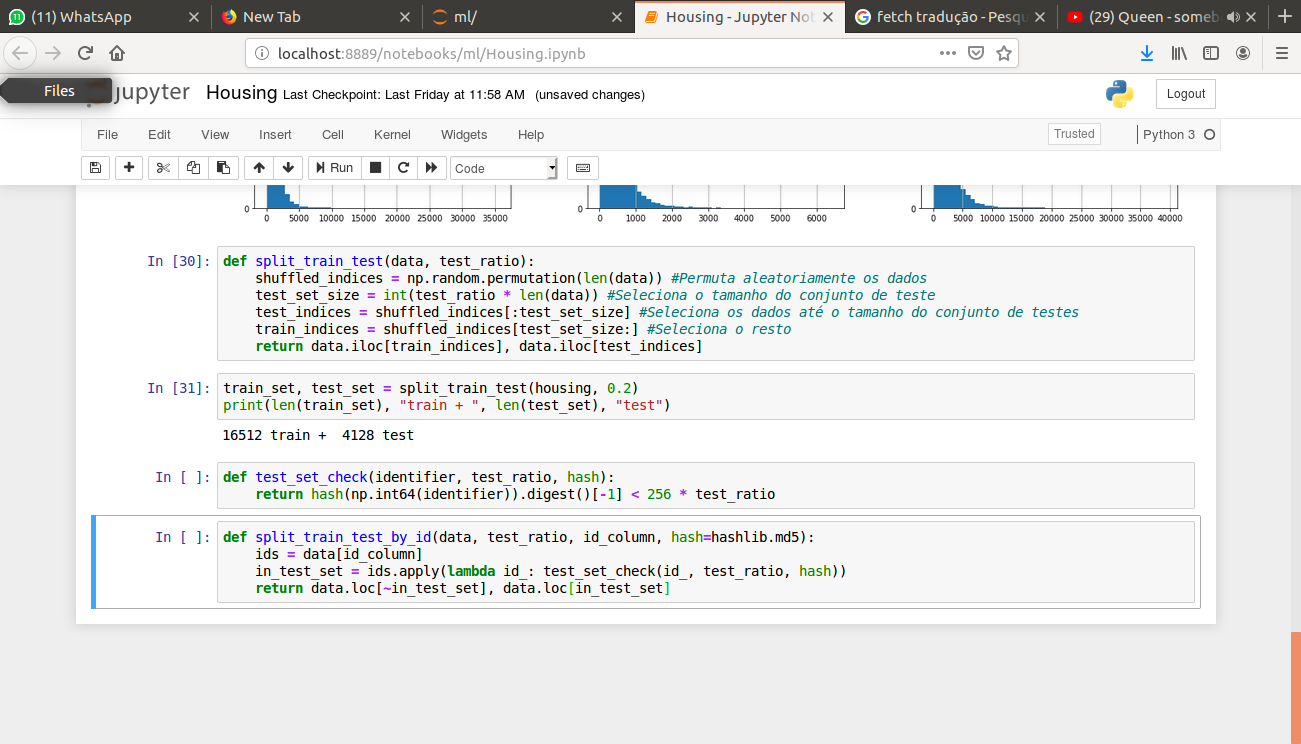
**Machine Learning**

**Criando um conjunto de teste**

Isto funciona, porém não é perfeito. Se rodarmos o programa novamente, ele irá gerar um diferente conjunto de teste. Ao longo do tempo, você, ou seu algoritmo de ML, vai poder observar todo o dataset, que é aquilo que você quer evitar.

Uma solução é salvar o conjunto de teste na primeira execução e depois carregá-lo em execuções subsequentes. Uma outra opção é setar a seed do gerador de números aleatórios (e.g., np.random.seed(42)) antes de chamar np.random.permutation(), assim, sempre irá gerar as mesmas permutações.

Mas ambas dessas soluções irão quebrar a próxima vez que você trazer um dataset atualizado. Uma solução comum é usar o identificador de casa instância para decidir se deve ou não ir pro test set (assumindo que cada instância tenha um um identificador imutável e único). Por exemplo, você pode computar uma parte de cada identificador, deixando apenas o último byte, e colocar esta instância no test set se este valor for menor ou igual a 51 (~20% de 256). Isto garante que o test set se manterá constante após multiplas execuções, mesmo que você atualize-o. O novo test set irá conter 20% das novas instâncias, mas não irá conter nenhuma instância que fora previamente utilizada no training set.

