Contents

1 Quick Info

Audience: This guide is made for beginners with basic knowledge in Python programming.

Goal: Predict house sale prices in a Kaggle beginner competition (predicting house sale price) using machine learning modules in Python.

Resources: For this guide you can download working code files, and other files.

Kaggle: House Prices - Advanced Regression Techniques

Python Script (Main): KagglC1.py

Python Script (Supplement) KagglDataC1.py

Last Edit: 2025 April 03

Credits: This guide is inspired by chapter 2 in "*Hands on Machine Learning*" by Aurélien Geron [HoML]. I am in no way associated with the author himself. This guide does not replicate any parts of the book, and the code presented here is based on publicly available source codes (see Colab).

2 Introduction

I want to use this introduction briefly to explain how to *learn* the basics of machine learning, because it can be quite intimidating for newcomers with little background knowledge. Even without much knowledge about Python, you can learn language on the fly by following this guide, but if you want more preparations, then you should get familiar with the most basic concepts (**variables, lists, tuples, dictionaries, functions, loops, if-else-statements**). You will also encounter other concepts like **lambda functions** or **classes**, but our use cases are rather simple.

You will probably find out that learning Python modules for machine learning or data scientists almost feels like learning a new language, anyway.

For now you do not need much mathematical background except very simple **school mathematics**. Of course more advanced topics require more knowledge (like basic linear algebra or probability theory), but as long as you do not intend to build your own machine learning tools, you can simply use the existing ones without knowing every mathematical or technical detail working under the hood.

The best way to get a grasp on machine learning is to start with very practical books like "***Hands on Machine Learning***" by Aurélien Geron, because they explain working source codes for real world examples. The alternative would be starting from scratch with very basic books, but you may not have time to learn every detail right from the beginning.

Of course practical books can have a very steep learning curve, but if you use learning techniques like **priming**, **incubation**, and the **24-hour rule** combined with practical coding you can get started with machine learning within just a few days or weeks. This means that you do should not try to memorize everything from the beginning, but rather skim through the working examples, and revisit the details later on, while experimenting with parts of the code. The more you repeat the first-skim-then-revisit-cycle the better you will get without wasting too much time on less important details.

One way to soften the steep learning curve is to start with crash courses like this one. So without further ado, let us begin.

3 Setup and installations

In order to run the Python code you only need a webbrowser, if you use Google's Colab Btw. Aurélien Geron's source code used in his book is also publicly available on Colab, even though it may not be very beginner friendly (see [GeronColab]).

However, I recommend running everything locally on your computer for the ease of use. We will be using Visual Code, which has a lot of nice comfort functions that are probably not available on Colab. The only downside is that the initial installation takes a bit of effort and about 10 GB space in total.

3.0.1 Jupyter notebooks

First you can install Visual Code and the Jupyter extension. Jupyter allows you to run certain parts of your code in any order you like. We will refer to these code parts as **Jupyter cells**.

All you need to do is create a Python script file with the .py ending, and open it in Visual Code. Each cell is separated by #%% at the beginning of each line. If you want to see how it works, you can experiment with the execution order of the cells in the code below.

Algorithm 1 Test

```
#%%
x = 3
#%%
print(x)
#%%
del x
```

Once you run a Jupyter cell, an interactive window will open in Visual Code, which shows you the outputs like numbers, arrays, tables or even plots.

Note that the **interactive window** may have a restart button, where all variables are reset, but this does not necessarily apply to module-level attributes like __file__. In this case you have to close the interactive window, before you can safely run the cells from a new script file. Otherwise some problems might occur, where e.g. __file__ is the file directory of a previously executed python script instead of the current one. Even restarting Visual Code itself is not a substitute for starting a new interactive window.

The reason, why Jupyter cells are useful, is that you can debug or modify the code without repeating previously computed time intensive cells. Of course you should be careful with this functionality. Sometimes it is better to restart the whole code from scratch before causing too much chaos.

3.1 Anaconda

After installing Jupyter, your Python setup also needs the core modules required for machine learning. Instead of downloading them separately, you can install **Anaconda**, which is widely used

for data science, because it can handle module dependencies well. It should be also compatible with other Python based tasks that are not related to machine learning.

Since I will not guide you through the Anaconda installation, you can get the instructions elsewhere (e.g. https://github.com/ageron/handson-ml3/blob/main/INSTALL.md).

In Visual Code you should be able to able select the Python version in the bottom right bar in order to activate the **conda**-version of Python. If everything worked well, then the following test code will run without problems.

Algorithm 2 Test

```
#%%
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

However, if there are some issues, you should remove other versions of Python on your computer, remove their environment variables from your system and user paths, and make sure that only the Anaconda paths are there (e.g. C:\Users\YourName\anaconda3). Otherwise Visual Code may not recognize modules that have been already installed for Anaconda. Also make sure to close the current interactive window before starting a new one.

In case you ever need to install missing modules, you should use the Anaconda Prompt (not the Anaconda PowerShell Prompt or any other command prompt). In addition, it is recommended to use commands starting with "conda install" instead of "pip install". E.g. if you ever need a specialized module like **xgboost**, then you can install it with

Algorithm 3 Anaconda Prompt

```
conda install -c conda-forge xgboost
```

In the command prompt you may also have to confirm the installation by entering "y".

3.2 Kaggle competitions

One way to practice machine learning is to participate in Kaggle competitions. We will demonstrate this with a competition for beginners: House Prices - Advanced Regression Techniques. There you can download the required data, which in this case is a rather small zip file.

4 How to read this guide

The source codes labeled as "**Jupyter Cell**" can be pasted in one python file in order to create a new Jupyter cell each time. The code labeled as "**Test**" is rather meant to deepen the understanding of the main code, but it is not required for any subsequent cell. There is also some code labeled as "**Output**", which just shows you the result of one of those cells.

5 Machine Learning with Python

In this guide we will start with some basics, then look at a small project example in order to get quickly familiarized with important concepts, and finally we will look at a bigger project example to show you further crucial steps in more detail.

The procedure in both projects is basically the same. First the dataset is transformed in a way that improves the quality of the **prediction model**. This is done by sending the dataset through a so called **pipeline**. Then the model is trained with known targets, before it can predict unknown targets from another dataset (supervised learning).

5.1 First Steps

Usually each Jupyter cell generates at most one output in the interactive window, but it can be very practical to have multiple outputs per cell. This can be achieved with the **display** function.

Algorithm 4 Jupyter Cell

```
#%%
from IPython.display import display
display("Hello world!")
display("second message")
```

Next we need to import the modules **numpy**, **pandas** and **matplotlib.pyplot** (usually abbreviated as **np**, **pd** and **plt**) that will help us to explore the data.

Algorithm 5 Jupyter Cell

```
#%%
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

While **pyplot** is used for visualizing data, the **numpy** and **pandas** modules provide us with three types of extensively used data containers. Before we dive deeper into the machine learning aspect of this guide, we should get familiar with the containers.

5.1.1 Numpy and Pandas Containers

Numpy Arrays or ndarray typed variables are often used for numerical computations due to their better performance speed. The data is stored in n-dimensional arrays. E.g. a 1D array has the shape of a vector, a 2D array the shape of a matrix, etc. The type of data stored in any given ndarray must always be the same, but other than that the data can have any valid type.

Pandas DataFrames are always 2 dimensional arrays, but in addition their rows and columns have a label as well. In this sense they are very similar to spreadsheets or SQL tables. Usually column lables describe the **features** of the data, while the row labels describe each **sample** or instance (usually an index). For this purpose dataframes can store data with different types, even when they are stored in the same dataframe variable.

Pandas Series are the 1 dimensional version of dataframes. This means their row has a label, and they can contain mixed data types as well. Whenever we extract a single column from a dataframe, we can obtain a pandas series.

For beginners it can be helpful to print the type of these containers, because sometimes it can be hard to distinguish them.

In order to get familiar with these types, it can be helpful to construct simple containers manually.

Numpy ndarrays First we want to create a numpy.ndarray with 3 rows and 5 columns with the number 1.2 in each entry. Then we want another ndarray containing 2 matrices with each having 4 columns, 3 rows, where the entries are defined by a custom function.

Algorithm 6 Test

```
#%%
myarray = np.full((3,5), 1.2)
display(myarray)
def myfunc(z, y, x):
    return x*100 + y*10 + z
myfuncarray = np.fromfunction(myfunc, (2,3,4))
display(myfuncarray)
```

Here we defined the dimensions of the ndarrays in n-tuples, which refers to the **shape** of an array. If you read these shapes from right to left, they give you the number of the columns, rows, etc.

An important programming concept is that we can operate along dimensional directions in arrays. Each dimensional direction is called an **axis**. E.g. moving along the direction of columns gives us the 0th axis, rows give use the 1st axis, etc. This makes more sense, once we look at the output of the previous code.

