import pandas as pd from category_encoders import OneHotEncoder

from lightgbm import LGBMRegressor from sklearn.compose import ColumnTransformer

from sklearn.pipeline import Pipeline, make_pipeline

from sklearn.preprocessing import StandardScaler


                                                    ""
def create_pipeline(df: pd.DataFrame) -> Pipeline: "

   Creates  a  sklearn  pipeline  which  takes  care  of  feature  model     training     and
transforms,                                                         inference.

""   #   Define   the   target  target_column:    str    =  #   Define   the   categorical
"   column                     "price"                      columns

 categorical_cols: List[str]  =  "brand"   "energy"   "first_hand"   "imported"   "model"
[                                ,         ,          ,              ,            ,

  "number_of_doors"  "automatic    #   Define   the   numerical  numerical_cols: List[str]  =
,                    "          ] columns                      [

  "din_horsepower"  "mileage"   "tax_horsepower"   "circulation_days"  "model_age"
,                   ,         ,                    ,                   ,

   "average_mileage_per_day  "tax_/_din    #     Generate     a
" ,                        "          ] Pipeline

  regressor:       Pipeline       =    #    Create    a    Column   ColumnTransforme
make_pipeline(                        Transformer                 r(                [

        ("num"      ,      StandardScaler(),         ("cat"       ,       OneHotEncoder(),    )
numerical_cols),                            categorical_cols),                             ] ,

  #    Create    a    LGBM  LGBMRegresso     boosting_type=          objective=
Regressor                r(              "gbdt" ,                  "regression" ,

          metric= ["l2_root" , "l1" ],   # Root Mean Squared Error and Mean
Absolute Error

  num_leaves=       learning_rate=       feature_fraction=       bagging_fraction=
512 ,              0.1 ,                0.7 ,                   0.7 ,

  reg_alpha= 0    reg_lambda=      verbose= 0 )      # Fit  the  Column  Transformer  and  the
,              0.1 ,              ,             ,  ) model

   regressor.fit(df[categorical_cols        +         numerical_cols],  return
df[target_column])                                                     regressor
```

**Parameters Optimization**

If you wonder how those parameters for our LGBM model were defined, I found them using a method called GridSearchCV. This method makes an exhaustive set of combinations of parameters from a given parameter space. It then fits an LGBM Regressor on k random folds of divided data, before scoring the model on the remaining fold. The combination of parameters giving the best score is kept. This method is compute-intensive, so I recommend that you also investigate RandomSearchCV and Bayesian search. Here’s a snippet of code to apply Grid search cross-validation:

```
import lightgbm as lgb from sklearn.model_selection import GridSearchCV params = {

  "boosting" : "gbdt" ,   # gdbt, rf, goss or  "objective" : "regression"
dart                                          ,

    "metrics" : ["l2_root" , "l1" ],   # metric(s) to be evaluated on the evaluation
set(s)

    "num_threads" : 2 ,   # number of threads for LightGBM (set to available real CPU
cores)

   "num_leaves" : 128 ,   # max number of leaves in one
tree

   "max_depth" : -1 ,   # limit the max depth for tree
model

  "learning_rate" : 0.1 ,   # shrinkage
rate

    "feature_fraction" : 1 ,   #  if you set it to 0.8, LightGBM will select 80% of features
before training each tree

    "bagging_fraction" : 1 ,   # like feature_fraction, but this will randomly select part of
data without resampling

  "lambda_l1"   :   0.0   ,      #   L1   "lambda_l2"   :   0.0   ,      #   L2
regularization                            regularization                         }

                                                      boosting=  params["boosting"
# Create a regressor mdl = lgb.LGBMRegressor( ],

 objective=  params["objective"   num_threads=  params["num_threads"
],                                ],

 num_leaves=  params["num_leaves"  max_depth=  params["max_depth"
],                                  ],

  learning_rate=  params["learning_rate"   feature_fraction=  params["feature_fraction"
],                                        ],

  bagging_fraction=  params["bagging_fraction"  lambda_l1=  params["lambda_l1"
],                                              ],

 lambda_l2=  params["lambda_l2"
],                                ) # Create parameters space to search gridParams = {

 "boosting_type"  :  ["gbdt"  "objective"  :  ["regression"  "random_state"  :  [501
],                            ],                            ],

  "num_leaves" : [128 , 256 , 512   "learning_rate" : [0.005 , 0.05 , 0.1
],                                  ],

  "feature_fraction" : [0.7 , 0.8 , 0.9   "bagging_fraction" : [0.7 , 0.8 , 0.9
],                                        ],

  "lambda_l1" : [0 , 0.1 , 0.5  "lambda_l2" : [0 , 0.1 , 0.5
],                              ],                              } # Create the grid
```

```
grid = GridSearchCV(mdl, gridParams, verbose= 1 , cv= 4 , n_jobs= 2 )

# Run the grid search of our training dataset

# (X is our features dataframe after Column transforms as above, # Y is the target price)

grid.fit(X, Y) # Print the best parameters found print(grid.best_params_)

print(grid.best_score_)
```

**Model Evaluation**

Let’s look at what kind of errors our model does. In our case, looking at the Root Mean Squared Error (RMSE or L2-root) or the Mean Absolute Error (L1) we obtain on a testing-set at the end of training won’t help us figure out how well our model performs. The reason is that we have applied Logarithm transform to our target price, meaning that the metrics will represent errors on the logarithm scale. So what we can do is:

- Make a train/test split
- Train on training-set
- Predict on testing-set
- Apply exponential to the prediction on the test set and the target in the test set
- Recompute an RMSE between the predictions and the actual prices observed

Let’s do it:

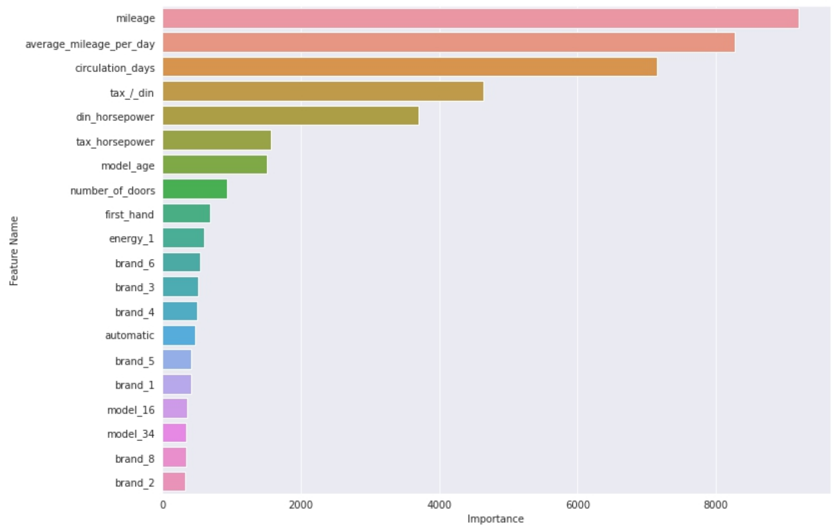
```
import numpy as np from sklearn.metrics import mean_squared_error

