# Interpreting Data Description

The results show 8 numbers for each column in your original dataset. The first number, the ****count****, shows how many rows have non-missing values.

Missing values arise for many reasons. For example, the size of the 2nd bedroom wouldn't be collected when surveying a 1 bedroom house. We'll come back to the topic of missing data.

The second value is the ****mean****, which is the average. Under that, ****std**** is the standard deviation, which measures how numerically spread out the values are.

To interpret the ****min****, ****25%****, ****50%****, ****75%**** and ****max**** values, imagine sorting each column from lowest to highest value. The first (smallest) value is the min. If you go a quarter way through the list, you'll find a number that is bigger than 25% of the values and smaller than 75% of the values. That is the ****25%**** value (pronounced "25th percentile"). The 50th and 75th percentiles are defined analogously, and the ****max**** is the largest number.

# Selecting Data for Modeling[¶](https://www.kaggle.com/code/dansbecker/your-first-machine-learning-model" \l "Selecting-Data-for-Modeling" \t "https://www.kaggleusercontent.com/kf/126670599/eyJhbGciOiJkaXIiLCJlbmMiOiJBMTI4Q0JDLUhTMjU2In0..rt2_ikJDjaB5t6SeAHCMMw.m8mHrl-zZ5ql6rHtuN4IaDXySdi8f-wf6PnmJb2SEHNaJbHFvoO8EYH1imfy7hzzlitlKEmtVDL0PC_S3GQyFc3j2KBlI9QAwR99_quVcdFVvrZ5Vbr0DYBJn_n2MgJskRN-mJed4BR8xEpCmdHgqzCd5lESXgXS8bFS-VqGT75FZVtOBBuJcT3biVFUn59M6lb-GDnkepgcpsNjs4DjnmWJ3LXrsc9Z6O4KNO2kDJdBnchr8K3Ehs_P35z1iyntfwRA1e9s3o0G6yZLyHEdttZc1ShiMVMDBnH3o5iC4p9oxWB3mfN7p6tYMnEIGav9yecOOMpGyIpe4yQg1w4P-EERH53S8dbrlKJOJqxzxu5FSH6Z0VACJw3HNXtySfRS7Wr5Ci_dK2mWEGspjOQU7Z1teqzxCJMR1zsMoJbJACEv17JMYAobWE3DEqz2CHjG3fi6nMNa_XVLsL8tzWARVRFwyCbHL_e4AtBJOjePueclWEJuTlkHNvmuwQzjYnz5q6YE1kQ-A8JEESXHPqIrqsqUsKGNioJsIiHvI8Pho1_i7BtTRHexSzeXi8qKDcfkAmHBmVbrqF2TFv3EOhCyrXus_25Y932TE5DkVrUjSAyIyi86rl-26-B7tKB-5H_JEJrIYonGW2i7N8hiimNG35W1gRdjdFQpr3PXRFYWRXz2VfzBg7TatGy5CwYPQD0I.oeKPSABqTNWhzfEL57juhQ/_self)

Your dataset had too many variables to wrap your head around, or even to print out nicely. How can you pare down this overwhelming amount of data to something you can understand?

We'll start by picking a few variables using our intuition. Later courses will show you statistical techniques to automatically prioritize variables.

To choose variables/columns, we'll need to see a list of all columns in the dataset. That is done with the ****columns**** property of the DataFrame (the bottom line of code below).

In [1]:

import pandas as pd

melbourne\_file\_path = '../input/melbourne-housing-snapshot/melb\_data.csv'melbourne\_data = pd.read\_csv(melbourne\_file\_path) melbourne\_data.columns

Out[1]:

Index(['Suburb', 'Address', 'Rooms', 'Type', 'Price', 'Method', 'SellerG',

'Date', 'Distance', 'Postcode', 'Bedroom2', 'Bathroom', 'Car',

'Landsize', 'BuildingArea', 'YearBuilt', 'CouncilArea', 'Lattitude',

'Longtitude', 'Regionname', 'Propertycount'],

dtype='object')

In [2]:

*# The Melbourne data has some missing values (some houses for which some variables weren't recorded.)# We'll learn to handle missing values in a later tutorial. # Your Iowa data doesn't have missing values in the columns you use. # So we will take the simplest option for now, and drop houses from our data. # Don't worry about this much for now, though the code is:*

*# dropna drops missing values (think of na as "not available")*melbourne\_data = melbourne\_data.dropna(axis=0)

There are many ways to select a subset of your data. The [Pandas course](https://www.kaggle.com/learn/pandas) covers these in more depth, but we will focus on two approaches for now.

1. Dot notation, which we use to select the "prediction target"
2. Selecting with a column list, which we use to select the "features"

## Selecting The Prediction Target

You can pull out a variable with ****dot-notation****. This single column is stored in a ****Series****, which is broadly like a DataFrame with only a single column of data.

We'll use the dot notation to select the column we want to predict, which is called the ****prediction target****. By convention, the prediction target is called ****y****. So the code we need to save the house prices in the Melbourne data is

In [3]:

y = melbourne\_data.Price

# Choosing "Features"

The columns that are inputted into our model (and later used to make predictions) are called "features." In our case, those would be the columns used to determine the home price. Sometimes, you will use all columns except the target as features. Other times you'll be better off with fewer features.