Algorithm 7 Output

```
array([[1.2, 1.2, 1.2, 1.2, 1.2],
       [1.2, 1.2, 1.2, 1.2, 1.2],
       [1.2, 1.2, 1.2, 1.2, 1.2]])

array([[[  0., 100., 200., 300.],
        [ 10., 110., 210., 310.],
        [ 20., 120., 220., 320.]],

       [[  1., 101., 201., 301.],
        [ 11., 111., 211., 311.],
        [ 21., 121., 221., 321.]]])
```

You can see a bunch of nested arrays inside brackets. The arrays in the inner most brackets have the same lengths as the number of columns. More precisely, the i-th inner array has the entries of the i-th matrix-row, which has the form row_i = [e_1, ..., e_n] with n being the number of columns (along **axis 0**).

On the next level we have a matrix of the form [row_1, ..., row_m], where m is the number of rows (**axis 1**). Then we could have an array of matrices of the form [matrix_1, matrix_2, ...] (**axis 2**). This pattern continues until we reached the last axis of the array.

E.g. the first array in the previous output has axis 0 length = 5 and axis 1 length = 3, while the scond array has axis 0 length = 4, axis 1 length = 3 and axis 2 length = 2.

The axis length should not be confused with the **array dimension** (or **rank**), which equals the total amount of axes. (As we will see later, only 2 dimensional arrays can be used for "pipelines".) In our example **myfuncarray** is a 3D and **myarray** a 2D array.

Another important value is the **size** of an array defined as the product of all axis lengths in the array shape (e.g. **myfuncarray** has the size $4 \cdot 3 \cdot 2 = 12$). Obviously, each time we want to reshape an array in order to rearrange its entries, the array size must be the same.

Of course viewing arrays like in the previous output can become hard to read at some point, but in most cases we do not have to look at the entirety of the entries. For instance we can get a single entry like **myfuncarray[1,0,3]**. Since the indices start counting at 0, this would give you the entry of the first row in the fourth column from the second matrix in **myfuncarray**, which is 301.0 in this case.

With Python's **slice indices** we can also replace each single index with a whole range of them. The syntax requires colons like in $i_{\text{Start}} : i_{\text{End}}$, where the indices start at i_{Start} and end at $i_{\text{End}} - 1$. It is also possible to omit i_{Start}, in which case the indices will start at 0. Similarly, we can omit i_{End} in order reach the last available index. E.g. if we only wanted the second and third columns of the very first matrix in **myfuncarray**, then we would write

Algorithm 8 Test

```
#%%
simplematr = myfuncarray[0, :, 1:3]
display(simplematr)
display(myfuncarray.shape)
display(simplematr.shape)
display(simplematr.ndim)
```

The output shows us that, while the shape of the original 3D array **myfuncarray** is (2, 3, 4), the new array **simplematr** has the shape (3, 2) and a dimension of 2.

Note that changing the contents of an array slice also changes the contents of the original array accordingly. E.g.

Algorithm 9 Test

```
#%%
simplematr[2,1] = 999
display(myfuncarray[0,2,2])
```

gives us the output **999**. If you do not want this kind of behavior, you can copy the content with the **.copy()** method. E.g.

Algorithm 10 Test

```
#%%
simplematr2 = myfuncarray[1, :, 1:3].copy()
simplematr2[2,1] = 999
display(myfuncarray[1,2,2])
```

passes the array "by value" instead of passing it "by reference". In general we say in the first case that the variable is returned as a **view** and in the second case as a **copy**. As a result the output of this code is **221** instead of **999**.

In some cases it is important to leave the dimension of an array untouched (e.g. for "pipelines"). This can be done either by using the colons syntax in each index component, or we can write single indices in double brackets. The following code shows both approaches. Here we want the second matrix of **myfuncarray** in form of a 3D array instead of 2D.

Algorithm 11 Test

```
#%%
newmatr_1 =  myfuncarray[1:2,:,:]
newmatr_2 =  myfuncarray[[1],:,:]
```

You can check that both **newmatr_1** and **newmatr_2** have the shape (1, 3, 4). This means that these arrays have only one element along their axis 2, which is the matrix we wanted.

At this point we have already learned a lot of important concepts. Still there are two main concepts left that are very versatile especially when combined together: **pointwise operations** and **broadcasting**.

In general a pointwise operation is applied on each entry of the involved arrays. Therefore, binary operations (like "+", "*", "<") can only be used for arrays with the same shape. Under certain conditions it is possible that arrays can be reshaped automatically in a way that these operations become possible (broadcasting).

Before we explain the rules of broadcasting in general, we want to look at a simple example, where we introduce two 1 dimensional arrays with the size 7. Note that in this case the shape is not written as (7) but rather as (7,).

Algorithm 12 Test

```
#%%
arr_1 = np.fromfunction(lambda x: x, (7,), dtype = np.int32)
arr_2 = np.fromfunction(lambda x: 2*x, (7,), dtype = np.int32)
display(arr_1)
display(arr_2)
display(arr_1 + arr_2)
display(arr_2 < 5)
display(arr_2[arr_2 < 5])
```

Here we kept the code short by making use of lambda functions, and with **dtype** we can replace the default float number outputs with easier to read integers.

Thanks to the broadcasting rules it is possible to apply binary operations on an array and a number like in **arr_2 < 5**: By simply repeating the number 5 along all axes of **arr_2**, we get matching shapes. Note that numbers can be considered as special types of arrays with the shape (1,). The last line in the output

Algorithm 13 Output

```
array([0, 1, 2, 3, 4, 5, 6])
array([ 0,  2,  4,  6,  8, 10, 12])
array([ 0,  3,  6,  9, 12, 15, 18])
array([ True,  True,  True, False, False, False, False])
array([0, 2, 4])
```

demonstrates a very hepful technique, where a 1D array with boolean values is used to select rows from another 1D array with the same size.

For now it should be enough to understand the broadcasting rule for numbers, but you should be already able to understand the rules in general, if you want:

1. If two arrays of the same dimension k have the shape $(n_0, \ldots, n_{k-1})$ and $(m_0, \ldots, m_{k-1})$, respectively, and if $n_i \neq m_i$ always implies that $n_i = 1$ or $m_i = 1$, then both arrays can be matched. E.g if $n_i = 1$, then there is only one element along the axis i. This element can then be repeated m_i times along the axis i until the axis lengths of both arrays match.

2. If two arrays do not have the same dimension, then the dimension of the smaller array is matched by appending axis lengths of "1" to the left of the shape-tupel. E.g. arrays with the shape (7, 5, 3) and (3,) are matched by changing the shape of the second array to (1, 1, 3), and then repeating the entries in the first and second axes until the second array has the shape (7, 5, 3) as well.

Of course these broadcasting rules do not apply in other circumstances, when there is an axis i such that $n_i \neq m_i$ and $n_i \neq 1 \neq m_i$.

Pandas dataframes and series Some of our knowledge about numpy ndarrays can be applied to pandas series and dataframes as well. Before diving in, we want to define a dataframe **df_customer_raw** storing the data of customer transactions. Each transaction has features like the product type (in form of strings), the price (floats) and the amount of products sold (integers).

Algorithm 14 Test

```
#%%
customer_list = [
    ["Laptop", 1199.99, 10],
    ["Phone", 812.35, 50],
    ["Tablet", 300.0, 1],
    ["Monitor", 511.5, 35],
]
df_customer_raw = pd.DataFrame(data = customer_list)
display(df_customer_raw)
```

If we define dataframes like this, their column and row labels will be automatically generated indices. More useful labels can be added, if we initialize dataframe with the arguments **index** and **columns**. In our example the rows have customer names and the columns feature names. The following code also shows how we can get the numpy ndarray version of a dataframe by using **.values**.

Algorithm 15 Test

```
#%%
df_customer = pd.DataFrame(
    data = customer_list,
    index = ["Alice", "Charles", "Bob", "Dan"],
    columns = ["Product", "Price", "Sold"]
)
display(df_customer)
display(df_customer.values)
```

The output

Algorithm 16 Output

```
        Product Price Sold
Alice  Laptop 1199.99 10
Charles  Phone 812.35 50
Bob  Tablet 300.00 1
Dan  Monitor 511.50 35

array([['Laptop', 1199.99, 10],
       ['Phone', 812.35, 50],
       ['Tablet', 300.0, 1],
       ['Monitor', 511.5, 35]], dtype=object)
```

shows that the datatype of the ndarray is **object**. This type already contains mixed types like strings or floats. Otherwise it would not be possible to mix different types in a ndarray. (Note that **object** types can decrease the computational time compared to pure numerical types.)

If we want to access specific columns or rows from dataframes, they will be converted to 1D pandas series.

Algorithm 17 Test

```
#%%
display( df_customer["Product"] ) #column
display( df_customer.loc["Bob"] ) #row
display( df_customer.loc["Bob", "Product"] ) #data
```

Here **.loc** gives us the row for the specified label **"Bob"**. If we ommitted **.loc**, we would get a column with the label **"Bob"**, which does not exist in our dataframe.

In the last line we could have also used the more intuitive syntax **df_customer["Product"]["Bob"]** instead, if we only wanted to read the data. However, if we intend to write the data, it is not guaranteed whether the data is returned as a copy or a view (it depends on internal implementations).