from sklearn.model_selection import train_test_split

# Make our random train test split (80% / 20%)

train, test = train_test_split(df, test_size= 0.2 )

# Create our regressor using the utils function we've made pipeline = create_pipeline(train)

# Predict on the test set log_predictions = pipeline.predict(test)

# Turn predictions back to prices predictions = np.expm1(log_predictions) # Compute target

target = np.expm1(test["price" ].tolist()) # Compute RMSE

np.sqrt(mean_squared_error(exp_target, exp_pred))
```

This calculation tells us that we have a Root Mean Squared Error of 1250 EUR. Not too bad for a simple model, particularly knowing that one price differentiator for cars is the set of options and that we don’t have this data in our dataframe. For example the GPS option immediately increases the price of a car by around 500 EUR. Neither do we know whether the car had an accident, whether the previous owner was smoking inside, etc.

We can look at brand or even model levels to have a better view of our model performance. For one of the most frequent models of our dataset (the Renault Clio), we have an RMSE of 545 EUR. Simple pandas selection allows us to do so, for example: test[test["brand"] == "renault"]

One more thing we can do to understand a bit more what the drivers of price are is to look at feature importance. LightGBM provides a feature_importance method returning scores of how much each feature impacts the price. By default this importance score is based on the number of decision tree splits done based on each feature. Let’s plot this:

```
import seaborn as sns import matplotlib.pyplot as plt color = sns.color_palette()


sns.set_style('darkgrid' ) # Get importance scores from the LGBMRegressor

importance = regressor.steps[1 ][1 ].feature_importances_ # Retrieve feature names

features_list = numerical_cols + list(

    regressor.named_steps['columntransformer'                    ].named_transformers_['cat'
].get_feature_names()                                                                          )

                                                        pd.Series(features_lis
# Create aa dataframe f_importance = pd.concat( [ t),

  pd.Series(importanc ]   axis=
e)                     ,   1       ) # Set column names

f_importance.columns = ['Feature Name' , 'Importance' ] # Sort by importance

metrics_sorted = f_importance.sort_values('Importance' , ascending= False ) #Plot

g = sns.barplot(x= "Importance" , y= "Feature Name" , data= metrics_sorted[:20 ])

g.figure.set_size_inches(12 , 9 ) plt.show()
```



## 3. Serving our model

In the previous chapter, we’ve built utils functions that create a Scikit-learn pipeline, allowing us to take raw incoming data, and to train a LightGBM Regressor on it before using it for actual inference on raw data. Now we’re going to create a Web Application around those utils function.

We’ll go for a simple single-page WebApp, on which users will be able to fill in a form describing a vehicle, submit it and get an estimation of the value of the vehicle.

### 3.0 WebApp creation

When building a WebApp with Python, the first step is selecting a framework. Classic frameworks for Python WebApps are Flask, Django, Sanic, Quart, Tornado, Pyramid, and some more. I recommend you to be fluent in some of those solutions. Knowing one simplifies the discovery and learning of another one. In this book, we'll be using Quart. The reason is that Quart mimics another very popular framework,Flask, with a major difference being default ASGI (Asynchronous Server Gateway Interface) support as opposed to default WSGI (Web Server Gateway Interface) for Flask. Asynchronous programming, introduced to Python with the asyncio library, and the async/await syntax from Python3.5, allow us to write concurrent code which translates in increased performance for networking, database connectors, web-servers, task-queues, etc.

Let's recall the project's structure:

```
ProjectDir/ ├── notebooks/ │   ├── modelling.ipynb ├── app/ │   ├── data/

│   │   ├── cleaned_data.csv │   ├── utils/ │   │   ├── __init__.py │   │   ├── preprocessor.py

│   │   ├── pipeline.py │   ├── __init__.py │   ├── __main__.py
```

Let's create our application. In the file called app/__main__.py:

```
# Prepare imports from datetime import datetime import numpy as np import pandas as pd


from quart import Quart, jsonify, render_template, request # Import our utils

from utils.pipeline import create_pipeline



from utils.preprocessor import create_features, process_target # Load our dataframe

df = pd.read_csv("data/cleaned_data.csv" , parse_dates= ["circulation_date" ])

# Process the target df = process_target(df) # Create features df = create_features(df)


# Create model pipeline regressor = create_pipeline(df) # Create a Quart app


app = Quart(__name__ ) # Create a GET route to render our single webpage

                                                                      ""
@app.route ("/" , methods= ["GET" ]) async def render_index(): "

  Renders   the   index.html   template ""    return await render_template("index.html"
file.                                    "    )


# Create a GET route to return models @app.route ("/models/" , methods= ["GET" ])

                         ""    Returns  a  json  holding  a  dictionary  binding  models  to
async def get_models(): "   brands.

   This is used to allow users of the app to select models   our  model  supports  for  each
that                                                        brand.