Infelizmente, o dataset de housing não apresenta uma coluna de identificação. A solução mais simples é usar o índice da linha como ID

*housing\_with\_id = housing.reset\_index() # Adiciona uma coluna de índice*

*train\_set, test\_set = split\_train\_test\_by\_id(housing\_with\_id, 0.2, "index")*

Se você utilizar o índice da linha como um identificador único, você necessita fazer com que os novos dados sejam adicionados no final do dataset, e que nenhuma linha seja deletada. Se isto não for possível, então você pode tentar usar as features mais estáveis para criar um identificador único. Por exemplo, a latitude e a longitude dos distritos são guarantidos de serem estáveis por milhões de anos, então você pode combiná-los com o ID.

*housing\_with\_id = housing.reset\_index() # Adiciona uma coluna de índice*

*housing\_with\_id["id"] = housing["longitude"]\*1000 + housing["latitude"]*

*train\_set, test\_set = split\_train\_test\_by\_id(housing\_with\_id, 0.2, "index")*

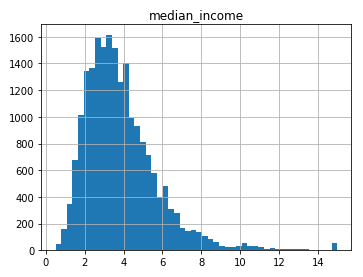
Scikit-Learn provê algumas funções para seprar datasets em múltiplos subconjuntos de várias formas. A função mais simples é train\_test\_split, a qual faz basicamente a mesma coisa que a função split\_train\_test definiu anteriormente, com alguns recursos adicionais. Primeiramente, existe um parâmetro random\_state que permite que você defina o gerador de seed aleatório como explicado previamente, e segundo, você pode passar múltiplos datasets com um número identico de linhas, e irá dividí-los com os mesmos índices (isto é muito útil, por exemplo, se você tem DataFrame separado para labels):

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

train\_set, test\_set = train\_test\_split(housing, test\_size=0.2, random\_state=42)

Até agora nós apenas consideramos métodos aleatórios de amostragem. Isso geralmente é tranquilo se o seu dataset é grande o suficiente (especialmente relativo ao número de atributos), mas se ele não é, você corre o risco de introduzir bias significantes. Por exemplo, quando uma empresa de pesquisas decide chamar 1000 pessoas para perguntá-las algumas questões, eles não simplesmente pegam 1000 pessoas aleatórias, eles tentam assegurar que essas 1000 pessoas são uma parcela representativa de toda a população. Por exemplo, a população america é composta de 51,3% de mulheres e 48,7% de homens, então uma pesquisa bem conduzida nos Estados UNidos tenta manter essa razão na amostragem: 513 mulheres e 487 homens. Isto é chamado *Stratified Sampling*: a população é dividida em subgrupos homogêneos chamados de stata, e o número exato de instâncias é amostrado para cada stratum de formar a garantir a representativa da população geral. Se eles usassem amostragens randômicas, teria uma chance de 12% de criar um test set com menos de 49% de mulheres ou mais de 54%. De qualquer mdo, os resultados apresentariam muitas bias.

Vamos supor que você conversou com experts que o disseram que a median income é um atributo muito importante para predizer median housing prices. Você pode querer assegurar que o test set é um representativo de várias categorias de incomes em todo o dataset. Já que o median income é um atributo numeral contínuo, você primeiramente deve criar um atributo de categoria de income. Vamos dar mais uma olhada no histograma de median income:



A maioria dos valores de renda média se concentram entre $20000 e $50000, mas algumas rendas medias vão bem mais além de $60000. É importante que haja um número suficiente de instâncias em seu dataset para cada stratum, senão a esimativa da importância do stratum pode ser prejudicada. Isso significa que você não deve ter muitos strata’s, mas cada stratum deve ser grande o suficiente. O próximo código cria um atributo de categoria de renda dividindo a renda média por 1.5 (para limitar o número de categorias de renda), arredondá-lo usando /ceil/ (para haver categorias discretas) e depois fundir todas as categorias maiores que 5 na categoria 5:

housing["income\_cat"] = np.ceil(housing["median\_income"] / 1.5)

housing["income\_cat"] = where(housing["income\_cat"] < 5, 5.0, inplace=True)

Agora estamos prontos para criar uma amostragem stratificada baseada na categoria de renda. Para isso, você pode usar a classe do Scikit-Learn StratifiedShuffleSplit.

Irei começar a escrever em inglês mesmo.