In addition, using **.loc** makes our intention clear, whether we want to refer to so called **label positions** or **ordinal positions**. E.g. **"Bob"** is a label position, but its ordinal position in our **index** list is 2 (while e.g. **"Alice"** has the position 0). Therefore an alternative way to access Bob's customer data is

Algorithm 18 Test

```
#%%
df_customer.iloc[2]
```

because **.iloc** always accesses ordinal positions. One of the advantages of ordinal positions is that we can use slicing in combination with **.iloc**. Just like for numpy ndarrays the slicing method preserves the dimension of the 2D dataframe instead of transforming it into a 1D series. E.g. **df_customer.iloc[0:2]** gives us the first two customers in form of a new dataframe. Of course slicing also works for series.

The general rule of thumb is to use **.iloc** for ordinal and **.loc** for label positions. Otherwise it can create confusing situations, where integer numbers are used as labels, while these labels do not represent the real ordinal positions.

As you may have noticed in some of the outputs, each series extracted from dataframes has a name equal to the label of the corresponding column or row. Conversely, we can add a series as a new column in dataframes. So let us add the birthyear for each customer with

Algorithm 19 Test

```
#%%
birthcol = pd.Series(
    data = [1990, 2000, 1980, 2010],
    name = "Birthyear",
    index = df_customer.index
)
df_customer[birthcol.name] = birthcol
display(df_customer)
```

It is a very intuitive and short way to create new columns like this, but if assume that **df_customer** was a slice of a bigger dataframe, then this syntax would not guarantee whether the original dataframe would also change after creating the new column. In those cases it is recommended to use the **.assign** method, which requires a lengthier syntax like **df_customer = df_customer.assign(Birthyear = birthcol)**.

For starters it is already sufficient to apply **.assign**. for this reason only. However, another situation where **.assign** is useful is for so called **chaining** methods, when we want to execute several operations in one expression without the need for intermediate variables.

This means we can create multiple new columns in a single **.assign** chain, even if some columns depend on others initialized earlier in the chain. We only have to make sure that **.assign** does not try to initialize all columns simultaneously. This is why we have to wrap the initializations in lambdas or functions, which ”delay” these operations until they are called sequentially in the chain:

Algorithm 20 Test

```
#%%
df_new = (df_customer
    .assign(Revenue = lambda df: df["Price"] * df["Sold"])
    .assign(HighProfit = lambda df: df["Revenue"] > 11000)
)
display(df_customer)
display(df_new)
```

Note that with this method changing any data inside **df_new** does not change **df_customer** at all. You also do not have to fear causing too much overhead thanks to "Copy-on-Write", where the memory of the new and the original dataframes share the same memory until they are changed.

Last but not least, we can remove existing columns or rows with **.drop**.

Algorithm 21 Test

```
#%%
df_customer = df_customer.drop(columns=["Sold",  "Price"])
df_customer = df_customer.drop(index=["Bob"])
display(df_customer)
```

A more traditional way of dropping columns or rows is to use axes arguments, i.e. we could have also used **df_customer.drop("Bob", axis = 0)** for dropping a row.

Now that you are familiar enough with numpy/pandas data container, you should have a solid knowledge to move on. However, if you want to deepen your knowledge at some point, I can recommend Aurélien Géron's online tutorial on Colab [GeronColab]. Or you can try out books like [VanderPlas].

5.1.2 Getting the data

Before analyzing the data, we have to load them into the system. There are multiple ways to do this. For instance we can automatically download and extract the data, or even use SQL directly in Jupyter cells in order to handle large datasets efficiently.

For now we want to keep it simple. Download and extract the **train.csv** and **test.csv** files manually from the Kaggle competition site, and save them in the same folder as your Python script.

Algorithm 22 Jupyter Cell

```
#%%
from pathlib import Path
sLocal_Folder_Path = Path(__file__).parent.resolve()
train_file_path = sLocal_Folder_Path / "train.csv" # Uses OS-appropriate separator
predict_file_path = sLocal_Folder_Path / "test.csv"
housing = pd.read_csv(train_file_path)
housing_unknown = pd.read_csv(predict_file_path)
display(housing)
display(housing_unknown)
```

Here we have stored the data as **housing** and **housing_unknown** dataframe variables. After running this cell you will see the table structure of **housing** and **housing_unknown**. The output should show you a few example rows for both dataframes.

It should also tell you that **housing** has 81 columns, while **housing_unknown** has only 80 columns. The missing column are the targets, which are the house sales prices we will have to predict before submitting it on Kaggle. The dataset from **housing** on the other hand will be used to train and test our prediction models before our final submission.

5.2 Small Project

Make sure you add the Jupyter cells from the previous **"First Steps"** section (algorithms 4 and 5).

In our case one **sample** is the data of a single house with **features** like the total number of bedrooms, while the target is simply the sale price. Before we can train a model we have to separate the target-column from the feature columns.

In addition, we need to separate the so called **train set** from the **test set**. One way to achieve this is by using the function **train_test_split**.

Algorithm 23 Jupyter Cell

```
#%%
from sklearn.model_selection import train_test_split
train_set, test_set = train_test_split(
    housing, test_size = 0.05, random_state = 42
)
housing = train_set.drop("SalePrice", axis=1)
housing_targets = train_set["SalePrice"].copy()

housing_final_test = test_set.drop("SalePrice", axis=1)
housing_targets_final_test = test_set["SalePrice"].copy()
```

The test set will help us to measure the performance of our model. This is done by computing the predicted target $h(x_i)$ based on the feature values of a sample x_i, and comparing it with the known targets y_i. Then we can estimate the error with measures like the so called **root mean square error** (**RMSE**)

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(y_i - h\left(x_i\right)\right)^2}$$

where n is the number of samples. A smaller RMSE indicates that the prediction of our model is more precise.

Of course we could use the samples x_i and targets y_i from the training set, i.e. the old dataset on which our model was trained. The problem is that the model prediction may be much worse for samples that are not part of the training set. The reason is that models can "get used" so much to the training set that they do not generalize very well.

This is called overfitting. It usually occurs, when the prediction model has too much freedom for finding highly specialized rules that can only predict the trained targets very well.

Therefore it is more reliable to calculate an error measure on the test set instead of train set. In our case the test set has only about 0.05% (**test_size**) of the original samples. Usually you want a more reliable test set ratio, but in our case it would decrease the number of samples in the training set too much, if **test_size** was much larger.

Next we want to choose one of the prediction models that is already available from the **sklearn**-module. For this simple example a linear regressor is fine. **Regressors** are used for predicting numerical targets like the house sale prices, otherwise we would need a **classifier** for predicting categorical targets (categorical targets are also called labels, which should not be confused with the column/row labels of dataframes).

Algorithm 24 Jupyter Cell

```
#%%
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import Pipeline
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import FunctionTransformer
from sklearn.compose import TransformedTargetRegressor
from sklearn.compose import ColumnTransformer
```

As you can see, we have also imported other modules that introduce new objects like **pipelines** and **transformers**. Before we continue with our project, we should take a look at a simple application of pipelines and transformers, where the values of the features **"OverallQual"** and **"GarageArea"** are incremented by +1 and then multiplied with the factor 2 after being transformed by the **example_pipe** pipeline.

Algorithm 25 Test

```
#%% Test
def increment_func(x):
    return x + 1
def double_func(x):
    return x * 2
example_pipe = make_pipeline(
    FunctionTransformer(func = increment_func),
    FunctionTransformer(func = double_func)
)
example_df = housing[["OverallQual", "GarageArea"]]
example_pipe.fit(example_df)
example_trafo = example_pipe.transform(example_df)
display(example_df)
display(example_trafo)
```

If e.g. an entry in **example_df** equals 3, then the corresponding entry in **example_trafo** is $(3+1)\cdot 2 = 8$. Note that the double brackets in **housing[["OverallQual", "GarageArea"]]** result in a dataframe with two feature columns from the **housing** dataframe. If we used only single brackets, we would have obtained a 1D pandas.series, which cannot be used as an input for pipelines.

In the next subsection, we will explain pipelines and transformers in a broader context.

5.2.1 Terminology

Features or **Attributes** are measurable properties of samples, although some samples may also have missing features.

Targets are known properties in the in train sets, but usually unknown other datasets.

Prediction Models in supervised learning tasks can predict targets based on their training and their train set. There are also unsupervised prediction models like cluster search models. Depending on whether the targets are numerical or categorical, prediction models are also called **regressors** or **classifiers**, respectively.

Estimators are objects in **sklearn** that can train their internal prediction model with their **.fit**(**X, y**) method (where **X** are the features and the targets **y** of the train set). Estimators also have the method **.predict**(...) to predict unknown targets.

Transformers are a bit similar to estimators, although **.fit**(...) has a more general meaning in the sense that it adjusts its internal parameters to the input of **.fit**(...) (a second argument is not needed here). Afterwards we can use the **.transform**(...) method to return a transformed version of its argument. For the purpose of calling e.g. **trafo.fit**(**X**) first and then **X** = **trafo.transform**(**X**), we may use the shortcut **X** = **trafo.fit_transform**(**X**) instead, which also has a faster computation speed.