""  models    =  for          brand          in
"   {}          df.brand.unique():

      models[brand]      =     ","     .join(sorted(df[df.brand     ==  return
brand].model.unique()))                                           jsonify(models)


# Create a POST route to make price predictions @app.route ("/" , methods= ["POST" ])

                              ""    Retrieves  a  data  form,  processes  the  data
async def make_prediction(): "   fields,
  and  passes  them  through  our  price  prediction ""  #     Retrieve
pipeline.                                             "   form

 form        =       await   #  Process  fields  which  types  need  to  be
request.form              changed

   form["circulation_date" ] = datetime.strptime(form["circulation_date" ], "%Y-%m-
%d" )

  form["model_year" ] = int(form["model_year"   form["mileage" ] = int(form["mileage"
])                                            ])

  form["din_horsepower" ] = int(form["din_horsepower"
])

  form["tax_horsepower" ] = int(form["tax_horsepower"
])

  form["number_of_doors" ] = int(form["number_of_doors"
])

   form["imported" ] = True if form["imported" ] == "True" else
False

   form["automatic" ] = True if form["automatic" ] == "True" else
False

   form["first_hand" ] = True if form["first_hand" ] == "True" else
False

  #  Create  prediction  row  with  row = pd.DataFrame(form, index= [0
features                             ])

 row                    =   # Predict  and  turn  log  prediction  back  into
create_features(row)       price

  prediction  =  int(np.expm1(regressor.predict(row)[0  #         Return
]))                                                   response

  return        jsonify({"price"        :
str(prediction)})                         # Run our application if __name__ == "__main__" :

  app.run(host= "0.0.0.0" , port= 5000 , debug=
True )
```

What we just did is:

- Prepare a regressor object using our dataframe and utils functions
- Create a Quart application object
- Create a first GET Route rendering our single-webpage, which will be made available at the root endpoint /, which means, in case our application is deployed on the following domain website.com, that the webapage will be accessible there http://website.com/
- Create a second GET Route at /models returning a dictionary of the brands and corresponding car models at each Http GET request at http://website.com/models/
- Create a POST Route at / so that each Http POST request made at http://website.com/ can send data to our price prediction model and gets a price prediction back
- Run the app and make it available at the machine’s local address 0.0.0.0:5000

Now the index.html file has to be created. We’ll use some very basic HTML and JavaScript to do so. We’ll create the file in an app/templates folder, as Quart will automatically look for templates in this folder.

```
ProjectDir/ ├── notebooks/ │   ├── modelling.ipynb ├── app/ │   ├── data/

│   │   ├── cleaned_data.csv │   ├── utils/ │   │   ├── __init__.py │   │   ├── preprocessor.py
```

```
|   |   ├── pipeline.py |   ├── templates/ |   |   ├── index.html |   ├── __init__.py
|   ├── __main__.py
```

Let’s start with the basic structure of our file, and particularly the head element:

```
<!DOCTYPE html> <html lang= "en" > <head> <title> App</title> <meta charset= "UTF-8" >

   <meta  name=  "viewport"  content=  "width=device-width,  initial-scale=1" >

    <link href= "{{ url_for('static', filename='css/style.css') }} " rel= "stylesheet" type= "text/css" />

    <script  src=  "https://ajax.googleapis.com/ajax/libs/jquery/2.1.3/jquery.min.js"  > </script>

    <link  href=  "https://cdn.jsdelivr.net/npm/select2@4.0.13/dist/css/select2.min.css"  rel= "stylesheet" />

    <script  src=  "https://cdn.jsdelivr.net/npm/select2@4.0.13/dist/js/select2.min.js"  > </script>

    <script  src=  "https://cdn.jsdelivr.net/npm/sweetalert2@9/dist/sweetalert2.min.js"  > </script>

    <link  href=  "https://cdn.jsdelivr.net/npm/sweetalert2@9/dist/sweetalert2.min.css"  id= "theme-styles" rel= "stylesheet" >

</head>
```

In the head, we’re defining some standard meta tags which are usually here to specify a character set, description, keywords, author, and viewport settings. We then define some link tags which are here to specify paths to some internal and external CSS stylesheets. And we define some script tags which specify the sources of some JavaScript packages that we will be using, namely sweetalert2 and select2, respectively used to create good looking alerts and dropdown lists.

Now for the body of our HTML, we will encapsulate a simple form element in a wrapper div, itself in a container div for later styling purposes.

```
<body> <div class= "container" > <div class= "wrapper" >

        <form id= "form_pred" class= "form validate-form" method= "post" > ... </form > </div > </div >

</body>
```

Within this form element, we’ll define one input per raw feature expected by our model as follows:

```
     <span  class=  "form-title" >  Welcome  to  your  Car  Price  Predictor!  </span>


       <div  class=  "form-input  input-select" >  <span  class=  "label-input"  > Brand</span>  <div>

               <select  class=  "selection"  id=  "brand" name= "brand" >
               <option value= "none" > Choose a            <option  value=  "audi"  >
brand </option>                                   Audi </option>

           <option  value=  "bmw"  >               <option  value=  "citroen"  >
Bmw </option>                          Citroen </option>

            <option  value=  "ford"  >               <option  value=  "mercedes"  >
Ford </option>                           Mercedes </option>

             <option  value=  "nissan"  >             <option  value=  "opel"  >
Nissan </option>                           Opel </option>

              <option  value=  "peugeot"  >              <option  value=  "renault"  >
Peugeot </option>                            Renault </option>

             <option  value=  "toyota"  >              <option  value=  "volkswagen"  >
Toyota </option>                           Volkswagen </option>

  </sele   </di         <span  class=  "focus-input"  </di
ct>       v>     ></span>                            v>


       <div  class=  "form-input  input-          <span  class=  "label-input"  >  <di
select" >                                Model</span>                            v>

               <select class= "selection" id=  "model"            <option>   Choose   above
name= "model" >                                      first</option>

  </sele   </di         <span  class=  "focus-input"  </di
ct>       v>     ></span>                            v>


               <div  class=  "form-input  validate-input"  data-validate=  "Circulation
date required." >

           <span class= "label-input"  > Circulation
date</span>

                     <input   class=   "input"   type=   "date"   id=   "circulation_date"   name=
"circulation_date" placeholder= "31/01/2009" >

        <span  class=  "focus-input"  </di
></span>                             v>


              <div  class=  "form-input  validate-input"  data-validate=  "Model
year required." >

          <span  class=  "label-input"  >  Model
Year</span>

                     <input class= "input" type= "number" step= "1" min= "1900" id= "model_year"
name= "model_year" placeholder= "2009" >

        <span  class=  "focus-input"  </di
></span>                             v>


              <div   class=   "form-input   validate-input"   data-validate=
"Mileage required." >

         <span   class=   "label-input"   >
Mileage</span>

                     <input class= "input" type= "number" step= "1" min= "0" max= "1000000" id=
"mileage" name= "mileage" placeholder= "145364" >
```