For now, we'll build a model with only a few features. Later on you'll see how to iterate and compare models built with different features.

We select multiple features by providing a list of column names inside brackets. Each item in that list should be a string (with quotes).

Here is an example:

In [4]:

melbourne\_features = ['Rooms', 'Bathroom', 'Landsize', 'Lattitude', 'Longtitude']

By convention, this data is called ****X****.

In [5]:

X = melbourne\_data[melbourne\_features]

Let's quickly review the data we'll be using to predict house prices using the describe method and the head method, which shows the top few rows.

In [6]:

X.describe()

Out[6]:

|  | Rooms | Bathroom | Landsize | Lattitude | Longtitude |
| --- | --- | --- | --- | --- | --- |
| count | 6196.000000 | 6196.000000 | 6196.000000 | 6196.000000 | 6196.000000 |
| mean | 2.931407 | 1.576340 | 471.006940 | -37.807904 | 144.990201 |
| std | 0.971079 | 0.711362 | 897.449881 | 0.075850 | 0.099165 |
| min | 1.000000 | 1.000000 | 0.000000 | -38.164920 | 144.542370 |
| 25% | 2.000000 | 1.000000 | 152.000000 | -37.855438 | 144.926198 |
| 50% | 3.000000 | 1.000000 | 373.000000 | -37.802250 | 144.995800 |
| 75% | 4.000000 | 2.000000 | 628.000000 | -37.758200 | 145.052700 |
| max | 8.000000 | 8.000000 | 37000.000000 | -37.457090 | 145.526350 |

In [7]:

X.head()

Out[7]:

|  | Rooms | Bathroom | Landsize | Lattitude | Longtitude |
| --- | --- | --- | --- | --- | --- |
| 1 | 2 | 1.0 | 156.0 | -37.8079 | 144.9934 |
| 2 | 3 | 2.0 | 134.0 | -37.8093 | 144.9944 |
| 4 | 4 | 1.0 | 120.0 | -37.8072 | 144.9941 |
| 6 | 3 | 2.0 | 245.0 | -37.8024 | 144.9993 |
| 7 | 2 | 1.0 | 256.0 | -37.8060 | 144.9954 |

Visually checking your data with these commands is an important part of a data scientist's job. You'll frequently find surprises in the dataset that deserve further inspection.

# Building Your Model

You will use the ****scikit-learn**** library to create your models. When coding, this library is written as ****sklearn****, as you will see in the sample code. Scikit-learn is easily the most popular library for modeling the types of data typically stored in DataFrames.

The steps to building and using a model are:

* ****Define:**** What type of model will it be? A decision tree? Some other type of model? Some other parameters of the model type are specified too.
* ****Fit:**** Capture patterns from provided data. This is the heart of modeling.
* ****Predict:**** Just what it sounds like
* ****Evaluate****: Determine how accurate the model's predictions are.

Here is an example of defining a decision tree model with scikit-learn and fitting it with the features and target variable.

In [8]:

from sklearn.tree import DecisionTreeRegressor

*# Define model. Specify a number for random\_state to ensure same results each run*melbourne\_model = DecisionTreeRegressor(random\_state=1)

*# Fit model*melbourne\_model.fit(X, y)

Out[8]:

DecisionTreeRegressor(random\_state=1)

Many machine learning models allow some randomness in model training. Specifying a number for random\_state ensures you get the same results in each run. This is considered a good practice. You use any number, and model quality won't depend meaningfully on exactly what value you choose.

We now have a fitted model that we can use to make predictions.

In practice, you'll want to make predictions for new houses coming on the market rather than the houses we already have prices for. But we'll make predictions for the first few rows of the training data to see how the predict function works.

In [9]:

print("Making predictions for the following 5 houses:")print(X.head())print("The predictions are")print(melbourne\_model.predict(X.head()))

Making predictions for the following 5 houses:

Rooms Bathroom Landsize Lattitude Longtitude

1 2 1.0 156.0 -37.8079 144.9934

2 3 2.0 134.0 -37.8093 144.9944

4 4 1.0 120.0 -37.8072 144.9941

6 3 2.0 245.0 -37.8024 144.9993

7 2 1.0 256.0 -37.8060 144.9954

The predictions are

[1035000. 1465000. 1600000. 1876000. 1636000.]