Pipelines are objects that contain a sequence of transformers, estimators or other existing pipelines (for nested pipelines). Each pipeline has a **.fit**(...) method that accepts a single 2D container as input, which must be either a pandas.dataframe or 2D numpy.ndarray. This input is then passed to the argument of **.fit_transform**(...) of the first transformer in the pipeline. Afterwards the output of this **.fit_transform**(...) is used for the input of the next transformer, etc. This process is repeated until the last element of the pipeline is reached, where only **.fit**(...) is called.
Finally, the pipeline object exposes the methods of its last element, i.e. the pipeline has a **.transform**(...) or **.predict**(...) method depending on whether its last element is a transformer or estimator, respectively.

Column Transformers are used to apply different transformers to selected sets of columns only. They are initialized by a list of 3-tuples, each of them representing a separate transformation. The first 3-tuple component is the string name of the transformation, the second component a transformer or even a whole pipeline, and the third component a list of feature labels for selecting the desired columns (in the case of 2D ndarrays these labels are integer indices).

5.2.2 Making predictions

Now we can take a look at the pipelines we are interested in.

Algorithm 26 Jupyter Cell

```
#%%
target_trafo = make_pipeline(
    SimpleImputer(strategy = "mean", add_indicator=True),
    FunctionTransformer(func = np.log, inverse_func = np.exp),
    StandardScaler()
)
feature_log_pipeline = make_pipeline(
    SimpleImputer(strategy="mean"),
    FunctionTransformer(func = np.log),
    StandardScaler()
)
feature_pipeline = make_pipeline(
    SimpleImputer(strategy="mean"),
    StandardScaler()
)
```

Here **make_pipeline** inserts the transformers in the piplines in the order it receives them in its argument list. E.g. the **target_trafo** pipeline (which will be applied to the house sale price sales later)

first applies the **SimpleImputer** transformer, which replaces possibly missing values with the mean value of all available values in the column (see **strategy = "mean"**). Even if there are no missing values, we may sometimes require that the very first transformer returns a 2D numpy.ndarray, which can be more practical for the remaining transformers in the pipeline. In this case **SimpleImputer** transforms a dataframe into a 2D ndarray.

Unlike for the transformation of features, we need to transform the targets back to their original form, i.e. an inverse transformation must be available. In **SimpleImputer** this is ensured by the argument **add_indicator=True**.

Next we have the **FunctionTransformer** in the **target_trafo** pipeline. In our example it simply returns the logarithm of its input values. For now it suffices to know that this will help our linear regressor to make better predictions. This time we make sure the inverse transformation is available by adding the argument **inverse_func = np.exp**.

The final transformer for our targets is **StandardScaler**, which centers the values around the mean of all values and scales it in a way that will improve the performance of our linear regressor. Otherwise unscaled values can become biased in favor of the available values on the whole scale instead of the actually occuring amount of values.

In a similar fashion we define the other pipelines **feature_log_pipeline** and **feature_pipeline** for the features. Another way to use the function transformer is to define custom functions. In our case we want to be able to calculate the ratio of two columns.

Algorithm 27 Jupyter Cell

```
#%%
def column_ratio(X):
    return X[:,[0]] / X[:,[1]]
feature_ratio = make_pipeline(
    SimpleImputer(strategy="median"),
    FunctionTransformer(func = column_ratio),
    StandardScaler()
)
```

As mentioned earlier, it is possible to apply transformers on different columns separately, if we use column transformers.

Algorithm 28 Jupyter Cell

```
#%%
column_trafo = ColumnTransformer(
    transformers = [
        ('num_log', feature_log_pipeline, ["GrLivArea"]),
        ('ratio', feature_ratio, ["BedroomAbvGr", "GrLivArea"]),
        ('num', feature_pipeline, ["OverallQual", "GarageArea"])
    ],
    remainder = "drop"
)
```

For the sake of simplicity, we ignore all other columns in the dataset (**remainder = "drop"**), although there are other options available to handle the remaining columns.

An alternative to **make_pipeline** is the **Pipeline** class itself, which allows us to name each object in the pipeline manually. These names can play a role for the automatic optimization of so called

hyperparameters, which are those parameters that cannot be adjusted by the prediction models during their training phase. However, we will not need to optimize any hyperparameter for this simple project.

Finally we can combine our previous pipelines and our prediction model in a single pipeline.

Algorithm 29 Jupyter Cell

```
#%%
regressor_pipeline = Pipeline([
    ('features', column_trafo),
    ('linear', LinearRegression())
])
```

Afterwards we can assemble all pipelines in a single estimator.

Algorithm 30 Jupyter Cell

```
#%%
model = TransformedTargetRegressor(
    transformer = target_trafo,
    regressor = regressor_pipeline
)
model.fit(housing, housing_targets)
housing_predicted_prices = model.predict(housing_final_test)
```

More precisely, this is called a **meta-estimator**, which enhances the functionality of normal estimators. Here the meta-estimator **TransformedTargetRegressor** automatically applies the inverse functions to the predicted targets. This is a bit less error-prone than applying each inverse function manually.

Of course there are much more interesting meta-estimators like **RandomizedSearchCV** used for hyperparameter tuning, or meta-estimators used for mixing different types of estimators.

In our simplified project the linear regressor is enough to make decent predictions. We store them in the **housing_predicted_prices** variable before measuring the RMSE with the test set.

Algorithm 31 Jupyter Cell

```
#%%
from sklearn.metrics import root_mean_squared_error
lin_rmse = root_mean_squared_error(
    housing_targets_final_test, housing_predicted_prices
)
print((lin_rmse / 1e3).round(2), "thousand dollar RMSE for housing prices")
```

The resulting RMSE should be around 28 thousand dollars. Finally, we can create a **submission.csv** file that has the right format for submitting it on Kaggle.

Algorithm 32 Jupyter Cell

```
#%%
housing_predicted_prices = model.predict(housing_unknown)
submission = pd.DataFrame({
    'Id': housing_unknown['Id'],
    'SalePrice': housing_predicted_prices
})
submission.to_csv(sLocal_Folder_Path / 'submission.csv', index=False)
```

The final score should be below 0.20 (less means better), which is acceptable for such a small project.

Before moving on, you can play around a little bit by removing the logarithm-transformations to see how the linear regressor performs. You can also replace it with another regressor like the **RandomForestRegressor**, where the performance does not depend very much on log-transformations or the standard scaler (except in extreme cases).

Once you have understood the most important concepts in this small project, you can tackle the same problem with more sophisticated methods in the big project that comes next. This will give you a much better score and a deeper understanding of the problems occuring in machine learning.

5.3 Big project

Compared to the previous small project the learning curve will be steeper here, but you will also learn a lot more.

Just like we did in the **First Steps** section, we need to import the basic modules and load the data.

Algorithm 33 Jupyter Cell (repetition)

```
# %%
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# %%
from IPython.display import display
from pathlib import Path
sLocal_Folder_Path = Path(__file__).parent.resolve()
train_file_path = sLocal_Folder_Path / "train.csv" # Uses OS-appropriate separator
predict_file_path = sLocal_Folder_Path / "test.csv"
housing = pd.read_csv(train_file_path)
housing_unknown = pd.read_csv(predict_file_path)
```

One way to get a grasp on large datasets is to plot the **histogram** of all numerical features. The numerical values are grouped together in **bins** that are arranged along the x-axis. The bar length along the y-axis shows how many samples occured in each bin. The following code plots the histograms of each feature next to each other.

Algorithm 34 Jupyter Cell

```
# %%
housing.hist(bins=50, figsize=(30,25))
plt.show()
```

When looking at histograms, it is often important to find heavy-tailed distributions (e.g., the left plot below). These distributions are asymmetrical in the sense that most of the values are amassed on one side of the x-axis, while the other side is "stretched thin" on the other side.

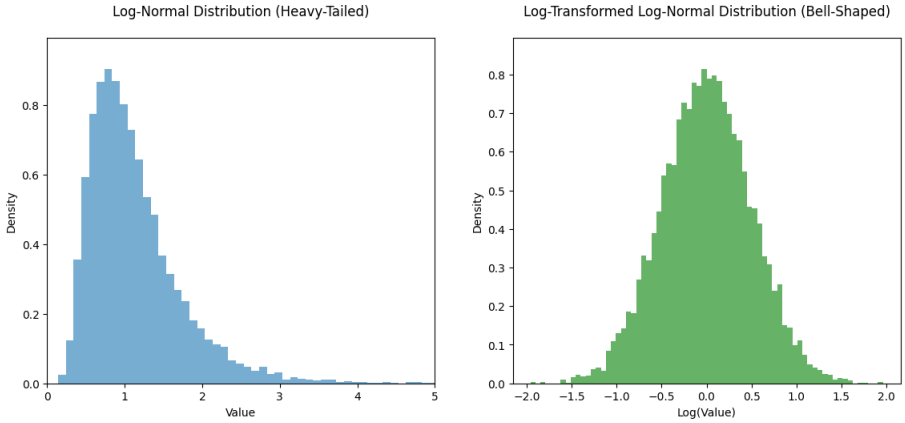

Figure 5.1: Heavy-tailed vs. normal distribution (generated with deepseek)

Heavy-tailed distributions can be usually converted to bell-shaped normal distribution by calculating the logarithm of the numerical values (on the x-axis). Many regressors can handle normal distributions much better.