```
        <span  class=  "focus-input"  </di
></span>                             v>


               <div   class=   "form-input   validate-input"   data-validate=   "DIN
horsepower required." >

           <span   class=   "label-input"   >   DIN
horsepower</span>

                      <input class= "input" type= "number" step= "1" min= "0" max= "1000" id=
"din_horsepower" name= "din_horsepower" placeholder= "130" >

        <span  class=  "focus-input"  </di
></span>                             v>


               <div   class=   "form-input   validate-input"   data-validate=   "Tax
horsepower required." >

           <span   class=   "label-input"   >   Tax
horsepower</span>

                      <input  class=  "input"  type=  "number"  step=  "1"  min=  "0"  max=  "30"  id=
"tax_horsepower" name= "tax_horsepower" placeholder= "7" >


        <span  class=  "focus-input"  </di         <div  class=  "form-input  input-
></span>                             v>     select" >

          <span   class=   "label-input"   >  <di
Énergie</span>                               v>

               <select  class=  "selection"  id=  "energy"
name= "energy" >

                 <option value= "diesel" selected= "" >
Diesel </option>

              <option   value=   "essence"   >                <option   value=   "electrique"   >
Gasoline </option>                               Electric </option>

                          <option  value=  "hybride  essence  électrique"  >  Hybrid
Gasoline Electric </option>

                         <option  value=  "hybride  diesel  électrique"  >  Hybrid
Diesel Electric </option>

                      <option  value=  "bicarburation  essence  gpl"  >
Gasoline GPL </option>

                          <option   value=   "bicarburation   essence   bioéthanol"   >
Gasoline bioethanol </option>

                      <option  value=  "bicarburation  essence  gnv"  >   </sele   </di
Gasoline GNV </option>                                               ct>       v>


        <span  class=  "focus-input"  </di         <div  class=  "form-input  input-
></span>                             v>     select" >

          <span   class=   "label-input"   >  <di
Imported</span>                               v>

                <select  class=  "selection"  id=  "imported"
name= "imported" >
```

```
           <option  value=  "True"  >               <option  value=  "False"  selected=
Yes </option>                          "" > No </option>

  </sele   </di         <span  class=  "focus-input"  </di
ct>       v>     ></span>                           v>


      <div  class=  "form-input  input-           <span   class=   "label-input"   >  <di
select" >                              Automatic</span>                            v>

                <select  class=  "selection"  id=  "automatic"
name= "automatic" >

           <option  value=  "True"  >               <option  value=  "False"  selected=
Yes </option>                          "" > No </option>

  </sele   </di         <span  class=  "focus-input"  </di
ct>       v>     ></span>                           v>


      <div  class=  "form-input  input-           <span  class=  "label-input"  >  First  <di
select" >                              hand</span>                                  v>

                <select  class=  "selection"  id=  "first_hand"
name= "first_hand" >

           <option  value=  "True"  >               <option  value=  "False"  selected=
Yes </option>                          "" > No </option>

  </sele   </di         <span  class=  "focus-input"  </di
ct>       v>     ></span>                           v>


              <div  class=  "form-input  validate-input"  data-validate=  "Number  of
doors required." >

           <span  class=  "label-input"  >  Number  of
doors</span>

                     <input   class=   "input"   type=   "text"   id=   "number_of_doors"   name=
"number_of_doors" placeholder= "5" >


       <span  class=  "focus-input"  </di        <div class= "container-form-
></span>                           v>     btn" >

      <div    class=    "wrap-          <div    class=    "form-
form-btn" >                bgbtn" ></div>

           <button  id=  "predict"  class=   <spa     Evalua    </sp    </butt   </di   </di
"form-btn" >                          n>     te !       an>     on>        v>     v>
```

At the end of our body, we now have to define some scripts for the dropdown lists to make use of select2:

```
 <div   id=   "dropDownSelect1"   >  <script    $  (".selection"  ).select2
</div>                             >         ({

    minimumResultsForSearch  :      dropdownParent         :         $  }    </script
20 ,                            ('#dropDownSelect1' )                   );   >
```

We also need a script to get the car models from the GET /models route we’ve created:

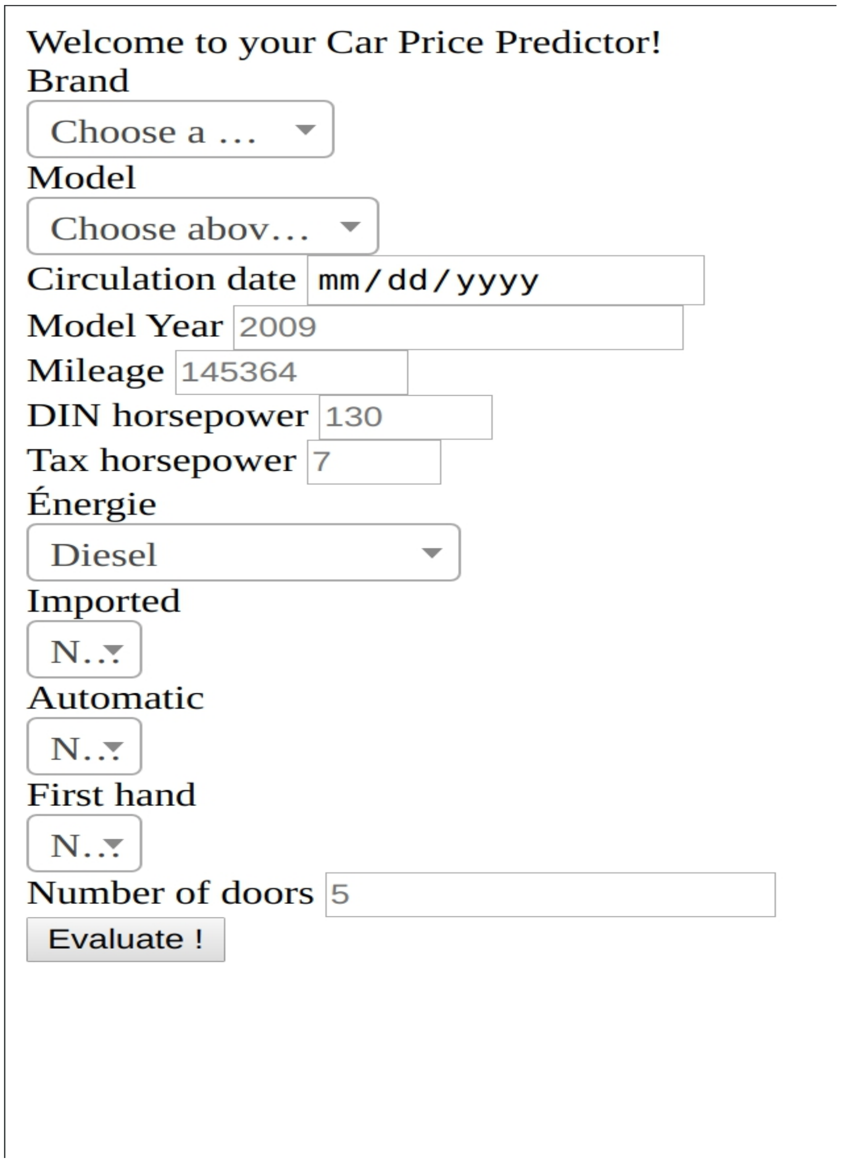
```
<script> $ ("#brand" ).change (function () {
var $dropdown = $ (this );