# What is Model Validation[¶](https://www.kaggle.com/code/dansbecker/model-validation" \l "What-is-Model-Validation" \t "https://www.kaggleusercontent.com/kf/126670592/eyJhbGciOiJkaXIiLCJlbmMiOiJBMTI4Q0JDLUhTMjU2In0..Io6hTYIUQaCIRUtKPiRl5A.V7oupvQ7FUtUfSdx4-9BRiNLlesK0hf3Ypkq2V1Mx63VpEeG6lrSWrJJ4oQqaNR56P95RfapXQO-grmrS5Zro336ihY7gD15rEHqKFOQiWT0nhF9Pz3YspsguSHTYoY8n4f6X8TTcuBN3pEXan-mizLyL1NAhggUD6h1jv-cr0ORX7Ymy_mO1b9UnHLLn69rHiLBGdMsJlloQHiNJlSdDeK10-9yWGKkDaT4p9-kb-FsSRm_mls0Gm6ThZNzZjdjBoG-eO27O2dlCxFl-kg9G_s25JWbFvEwgXH8L2g4vIb_c9PuonghMVWknNYTsrw3jFMC1OT0WSjhyTB547e9p4wT-bb91c0v3BLjEbcnpSox6tAz7lSJiwDiMXnIOILfFPejz9A7F0SVBGLuWndZ-_6oveHPvY-Ldz08nQdP5V-HZ5F7z46mA1vxBZJKSJok4kT9s6X4IpdIaVBakNcaxuAtbBO-UDz7hep2yYjZHCM5e0VJ5Mg4JiZeZ_3Miv1KwG83HJvbDkhfmWBu9jzjfMHE_DDc9_qnZwLCH6j4_k-6TibVEyR1TetloFd_DxiZIf9amztRladFo6gGBEGUmqtwk5Z4wm1dDCItOcBnqb_3PE385kLcMCTv6l_aLYB7u9XvUrEy3kIWQBx22f8PPs9FZeWl2-Y_9F6QAc-aRLc.VU6GCzNLouVwB86bJB9Unw/_self)

You'll want to evaluate almost every model you ever build. In most (though not all) applications, the relevant measure of model quality is predictive accuracy. In other words, will the model's predictions be close to what actually happens.

Many people make a huge mistake when measuring predictive accuracy. They make predictions with their *training data* and compare those predictions to the target values in the *training data*. You'll see the problem with this approach and how to solve it in a moment, but let's think about how we'd do this first.

You'd first need to summarize the model quality into an understandable way. If you compare predicted and actual home values for 10,000 houses, you'll likely find mix of good and bad predictions. Looking through a list of 10,000 predicted and actual values would be pointless. We need to summarize this into a single metric.

There are many metrics for summarizing model quality, but we'll start with one called ****Mean Absolute Error**** (also called ****MAE****). Let's break down this metric starting with the last word, error.

The prediction error for each house is:

error=actual−predicted

So, if a house cost $150,000 and you predicted it would cost $100,000 the error is $50,000.

With the MAE metric, we take the absolute value of each error. This converts each error to a positive number. We then take the average of those absolute errors. This is our measure of model quality. In plain English, it can be said as

On average, our predictions are off by about X.

To calculate MAE, we first need a model. That is built in a hidden cell below, which you can review by clicking the code button.

Once we have a model, here is how we calculate the mean absolute error:

In [2]:

from sklearn.metrics import mean\_absolute\_error

predicted\_home\_prices = melbourne\_model.predict(X)mean\_absolute\_error(y, predicted\_home\_prices)

Out[2]:

434.71594577146544

# The Problem with "In-Sample" Scores

The measure we just computed can be called an "in-sample" score. We used a single "sample" of houses for both building the model and evaluating it. Here's why this is bad.

Imagine that, in the large real estate market, door color is unrelated to home price.

However, in the sample of data you used to build the model, all homes with green doors were very expensive. The model's job is to find patterns that predict home prices, so it will see this pattern, and it will always predict high prices for homes with green doors.

Since this pattern was derived from the training data, the model will appear accurate in the training data.

But if this pattern doesn't hold when the model sees new data, the model would be very inaccurate when used in practice.

Since models' practical value come from making predictions on new data, we measure performance on data that wasn't used to build the model. The most straightforward way to do this is to exclude some data from the model-building process, and then use those to test the model's accuracy on data it hasn't seen before. This data is called ****validation data****.

# Coding It

The scikit-learn library has a function train\_test\_split to break up the data into two pieces. We'll use some of that data as training data to fit the model, and we'll use the other data as validation data to calculate mean\_absolute\_error.

Here is the code:

In [3]:

from sklearn.model\_selection import train\_test\_split

*# split data into training and validation data, for both features and target# The split is based on a random number generator. Supplying a numeric value to# the random\_state argument guarantees we get the same split every time we# run this script.*train\_X, val\_X, train\_y, val\_y = train\_test\_split(X, y, random\_state = 0)*# Define model*melbourne\_model = DecisionTreeRegressor()*# Fit model*melbourne\_model.fit(train\_X, train\_y)

*# get predicted prices on validation data*val\_predictions = melbourne\_model.predict(val\_X)print(mean\_absolute\_error(val\_y, val\_predictions))

265806.91478373145

# Wow!

Your mean absolute error for the in-sample data was about 500 dollars. Out-of-sample it is more than 250,000 dollars.

This is the difference between a model that is almost exactly right, and one that is unusable for most practical purposes. As a point of reference, the average home value in the validation data is 1.1 million dollars. So the error in new data is about a quarter of the average home value.

There are many ways to improve this model, such as experimenting to find better features or different model types.