For now we will not worry about transforming the dataset. Instead, we collect all heavy-tailed feature labels in the **housing** dataset, and list them in **heavy_tailed_features** for later use.

Algorithm 35 Jupyter Cell

```
# %%
heavy_tailed_features = ["LotFrontage", "LotArea", "1stFlrSF", "TotalBsmtSF", "GrLi-
vArea"]
housing[heavy_tailed_features].hist(bins=50, figsize=(12,8))
plt.show()
```

The output of this cell shows us the heavy-tailed features only.

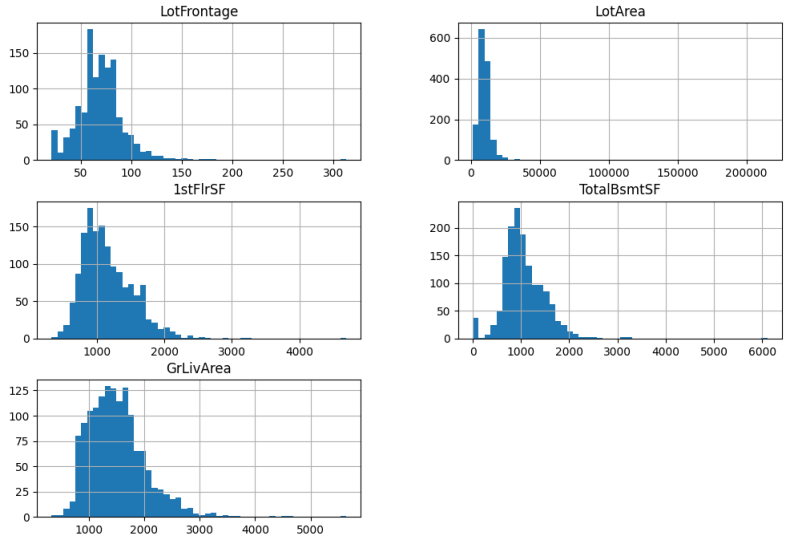

Figure 5.2: Heavy-tailed vs. normal distribution

Besides the numerical features we have to deal with categorical ones as well. Fortunately, there is a file **data_description.txt** from the zip-file we downloaded from Kaggle. If you open it in a text-editor, you can see a description of all features like e.g.

Algorithm 36 Snippet from data_description.txt

```
# %%
ExterQual: Evaluates the quality of the material on the exterior
       Ex Excellent
       Gd Good
       TA Average/Typical
       Fa Fair
       Po Poor
```

Thanks to this description we can try to convert the feature labeled as **ExterQual** into a numerical one. It may seem curious at first, but since evaluations like **Excellent** or **Poor** are based on human estimations, it makes sense to transform them into fibonnaci-numbers rather than linearly increasing ones.

The reason behind this is that the difference between two adjacent numbers in a fibonnaci sequence is increasing based on the previous number in the sequence. Otherwise humans would have a harder time to distinguish between them (for more details see e.g. [ScrumRef]).

As a further justification for this approach, we will see that the numerical value of **ExterQual** is quite important for predicting the sales price targets.

Of course there are also features, where it does not make much sense converting them to any number like e.g. the type of the roof material. For now we will keep them as categorical values (i.e. strings).

For our feature-to-number-mapping we can use dictionaries in Python. Since putting all feature-mapping dictionaries in one .py file would clutter the code, we store them in a file **KagglDataC1.py** instead, and put this file in the same folder as our main Python script.

Algorithm 37 Snippet from KagglDataC1.py

```
# %%
# ExterQual: Evaluates the quality of the material on the exterior
fibonacci_mapping_ExterQual = {
"Po": 1,    # Poor
   "Fa": 2,  # Fair
   "TA": 3,  # Average/Typical
   "Gd": 5, # Good
   "Ex": 8, # Excellent
}
```

The complete file can be found on my Github page (see [1]), but you can also create it yourself. Now we can go back to our main Python file, and add the following cell.

Algorithm 38 Jupyter Cell

```
# %%
from KagglDataC1 import *
ranked_category_columns = ["BsmtQual", "BsmtCond", "BsmtExposure",
    "BsmtFinType1", "BsmtFinType2", "HeatingQC", "KitchenQual",
    "Functional", "FireplaceQu", "GarageFinish", "GarageQual",
    "GarageCond", "PavedDrive", "PoolQC", "Fence", "ExterCond", "ExterQual"
]

def transform_categories_to_ranked(data):
    for col in ranked_category_columns:
        data[f"Ranked_{col}"] = data[col].map(globals()[f"fibonacci_mapping_{col}"])
    data = data.drop(columns=ranked_category_columns)
    return data
housing = transform_categories_to_ranked(housing)
housing_unknown = transform_categories_to_ranked(housing_unknown)
```

This imports the contents of our KagglDataC1.py file, automatically converts features like **ExterQual** to a numerical value, adds them as new features in the dataframes **housing** and **housing_unknown**, and deletes the columns of the original non-numerical features. In addition, we added the prefix "**Ranked_**" to the new features in order to distinguish them from other features.

Of course we should check, whether the modified housing dataframes are correct. One way to do that is to run

Algorithm 39 Jupyter Cell

```
# %%
display(housing_unknown.info())
display(housing.info())
```

with the **.info()** method. It shows a list of all features labels from each column together with their data type and the amount of samples that have this feature. E.g. **int64** stands for integers, **float64** for float values, whereas **object** indicates a non-numerical or mixed feature type.

If the output of **.info()** is too large, you can maximize the whole list by clicking on "**scrollable element**" in Visual Code. If everything went fine, then it should look like this.

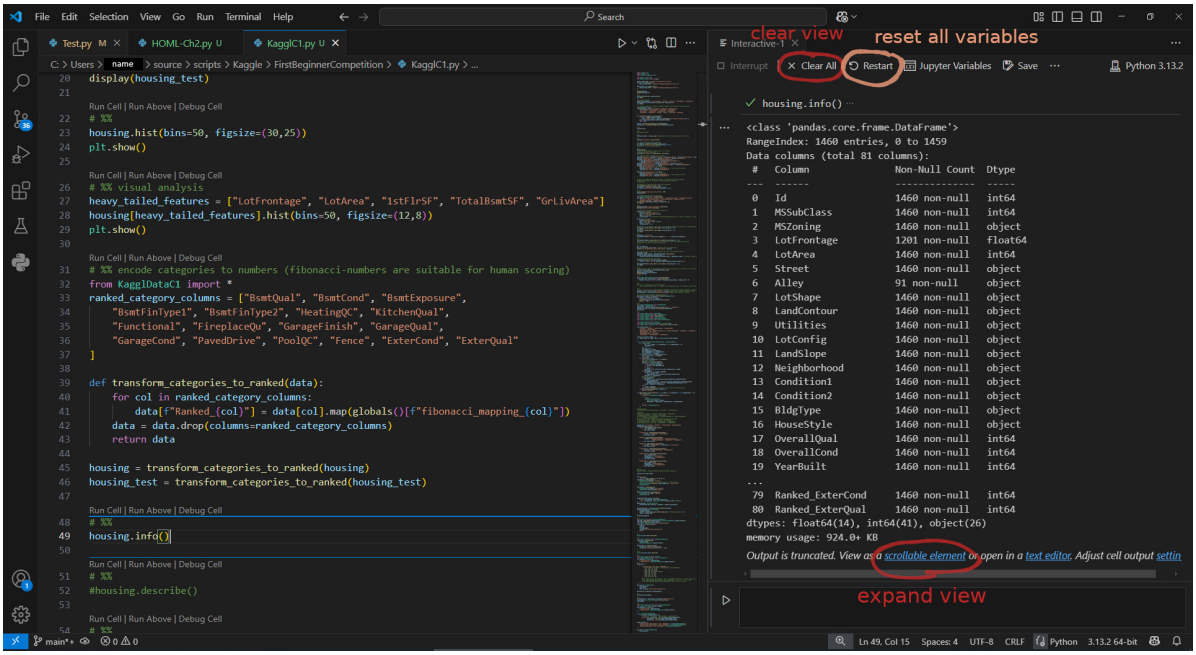

Figure 5.3: Visual Code showing housing info

If there is no option to show the complete output, then it is probably because there are too many columns/features in your dataset. Fortunately you can fix this by changing the settings with a code like

Algorithm 40 Jupyter Cell

```
# %%
pd.set_option('display.max_info_columns', 250)
pd.set_option('display.max_rows', 250)
```

For the output of **.info()** the setting **max_info_columns** is enough, but we also want to set **max_rows** for other outputs as well (e.g. when we display the correlation matrix later on).

Another way of looking at the housing dataset is to run

Algorithm 41 Jupyter Cell

```
# %%
housing.describe()
```

which will show you information like the mean-value or the minimum/maximum values of all numerical features in the **housing** dataframe.