$ .getJSON ("/models" , function (data) {
var key = $dropdown .val ();

var vals = data[key].split (",");
var $secondChoice = $ ("#model" );
$secondChoice .empty ();

$ .each (vals, function (index, value) {

$secondChoice .append ("<option>" + value + "</option>" );
} );
} );
} );
</script>
```

And finally we need some script to actually call our POST / route when the user pushes the form button to evaluate the price of a car. We’ll make this call using ajax:

```
<script type= "text/javascript" > $ (function () {
$ ("button#predict" ).click (function (e){

e .preventDefault ();
/*Get variables from form*/
var brand = $ ("#brand" ).val ();

var model = $ ("#model" ).val ();
var circulation_date = $ ("#circulation_date" ).val ();

var model_year = $ ("#model_year" ).val ();
var mileage = $ ("#mileage" ).val ();

var din_horsepower = $ ("#din_horsepower" ).val ();

var tax_horsepower = $ ("#tax_horsepower" ).val ();

var energy = $ ("#energy" ).val (), ext_color = $ ("#ext_color" ).val ();

var imported = $ ("#imported" ).val ();
var automatic = $ ("#automatic" ).val ();

var first_hand = $ ("#first_hand" ).val ();

var number_of_doors = $ ("#number_of_doors" ).val ();
/*Create the JSON payload*/
var data = {

"brand" : brand, "model" : model, "circulation_date" : circulation_date,

"model_year" : model_year, "mileage" : mileage, "din_horsepower" : din_horsepower,

"tax_horsepower" : tax_horsepower, "energy" : energy, "imported" : imported,

"automatic" : automatic, "first_hand" : first_hand, "number_of_doors" : number_of_doors,
};
```

```
          /*Send the ajax request and fire a sweetalert   $   .ajax    method       :
on success*/                                           ({         "POST" ,

      url  : window  .location    data     :        success    :   function
.href ,                        data,        (result){

      var  json_result  =           var         price         =
result;                    json_result['price' ];

                Swal  .fire  ('Estimated Price: '  + price+  ' EUR'  }       error          :
, '' , 'success' )                                                  ,   function  (){

      console      .log     }    }    }     </script
("error" )               } )    )    );    >
```

The HTML of our webpage is now ready. All we need to do is style it a little as it currently looks like that:


Welcome to your Car Price Predictor!

| | |
|---|---|
| Brand | Choose a … |
| Model | Choose abov… |
| Circulation date | mm/dd/yyyy |
| Model Year | 2009 |
| Mileage | 145364 |
| DIN horsepower | 130 |
| Tax horsepower | 7 |
| Énergie | Diesel |
| Imported | N… |
| Automatic | N… |
| First hand | N… |
| Number of doors | 5 |

So let's create a new file in app/static/css/ called style.css, which you can see we import in the head of our HTML. Quart will automatically look for stylesheets in this folder. Let's start by importing a font and restyling the basic tags:

```
@import                                                 url('https://fonts.googleapis.com/css2?
family=Roboto+Condensed:wght@300;400;700&display=swap' );

    margin  :  0px  padding  :  0px  box-sizing  : border-box                 height  :  100%
* { ;              ;                ;                         } body, html { ;

  font-family  :  "Roboto  Condensed"  ,  sans-         font-family : "Roboto Condensed"
serif ;                                          } a { ;

 font-size  :  14px  line-height  :  1.7  color  :  #666666  margin  :  0px  transition  : all  0.4s
;                  ;                   ;                 ;              ;

 -webkit-transition  : all  0.4s  -o-transition  : all  0.4s  -moz-transition  : all  0.4s
;                               ;                          ;                             }

           outline  :  none  !important              text-decoration  :  none
a:focus { ;                                    } a:hover { ;                          } h1, h2, h3, h4, h5,

     margin  :  0px         font-family  : "Roboto Condensed"  font-size  :  14px
h6 { ;               } p { ;                                  ;

 line-height  :  1.7  color  :  #666666  margin  :  0px            margin  :  0px
;                   ;                 ;              } ul, li { ;

 list-style-type  :  none            outline  :  none  border  :  none
;                        } input { ;                 ;               } textarea {

 outline  :  none  border  :  none
;                ;               } textarea:focus , input:focus {

 border-color  :  transparent !important
;                                       } input:focus:: -webkit-input-placeholder {

 color  :  transparent                                  color  :  transparent
;                     } input:focus: -moz-placeholder { ;                     }

                                  color  :  transparent
input:focus:: -moz-placeholder { ;                     } input:focus: -ms-input-placeholder {

 color  :  transparent                                               color  :  transparent
;                     } textarea:focus:: -webkit-input-placeholder { ;                     }

                                    color  :  transparent
textarea:focus: -moz-placeholder { ;                     } textarea:focus:: -moz-placeholder {

 color  :  transparent                                          color  :  transparent
;                     } textarea:focus: -ms-input-placeholder { ;                     }

                                    color  :  #adadad
input:: -webkit-input-placeholder { ;                 } input: -moz-placeholder {

 color  :  #adadad                         color  :  #adadad
;                 } input:: -moz-placeholder { ;                 }

                                color  :  #adadad
input: -ms-input-placeholder { ;                 } textarea:: -webkit-input-placeholder {

 color  :  #adadad                           color  :  #adadad
;                 } textarea: -moz-placeholder { ;                 }

                              color  :  #adadad
textarea:: -moz-placeholder { ;                 } textarea: -ms-input-placeholder {

 color  :  #adadad              outline  : none  !important  border  :  none
;                 } button { ;                           ;
```

```
 background : transparent                   cursor : pointer
;                          } button:hover { ;                  } iframe {

 border : none !important
;                          }
```

Now we can restyle our main container div:

```
             width : 100% min-height : 100vh display :  -webkit- display  :  -webkit-
.container { ;            ;                    box;                   flex;

 display  :  -moz-  display   :   -ms-  display     : flex-wrap     :
box;                flexbox;              flex;          wrap;

 justify-content : center  align-items : center padding : 15px  background : #a64bf4
;                          ;                      ;                 ;

   background : -webkit-linear-gradient(45deg , #03ce9b , #10abd1
);

   background : -o-linear-gradient(45deg , #03ce9b , #10abd1
);

   background : -moz-linear-gradient(45deg , #03ce9b , #10abd1
);

  background : linear-gradient(45deg , #03ce9b , #10abd1
);                                                         }
```

Our wrapper div:

```
           width : 500px background : #fff  border-radius : 10px overflow : hidden
.wrapper { ;               ;                   ;                      ;

 padding : 42px 55px 45px 55px
;                               }
```

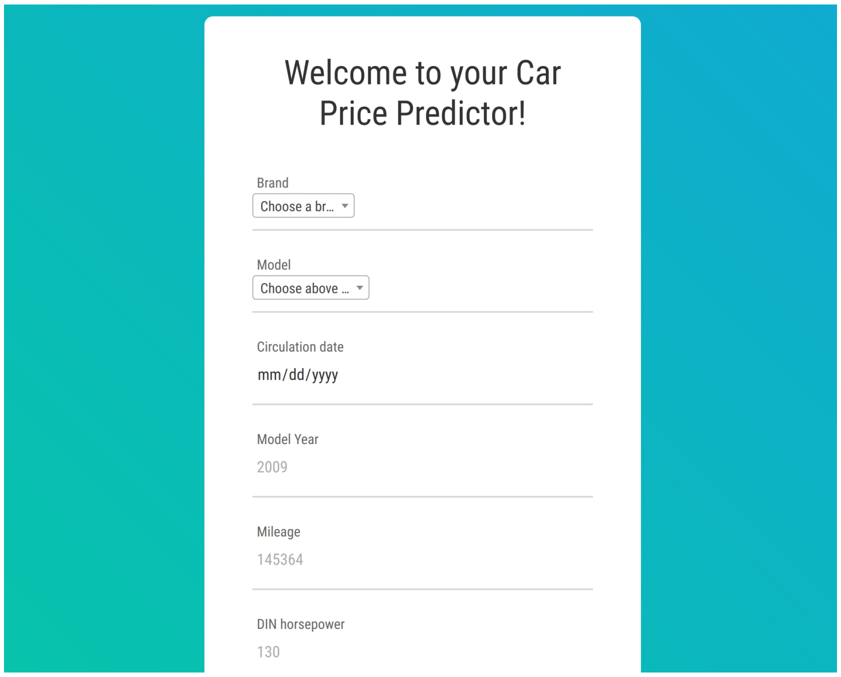
Our form element, its inputs and button:

```
        width : 100%                 display : block  font-family : "Roboto Condensed"
.form { ;               } .form-title { ;                  ;

 font-size : 39px  color : #333333 line-height : 1.2  text-align : center
;                  ;                 ;                   ;

 padding-bottom : 44px                 width : 100% position : relative
;                       } .form-input { ;              ;

 border-bottom : 2px solid #d9d9d9 padding-bottom : 13px margin-bottom : 27px
;                                  ;                       ;                      }

                 font-family : "Roboto Condensed" font-size : 16px color : #666666
.label-input { ;                                  ;                   ;

 line-height : 1.5 padding-left : 5px             display : block width : 100%
;                  ;                    } .input { ;                 ;

 background : transparent  font-family : "Roboto Condensed" font-size : 18px
;                          ;                                  ;

 color : #333333 line-height : 1.2 padding : 0 5px                  position : absolute
;                ;                   ;                  } .focus-input { ;
```

```
 display  :  block  width  :  100%  height  :   100% top  :   0 left   :  0  pointer-events  :  none
;                 ;               ;               ;         ;           ;                       }

                         content  :  "" display   :  block  position  :  absolute  bottom  :  -2px
.focus-input ::before { ;               ;                 ;                      ;


 left  :  0  width  :  0  height  :  2px  -webkit-transition  : all 0.4s  -o-transition  : all  0.4s
;          ;            ;              ;                                ;


 -moz-transition  : all 0.4s   transition  : all 0.4s  background  : #7f7f7f
;                             ;                       ;                       } input.input {

 height  : 40px                     min-height   : 110px  padding-top  : 9px
;               } textarea.input { ;                     ;

 padding-bottom  : 13px                                            width  : 100%
;                        } .input :focus +.focus-input ::before { ;               }

                                         width  :  100%
.has-val.input +.focus-input ::before { ;               } .container-form-btn {

 display   :  -webkit-  display    :   -webkit-  display   :  -moz-  display     :     -ms-
box;                    flex;                   box;                flexbox;

 display       : flex-wrap       :  justify-content  : center  padding-top  : 13px
flex;           wrap;              ;                          ;                     }

                  width  : 100%  display  :  block  position  : relative  z-index  :  1
.wrap-form-btn { ;               ;                 ;                      ;

 border-radius  : 25px  overflow  :  hidden margin  :  0 auto
;                       ;                   ;                 } .form-bgbtn {

 position  : absolute  z-index  :  -1 width  :  300% height   : 100%  background  :  #a64bf4
;                      ;              ;              ;               ;

    background  :  -webkit-linear-gradient(left  ,  #03ce9b  ,  #10abd1  ,  #03ce9b  ,
#10abd1 );

   background  :  -o-linear-gradient(left  ,  #03ce9b  ,  #10abd1  ,  #03ce9b  ,
#10abd1 );

   background  :  -moz-linear-gradient(left  ,  #03ce9b  ,  #10abd1  ,  #03ce9b  ,
#10abd1 );

   background  :  linear-gradient(left  ,  #03ce9b  ,  #10abd1  ,  #03ce9b  ,  top  :  0
#10abd1 );                                                                     ;


 left  : -100%   -webkit-transition  : all 0.4s  -o-transition  : all 0.4s
;               ;                               ;