The set of numerical features can be further divided in continuous and discrete ones. The discrete features are the ones, where the set of all possible values small or very limited. E.g. the housing feature **OverallQual** can have integer values from 1 to 10 only. For discrete features, we can therefore use

Algorithm 42 Test

```
# %%
(housing["OverallQual"]).value_counts().sort_index()
```

which results in the output

Algorithm 43 Output

```
OverallQual
1       2
2       3
3      20
4     116
5     397
6     374
7     319
8     168
9      43
10     18
Name: count, dtype: int64
```

As you can see there are only 2 houses with a terrible **OverallQual** of 1. Now is a good time to use these methods to explore some of the other features on your own before moving on. Maybe you can find some interesting observations. E.g. you may want to look at how many kitchen or other types of rooms the house samples have.

Later on we will modify the housing dataset for temporary purposes (e.g. stratified samples), which is why we should keep a copy of the current housing dataset.

Algorithm 44 Jupyter Cell

```
# %%
housing_original = housing.copy()
```

Now we should look at the so called **standard correlation coefficients** between each feature. In short, the correlation measures the linear dependency between two features. The correlation is close to +1 or -1, if there is a strong positive or negative linear relationship between two features.

If a more complex dependency exists, or if there is no dependency at all, then the correlation should be close to 0. Of course it can be possbile to transform non-linear dependencies into linear ones before measuring the correlations, but this topic is not part of our current project.

These correlation coefficients are conveniently stored in a so called **correlation matrix**. Since we are very much interested in the correlation of the target sale price and every other feature, we only have to look up the **SalePrice** column of the correlation matrix.

Algorithm 45 Jupyter Cell

```
# %%
corr_matrix = housing.corr(numeric_only = True)
corr_matrix["SalePrice"].sort_values(ascending = False)
```

If you run this cell, you should get a list of the correlations of each feature with respect to the targets. It should not surprise you that **SalePrice** has a correlation of 1.0 with itself.

More interesting examples are **OverallQual**, which has a high correlation of 0.79. There are also features we derived from the fibonnaci-numbers with a relatively high correlation (e.g. **Ranked_ExterQual** has 0.69 and **Ranked_KitchenQual** has 0.68).

If the absolute value of the correlation between a feature and the target is high, then it is a good indicator that it may be important for prediction models. The advantage of this method is that is quite easy to find those features.

Of course it may be also possible that we miss some of the other important features, if their correlation is close to 0. In those cases we would need a more sophisticated analysis.

5.3.1 Creating new features

Based on the old features, we want to create new features that are more meaningful for predict the targets. This requires educated guesses that are tailored to the concrete problem (in this case the house dataset). Here are some thoughts:

- There are features in the housing dataset, which count the amount of full bathrooms **FullBath** (including a shower) and **HalfBath** (i.e. toilets only) separately. Furthermore, these numbers do not take into account the baths in the basement (**BsmtFullBath** and **BsmtHalfBath**). By computing the sum of these four features, we get a more meaningful number of the total amount of bathrooms.
- The total area **GrLivArea** of living space (basement not included) is already an important feature, but if we multiply it with **OverallQual** we may get an even more meaningful one.
- The fibonnaci-ranked features **Ranked_PavedDrive**, **GarageFinish**, **Ranked_GarageQual**, **GarageCars** and **GarageArea** are probably related to each other such that it can make sense to calculate their product.
- The same is probably true for the quality of the heating **Ranked_HeatingQC** and the total amount of rooms **TotRmsAbvGrd** (basement not included).
- Ratios like the amount of bedrooms per living area, or the amount of bathrooms per bedrooms can also lead to important new features.

We can translate these ideas to the code below. Note that when we compute the ratios, we have to be careful not to divide by 0. In our example we solve this problem by checking whether the feature of a given sample is 0, and then provide an alternative feature that is guaranteed to have a different value than 0.

Algorithm 46 Jupyter Cell

```
# %%
housing["bath_sum"] = \
    housing["FullBath"] + housing["HalfBath"] \
    + housing["BsmtFullBath"] + housing["BsmtHalfBath"]
housing["areaquality_product"] = housing["GrLivArea"] * housing["OverallQual"]
housing["garage_product"] = housing["Ranked_PavedDrive"] \
    * housing["Ranked_GarageFinish"] * housing["Ranked_GarageQual"] \
    * housing["GarageCars"] * housing["GarageArea"]
housing["bedrooms_ratio"] = housing["BedroomAbvGr"] / housing["GrLivArea"]
housing["roomquality_product"] = housing["Ranked_HeatingQC"] \
    * housing["TotRmsAbvGrd"]
housing["bath_kitchen_ratio"] = np.where(
    housing["KitchenAbvGr"] != 0,  # Condition
    (housing["bath_sum"]) / housing["KitchenAbvGr"],  # True: Perform division
    (housing["bath_sum"]) / housing["TotRmsAbvGrd"]
)
housing["bath_bedroom_ratio"] = np.where(
    housing["BedroomAbvGr"] != 0,  # Condition
    (housing["bath_sum"]) / housing["BedroomAbvGr"],  # True: Perform division
    (housing["bath_sum"]) / housing["TotRmsAbvGrd"]
)
```

Afterwards we can quickly check how the correlations of the new features look like compared to the old ones.

Algorithm 47 Jupyter Cell

```
# %%
corr_matrix = housing.corr(numeric_only = True)
corr_matrix["SalePrice"].sort_values(ascending = False)
```

The new output tells us e.g. that **areaquality_product** has the highest correlation of any other feature, which is already an improvement. We can also see that **garage_product** has a significantly higher correlation than any of its factors.

We could go further by dropping the old features after replacing them with better ones, or use advanced techniques like the principal component analysis (PCA), but for the sake of keeping this guide simple we will leave the old and new features as they are right now.

If we do not want to rely too much on the correlation coefficients, we can also use the so called **scatter matrix**, where each feature is plotted against each other. This can help to find out non-linear dependencies or clusters. Of course we can also find non-existing dependencies, if the plotted points are mostly aligned around a vertical or horizontal line the plot.

Note that plotting a feature against itself does not result in any interesting plot, which is why they are replaced by their corresponding histogram.

Algorithm 48 Jupyter Cell

```
# %%
from pandas.plotting import scatter_matrix
attributes = ["SalePrice", "garage_product", "areaquality_product",
    "roomquality_product", "bedrooms_ratio",
]
scatter_matrix(housing[attributes], figsize=(10, 10))
plt.show()
```

As a result we obtain the following plot. It shows e.g. how our bedrooms ratio has a non-linear dependency with other features like the sale price.

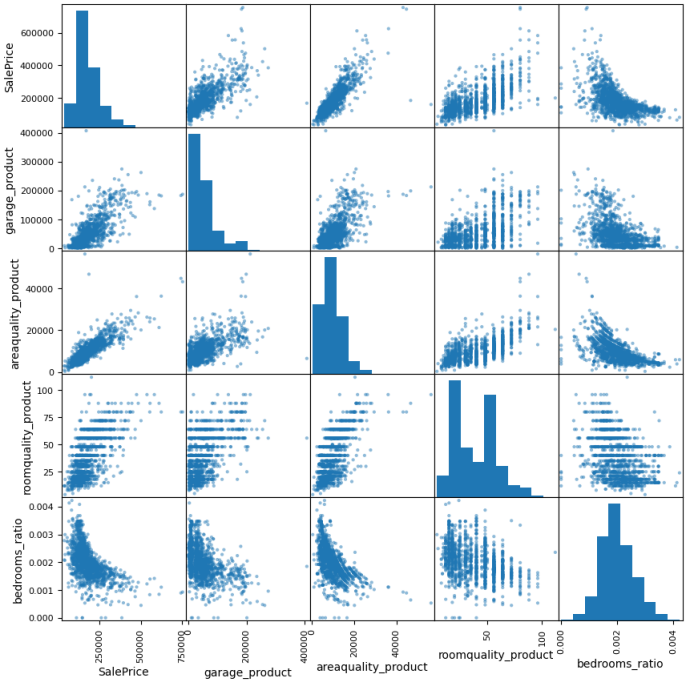

Figure 5.4: Scatter matrix

5.3.2 Strata

Like we have done in our small project we need to split the dataset into train and test sets, but this time we do no want to do this randomly. Instead we want them to be split as evenly as possibly accross certain groups of samples that are called **stratas**. This makes sure that certain stratas are not over- or underrepresented.

E.g. if we want to predict the performance of a new drug, then we want to represent the ages of all patients as evenly as possible. Of course we would have to group different ages together, i.e. we could divide the ages into different bins for the ages 0-10, 10-20, 30-40, etc.

This method is called **stratified sampling**. In our case we have to make an educated guess to find good strata for predicting house prices. E.g. we could use the **areaquality_product** and the **bedrooms_ratio** as stratas. For this purpose we use the function **pd.cut**, where the **bins** define the boundaries of each strata. The stratas can also be named with the function argument **labels**.