 -moz-transition  : all 0.4s   transition  : all 0.4s                display   :   -webkit-
;                             ;                       } .form-btn { box;

 display    :  -webkit-  display    :  -moz-  display     :    -ms-  display       :
flex;                   box;                 flexbox;                flex;

 justify-content  : center  align-items   : center  padding  : 0 20px  width  : 100%
;                           ;                      ;                   ;


 height  : 50px   font-family  : "Roboto Condensed"  font-size  : 16px  color  :  #fff
;                ;                                 ;                 ;
```

```
 line-height  :  1.2                                        left  :  0
;                    } .wrap-form-btn :hover .form-bgbtn { ;           } .form-btn i {

 -webkit-transition : all 0.4s  -o-transition : all 0.4s  -moz-transition : all 0.4s
;                                ;                           ;

 transition : all 0.4s                         -webkit-transform  : translateX(10px
;                        } .form-btn :hover i { );

 -moz-transform  : translateX(10px  -ms-transform  : translateX(10px
);                                    );

 -o-transform:  translateX(10px  transform  : translateX(10px
);                                 );                         }
```

And now our webpage looks much better!


Welcome to your Car Price Predictor!

Brand
Choose a br...

Model
Choose above ...

Circulation date
mm/dd/yyyy

Model Year
2009

Mileage
145364

DIN horsepower
130

### 3.1 Containerization

From now on, we want to deploy this application. My favorite way of doing this is using Docker. Docker is your best friend. Docker is based on so-called Containers which are standardized units of software allowing developers to isolate applications from their environments. To put it simply, by containerizing your application, create a standard piece of software that can run on any machine equipped with Docker. This means no more cases where your code runs on a given machine but not on another one. Docker is compatible with most of the Operating Systems out there, meaning that it simplifies passing from an OS to another. You can now proudly program on Windows and deploy on Ubuntu server without a headache (but I still recommend you to go for Linux).

So how does docker create a container? Docker uses so-called images, which you can see as a snapshot of your application's software and its required environment. So let's create a Dockerfile next to our app folder:

```
ProjectDir/ ├── notebooks/ │   ├── modeling.ipynb ├── app/ │   ├── data/

│   │   ├── cleaned_data.csv │   ├── utils/ │   │   ├── __init__.py │   │   ├── preprocessor.py
│   │   ├── pipeline.py │   ├── static/ │   │   ├── css/ │   │   │   ├── style.css
│   ├── templates/ │   │   ├── index.html │   ├── __init__.py │   ├── __main__.py ├── Dockerfile
```

In our Dockerfile we will:

- specify an image, meaning the operating system in which we'll run our application. In our case, we can go for miniconda3

```
FROM continuumio/miniconda3 as base
RUN apt-get update && apt install -y build-essential
```

- create a python environment and install the required packages

```
RUN conda create -n app python
RUN export PIP_CACHE_DIR="/opt/cache/pip"
RUN echo "source activate app" > ~/.bashrc
ENV PATH /opt/conda/envs/app/bin:$PATH
RUN pip install \
        'quart==0.11.5' \
        'pandas==1.0.1' \
        'numpy==1.18.1' \
        'lightgbm==2.3.1'
```

- create a production image of our app using just the minimally required python environment:

```
# PRODUCTION IMAGE
FROM continuumio/miniconda3 as prod
COPY --from=base /opt/conda/envs /opt/conda/envs
RUN echo "source activate app" > ~/.bashrc
ENV PATH /opt/conda/envs/app/bin:$PATH
COPY . .
WORKDIR /app
```

- create a test image of our app which will run flake8 linting of our code during the image creation process:

```
# TESTING IMAGE
FROM continuumio/miniconda3 as test
COPY --from=base /opt/conda/envs /opt/conda/envs
RUN echo "source activate app" > ~/.bashrc
ENV PATH /opt/conda/envs/app/bin:$PATH
RUN pip install \
        'flake8==3.8.1'
COPY . .
WORKDIR /app
RUN flake8
```

Talking about flake8 our python linter, let's define a small .flake8 file in our app folder and in which we specify some code standards like the maximum line length:

```
[flake8]
max-line-length = 88
```

Once done, we can use the Docker CLI to build our test image by running:

```
docker build --target test .
```

This build command point at the Dockerfile and aims particularly at the test image. Docker will download create the base miniconda3 image and its python environment, before switching to the test image by copying the base one and installing and running flake8 also. We could do the same with Pytest, which means adding it to the python environment before running Pytest during the image creation process. Doing so, any failing test would cancel the image creation, which is a very valuable feature to have in a Continuous Integration / Continuous Deployment pipeline.

## 3.2 Deployment

Now that we have a ready Dockerfile, we will use an even better tool called Docker Compose. Docker Compose is a tool used to define and run multi-container Docker applications. Modern applications tend to be more and more microservice-based. We can now build simple YAML files to define and run all the services of an application. Let's create a docker-compose.dev.yml file next to our existing Dockerfile. In this file, we will define two services, namely:

- our price prediction web app
- a reverse proxy, which we place in front of our web app to deal with incoming requests, and therefore acting as an intermediary between the web and our web app. We'll use Traefik, a docker compliant Edge router to do so.

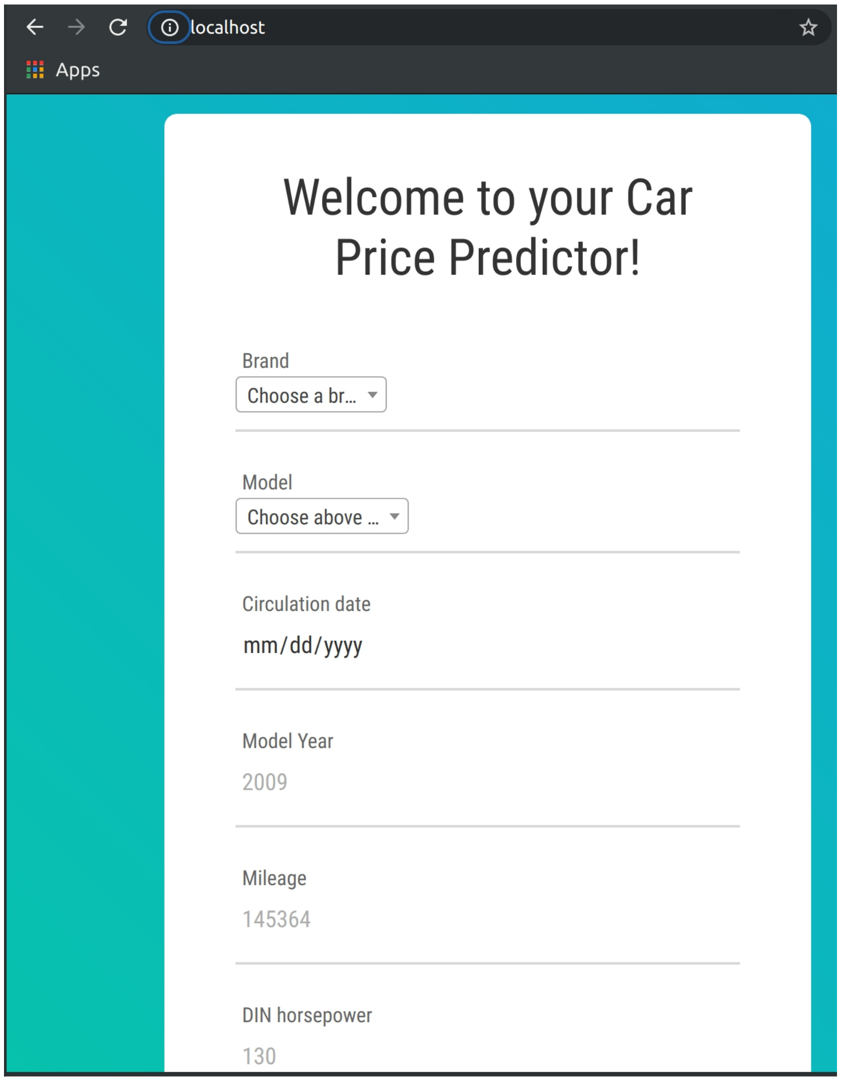
Here's our docker-compose.dev.yml file:

```
                          traefik   image:               container_name:
version: "3.5" services: :        "traefik:v2.2"       "traefik"

 restart:        command    -                    "--
always           :        providers.docker=true"

   -                                      "--    -                         "-- ports   -
providers.docker.exposedbydefault=false"      entrypoints.web.address=:80"      :      "80:80"

 volumes     -                                            webapp  build  context:   target:
:        "/var/run/docker.sock:/var/run/docker.sock:ro"   :       :      .          prod

 container_name:          restart:        labels    -
"webapp"                 always          :       "traefik.enable=true"

    -
"traefik.http.routers.pwrouter.rule=Host(`localhost`)"

   -                                                ports   -      command:           python3
"traefik.http.routers.pwrouter.entrypoints=web"    :      5000   __main__.py
```

Let's run this configuration using the Docker Compose CLI:

```
docker-compose -f docker-compose.dev.yml up --build
```

Docker will then automatically create a local docker network in which all our services will be able to interact. Then it will download the specified Traefik image and boot a container with the specified commands, which are standard configurations to expose our docker application on port 80 to the rest of the world. Once done, it will build the prod image of our web app, boot a container of this image, set some labels for Traefik, and launch the application by running the command python3 __main__.py before exposing the port 5000 (see __main__.py). Once done, if you open your browser and go to localhost, you should see the web app up and running:



Welcome to your Car Price Predictor!

Brand
Choose a br...

Model
Choose above ...

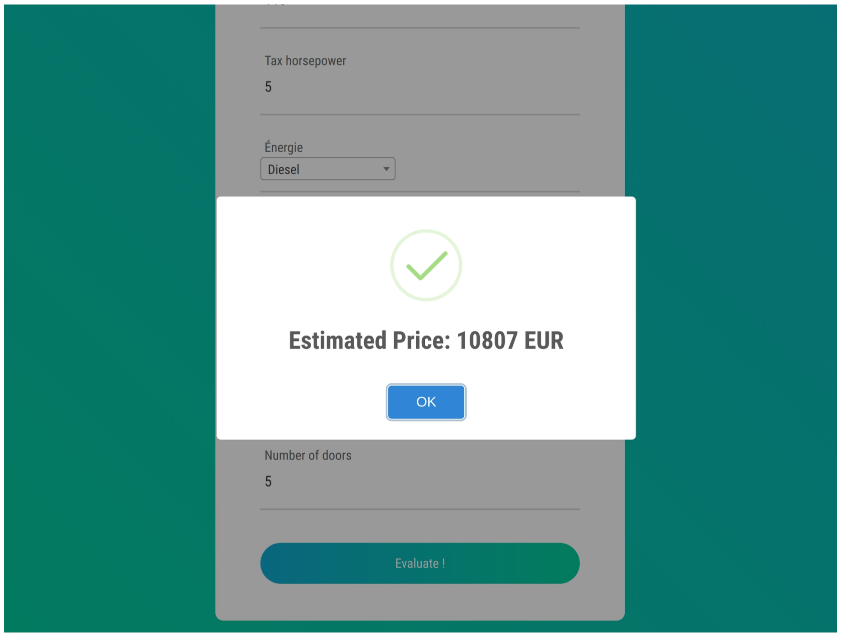
Circulation date
mm/dd/yyyy

Model Year
2009

Mileage
145364

DIN horsepower
130

Let's make a price prediction for a random ad we've found:



Congrats, your web app is running, and it serves the results of a Machine Learning model!

## 4. Conclusion

After deploying locally, we are just one step away from deploying to the web. All it would take is just a domain name and a virtual machine. SSH in the machine, and run a similar docker-compose file with your domain URL as Host instead of localhost. Some more complexities might occur with getting Https certificates. But we'll cover more complex deployments, using Kubernetes, in another ebook.

I hope this book convinced you of the power of mastering Data Science, Machine Learning, and DevOps with Python. Whatever the questions you want to answer, there is a way to answer them and to share your insights using a similar process as the one we went through.