Algorithm 49 Jupyter Cell

```
# %%strata_cat_1 = pd.cut(
    housing["areaquality_product"],
    bins = [0, 5e3, 8e3, 12e3, np.inf],
    labels = [1,2,3,4],
    include_lowest = True
)
strata_cat_2 = pd.cut(
    housing["bedrooms_ratio"],
    bins = [0.0, 16e-4, 21e-4, np.inf],
    labels = [1,2,3],
    include_lowest = True
) strata_cat_2 (otherwise outcomment this line)
strata_cat_1.value_counts().sort_index().plot.bar(grid = True)
plt.show()
strata_cat_2.value_counts().sort_index().plot.bar(grid = True)
plt.show()
```

It is noteworthy that argument **include_lowest** prevents the case, where values at the edges of a bin are not included, i.e. transformed into **NaN** (not any number). E.g. if the **bedrooms_ratio** is 0 for a given sample, then **pd.cut** would convert it into **NaN**, because the **bins** start 0.

The resulting plots show us the histograms for both strata features. If we want to make sure that **pd.cut** did not create any **NaN** values, we can check this with the following code.

Algorithm 50 Test

```
# %%
has_nan = strata_cat_2.isna().any().any()
print("Does the DataFrame contain NaN values?", has_nan)
```

One problem we have not addressed, yet, is that we can split the housing dataset only with respect to one feature. If we wanted to combine two features, then we would have to use the following trick.

Algorithm 51 Jupyter Cell

```
# %%
sStrataCat = "Strata_Cat"
housing[sStrataCat] = strata_cat_1.astype(str) + "_" + strata_cat_2.astype(str)
housing[sStrataCat].value_counts().sort_index().plot.bar(grid = True)
```

Here we simply concatenate the strings of the labels of each strata feature, which serve as labels for the combined stata feature. The only problem is that some of the combined stratas have very view samples, which could lead to distortions in our prediction model or even to runtime errors.

We can avoid this by simply putting all underrepresented strata in a new stratum with the label **"Other"**. In the code below the integer **iMinCounts** defines the minimum amount of samples each strata needs to have without being merged with the **Other** stratum.

Algorithm 52 Jupyter Cell

```
# %%
iMinCounts = 100
housing_stratacat_counts = housing[sStrataCat].value_counts()
indices_of_small_housing_stratacat_counts = housing_stratacat_counts[
    housing_stratacat_counts < iMinCounts
].index
housing[sStrataCat] = housing[sStrataCat].apply(
    lambda x: 'Other' if x in indices_of_small_housing_stratacat_counts else x
)
housing[sStrataCat].value_counts().sort_index().plot.bar(grid = True)
plt.show()
```

Here **.apply** returns a copy of the original series **housing[sStrataCat]** , in which each value **x** is replaced by the return value of the lamda function. The resulting histogram looks much better now, after the small strata have vanished.

Now we that we have obtained the desired strata, we can finally split the data with the **stratify** argument. Since we will not need the strata category afterwards, we can revert the housing dataset back to its original form before applying the split.

Algorithm 53 Jupyter Cell

```
# %%
from sklearn.model_selection import train_test_split
housing_strata_category = housing[sStrataCat].copy()
housing = housing_original
strat_train_set, strat_test_set = train_test_split(
    housing, test_size = 0.15, stratify = housing_strata_category, random_state = 42
)
housing = strat_train_set.drop("SalePrice", axis=1)
housing_targets = strat_train_set["SalePrice"].copy()
```

5.3.3 Pipelines

Next we import the required pipeline modules that we will be using.

Algorithm 54 Jupyter Cell

```
# %%
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import OneHotEncoder
from sklearn.pipeline import make_pipeline
from sklearn.compose import ColumnTransformer
from sklearn.compose import make_column_selector
from sklearn.preprocessing import FunctionTransformer
from sklearn.pipeline import Pipeline
from sklearn.base import BaseEstimator, TransformerMixin
```

The last imports will be useful for implementing our own transformers, and thus create custom hyperparameters. For this purpose it is helpful to define a list of the column features we want to transform, and an inverted version of this list in form of a dictionary.

Algorithm 55 Jupyter Cell

```
# %%
list_trafo_columns = [
    "FullBath", "HalfBath", "BsmtFullBath", "BsmtHalfBath",
    "GrLivArea", "OverallQual",
    "Ranked_PavedDrive", "Ranked_GarageFinish", "Ranked_GarageQual", "Garage-
Cars", "GarageArea",
    "BedroomAbvGr", "GrLivArea",
    "Ranked_HeatingQC", "GrLivArea",
    "KitchenAbvGr", "BedroomAbvGr", "TotRmsAbvGrd",
]
inverse_list_trafo_columns = {
    value: index for index, value in enumerate(list_trafo_columns)
}
```

In this project our custom transformer **ColumnFormulaTransformer** should have the following properties.

- Ability to calculate ratios, where the numerator can be a sum or product of features. The denominator can be a column, where some values may be 0. In this case we need to be able to provide an alternative column for the denominator, where the values are guaranteed to be $\neq 0$.
- We want to make sure the the first terms or factors in the numerator are the most important ones, while the less important ones can be omitted depending on which value the hyperparameter **iTermCutoff** has. The important part here is that we will see how to implement hyperparameters in custom transformers, even though we will also see that it may not always be a good idea to engineer too many hyperparameters.
- For best practice reasons the **fit** and **transform** methods should make sure that the number of features is consistent (see **X.shape[1]** further below).
- The transformer should have a **get_feature_names_out** method. As we will see this controls how the lables of new features are named. In our case it will simply concatenate the prefix **"formula"** with "_" and the corresponding transformer name.
- For compatibility reasons the **fit** method should return **self** and have an input argument called **y**, even if we will not use it.
- The transformer class should be derived from the **TransformerMixin** class, which will create the **fit_transform** automatically. We will only have to implement **fit** and **transform** ourselves.
- It should be also derived from **BaseEstimator**, if we want to use automatic hyperparameter tuning. The tuning process must have access to the arguments of the class initializer method **__init__**. This is usually already the case as long as **__init__** does not include special Python syntax like ***args** and ****kwargs**.

We can achieve this with the following implementation. It may appear a bit overwhelming at first, but the previous explanations should make it clear.

Algorithm 56 Jupyter Cell

```
# %%
class ColumnFormulaTransformer(BaseEstimator, TransformerMixin):
    def __init__(self,
            sum = [], product = [], denominator = [], altdenominator = [],
            iTermCutoff = np.inf
        ):
        self.sum = sum
        self.product = product
        self.denominator = denominator
        self.altdenominator = altdenominator
        self.iTermCutoff = iTermCutoff
    def fit(self, X, y=None):
        self.n_features_in_ = X.shape[1]
        return self
    def transform(self, X):
        assert self.n_features_in_ == X.shape[1]
        #calculate nominator:
        numerator = np.zeros(X.shape[0])
        for id, col in enumerate(self.sum):
            if id >= self.iTermCutoff:
                break
            numerator += X[:,inverse_list_trafo_columns[col]]
        if self.product:
            prodnumerator = np.ones(X.shape[0])
            for id, col in enumerate(self.product):
                if id >= self.iTermCutoff:
                    break
                prodnumerator *= X[:,inverse_list_trafo_columns[col]]
            numerator += prodnumerator
        #calculate denominator:
        if self.denominator:
            denominator = X[:, inverse_list_trafo_columns[self.denominator[0]]]
            if self.altdenominator:
                altdenominator = \
                    X[:, inverse_list_trafo_columns[self.altdenominator[0]]]
                denominator[denominator == 0] = altdenominator[denominator == 0]
            result = numerator / denominator
        else:
            result = numerator
        return result.reshape(-1, 1) #convert result from 1D to 2D NumPy array
    def get_feature_names_out(self, names=None):
        return ["formula"]
```

Next we want to include this transformer in pipelines that first apply **SimpleImputer**, then our custom transformer and finally **StandardScaler**. Since we will need several of these pipelines (and since the internal parameters of those pipelines should not be mixed with each other), we want to implement a function that can create new instances of our desired pipeline design.

Algorithm 57 Jupyter Cell

```
# %%
def make_pipeline_with_formula(
        sum = [], product = [],
        denominator = [], altdenominator = [],
        iTermCutoff = np.inf
    ):
    return Pipeline([
        ("imputer", SimpleImputer(strategy="median")),
        (
            "colformula",
            ColumnFormulaTransformer(
                sum = sum, product = product,
                denominator = denominator, altdenominator = altdenominator,
                iTermCutoff = iTermCutoff
            )
        ),
        ("scaler", StandardScaler())
    ])
```

Since we will be using column transformers as well, it is very convenient to arrange the required 3-tuples in **ColumnTransformer_TupleList**. As you will see later, this list reflects our previous ideas for creating new features like e.g. the bedroom ratio.

Algorithm 58 Jupyter Cell

```
# %%
bathsum_list = ["FullBath", "HalfBath", "BsmtFullBath", "BsmtHalfBath"]
ColumnTransformer_TupleList = [
    ("bath", make_pipeline_with_formula(
            sum = bathsum_list
        ), list_trafo_columns
    ),
    ("areaquality", make_pipeline_with_formula(
            product = ["GrLivArea", "OverallQual"]
        ), list_trafo_columns
    ),
    ("garage", make_pipeline_with_formula(
            product = ["GarageCars", "GarageArea", "Ranked_GarageFinish",
                "Ranked_GarageQual", "Ranked_GarageCond", "Ranked_PavedDrive",
            ]
        ), list_trafo_columns
    ),
    ("bedroom", make_pipeline_with_formula(
            product = ["BedroomAbvGr"], denominator = ["GrLivArea"]
        ), list_trafo_columns
    ),
    ("roomquality", make_pipeline_with_formula(
            product = ["Ranked_HeatingQC", "TotRmsAbvGrd"]
        ), list_trafo_columns
    ),
    ("bath_kitchen", make_pipeline_with_formula(
            sum = bathsum_list,
            denominator = ["KitchenAbvGr"],
            altdenominator = ["TotRmsAbvGrd"]
        ), list_trafo_columns
    ),
    ("bath_bedroom", make_pipeline_with_formula(
            sum = bathsum_list,
            denominator = ["BedroomAbvGr"],
            altdenominator = ["TotRmsAbvGrd"]
        ), list_trafo_columns
    ),
]
```

We also have to set up simple pipelines like e.g. log-transformations for heavy-tailed features. One important transformation that we have not discussed, yet, is **OneHotEncoder**. It transforms categorical features to numbers, which can then be used by regressors. The important part is that it makes sure each categorical value gets its own feature.

One of the reason for this is that two categorical values belonging to the same numerical feature would have an order between them (e.g. 5 is greater than 3), even though there may be no sensible order between those categories at all. This would create artificial patterns in the training set.

Algorithm 59 Jupyter Cell

```
# %%
def safe_log(x):
    return np.log(np.where(x <= 0, 1e-10, x))
log_pipeline = make_pipeline(
    SimpleImputer(strategy = "median"),
    FunctionTransformer(safe_log, feature_names_out = "one-to-one"),
    StandardScaler()
)
cat_pipeline = make_pipeline(
    SimpleImputer(strategy="most_frequent"),
    OneHotEncoder(handle_unknown="ignore")
)
default_num_pipeline = make_pipeline(
    SimpleImputer(strategy = "median"),
    StandardScaler()
)
ColumnTransformer_TupleList.extend([
    ("log", log_pipeline, heavy_tailed_features),
    ("cat", cat_pipeline, make_column_selector(dtype_include = object)),
])
preprocessing = ColumnTransformer(
    ColumnTransformer_TupleList, remainder = default_num_pipeline
)
```

5.3.4 Prediction models

Instead of a linear regressor we will be using a so called random forest regressor, although for now you do not have to understand how it works.

Algorithm 60 Jupyter Cell

```
# %%
from sklearn.ensemble import RandomForestRegressor
from scipy.stats import uniform, randint
full_pipeline = Pipeline([
    ("preprocessing", preprocessing),
    ("random_forest", RandomForestRegressor(random_state=42)),
])
```

The hyperparameter tuning can be done in different ways. We chose **RandomizedSearchCV** for this purpose, because it allows us to tune many different configurations without trying out all of them. This means the results will be computed in a significantly shorter time.

In the code below we can see how the transformer names and the function argument names are used to define which hyperparameters are changed in the end. Their names are separated by "__". Of course the order of these names play a role, because they reflect the nested structures in which the hyperparameters can be found.

Algorithm 61 Jupyter Cell

```
# %%
from sklearn.model_selection import RandomizedSearchCV
param_distribs = {
    'preprocessing__garage__colformula__iTermCutoff':
        RandomTermCuttof(2,6),
    'preprocessing__bath__colformula__iTermCutoff':
        RandomTermCuttof(1,4),
    'preprocessing__bath_kitchen__colformula__iTermCutoff':
        RandomTermCuttof(1,4),
    'preprocessing__bath_bedroom__colformula__iTermCutoff':
        RandomTermCuttof(1,4),
}
rnd_search = RandomizedSearchCV(
    full_pipeline,
    param_distributions = param_distribs,
    n_iter=30,
    cv=5,
    scoring='neg_root_mean_squared_error',
    random_state=42,
    return_train_score=True,  # Critical for overfitting check
)
rnd_search.fit(housing, housing_targets)
```

Here the argument **cv = 5** refers to the number of subdivisions of the training set required for a method called **cross validation**. The training set is subdivided in k many subsets. Then one of them is used as a test set, and the entirety of the remaining $k-1$ subsets is used as train set. This allows us to estimate the overall performance of our prediction model.

Furthermore, **return_train_score** helps us to detect possible overfitting issues, because then we can also get the **mean_train_score** instead of just the test scores:

Algorithm 62 Jupyter Cell

```
# %%
results = pd.DataFrame(rnd_search.cv_results_)
results[['params', 'mean_train_score', 'mean_test_score']]
```

The output of this cell shows us that

Algorithm 63 Output (snippet)

```
   id params mean_train_score mean_test_score
0  {'preprocessing__bath_ -11654.712929 -29645.010378
1  {'preprocessing__bath__colformula__iTermCutoff...  -11736.464726 -30044.265372
2  {'preprocessing__bath__colformula__iTermCutoff...  -11586.796143 -29920.220671
3  {'preprocessing__bath__colformula__iTermCutoff...  -11638.117710 -29604.245397
```

the mean train score is systematically much higher than the mean test score (with a difference of approximately 20,000), indicating that our current hyperparameter-tuned model has excessive freedom, leading to overfitting.

We would have to go back and eliminate some of this freedoms. E.g. we could reduce the number of hyperparameters and see whether the performance of the prediction model changes as a result.

For the scope of this guide we will not go deeper into this topic, but the key takeaway is that hyperparameters should be carefully engineered.

Note that we can also examine the **mean_test_score** only, if we want to see further details.

Algorithm 64 Jupyter Cell

```
# %%
cv_rmse_scores = -rnd_search.cv_results_['mean_test_score']
rmse_summary = pd.Series(cv_rmse_scores).describe()
rmse_summary
```

In any way we can now get the prediction model with the best hyperparameters our script has found so far. Our main train set (from the train test split) helps us to estimate the RMSE of its prediction.

Algorithm 65 Jupyter Cell

```
# %%
from sklearn.metrics import root_mean_squared_error
final_model = rnd_search.best_estimator_
housing_predicted_prices = final_model.predict(housing_final_test)
tree_rmse = root_mean_squared_error(housing_targets_final_test, hous-
ing_predicted_prices)
tree_rmse
```

Another way to estimate values like the RMSE are **confidence intervals** that depend on the **confidence level**. E.g. if we obtain RMSE = 50,000 ± 3,000, then the confidence interval is spanning from 50,000 - 3,000 to 50,000 + 3,000.

A confidence level of 95% means that if we constructed these confidence intervals repeatedly in the same way, then 95% of them would contain the true RMSE.

The following code from [GeronColab] gives us an array storing the upper and lower bound of such a confidence interval for the squared RMSE.

Algorithm 66 Jupyter Cell

```
# %%
from scipy import stats
confidence = 0.95
squared_errors = (housing_predicted_prices -
np.array(housing_targets_final_test)) ** 2
np.sqrt(stats.t.interval(
        confidence, len(squared_errors) - 1, loc=squared_errors.mean(),
        scale=stats.sem(squared_errors)
    )
)
```

Finally, we can use the prediction model to predict the targets (i.e. the sale prices) of the dataset, where the targets are unknown.

Algorithm 67 Jupyter Cell

```
# %%
housing_predicted_prices = rnd_search.predict(housing_unknown)
submission = pd.DataFrame({
    'Id': housing_unknown['Id'],
    'SalePrice': housing_predicted_prices
})
submission.to_csv(sLocal_Folder_Path + '/submission.csv', index=False)
```

In this case the predicted values are saved as a file called **submission.csv**. This file can then be uploaded on Kaggle, where you can see your test results in form of the root mean squared error.

If this error is below 0.20, you have decent result for this competition. Below 0.15 is a solid result, and below 0.10 means you are really good. In our case we reach a score of about 0.14.

References

[Kaggle Housing] House Prices - Advanced Regression Techniques

[AnacondaInstall] https://github.com/ageron/handson-ml3/blob/main/INSTALL.md

[GeronColab] https://colab.research.google.com/github/ageron/handson-ml3/blob/main/index.ipynb#scrollTo=-KAqK1NXk8Eu 3, 12, 35

[1] https://github.com/ynaghibi/BlogsResources/blob/main/KagglDataC1.py 21

[HoML] Aurélien Géron (2019). *Hands-On Machine Learning with Scikit-Learn, Keras, and Tensorflow: Concepts, Tools, and Techniques to Build Intelligent Systems.* O'Reilly Media 2

[VanderPlas] Jake VanderPlas (2016). *Python Data Science Handbook* O'Reilly Media 12

[ScrumRef] Jeff Sutherland (2014). *Scrum: The Art of Doing Twice the Work in Half the Time.* Crown Currency 20