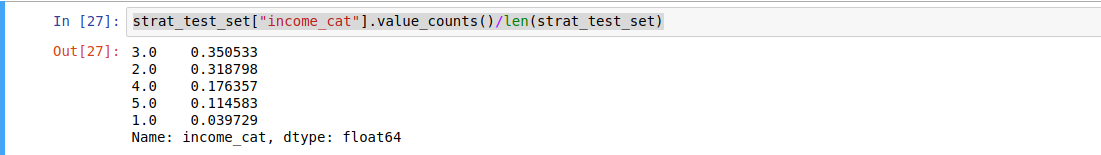
split = StratifiedShuffleSplit(n\_splits = 1, test\_size = 0.2, random\_state = 42)

for train\_inde, test\_index in split.split(housing, housing["income\_cat"]):

strat\_train\_set = housing.loc[train\_index]

strat\_test\_set = housing.loc[test\_index]

Let’s see if this worked as expected. You can start by looking at the income category proportions in the test set:



With similar code you can measure the income category proportions in the full dataset. Figure 2-10 (on the book) compares the income category proportions in the overall dataset, in the test set generated with stratified sampling and in test set generated using purely random sampling. As you can see, the test set generated using stratified sampling has income category proportions almost identical to those in the full dataset, whereas the test set generated using purely random sampling is quite skewed(distorcido).

Now you should remove the income\_cat attribute so the data is back to its original state:



We spent quite a bit of time on test set generation for a good reason: this is an often neglected but critical part of a Machine Learning project. Moreover, many of these ideas will be useful later when we discuss cross-validation. Now it’s time to move on to the next stage: exploring the data.

**Discover and Visualize the Data to Gain Insights**

So far you have only taken a quick glance at the data to get general understanding of the kind of data you are manipulating. Now the goal is to go a little bit more in depth.

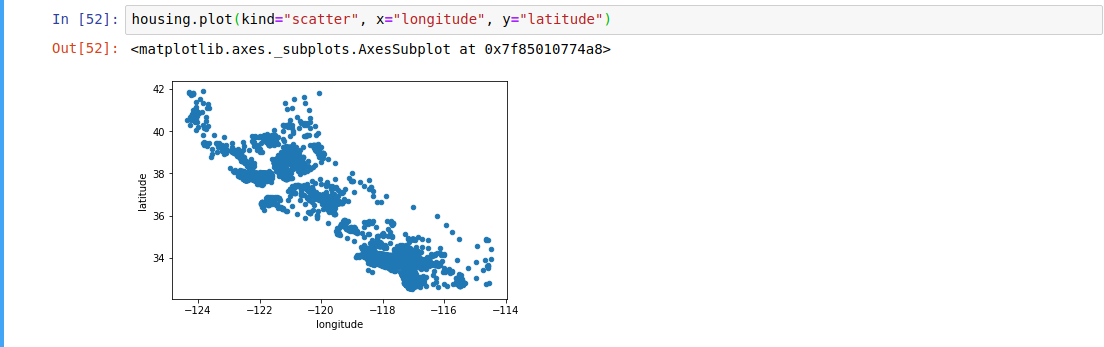
First, make sure you have put the test set aside and you are only exploring the training set. Also, if the training set is very large, you may wanto to sample an exploration set, to make manipulations easy and fast. In our case, the set is quite small so you can just work directly on the full set. Let’s create a copy so you can play with it without harming the training set:

housing = strat\_train\_set.copy()

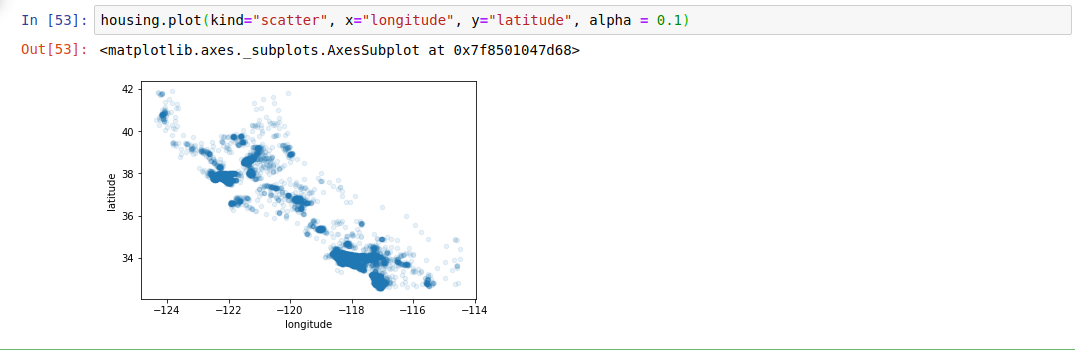
**Visualizing Geographical Data**

Since there is geographical information (latitude and longitude), it is a good idea to create a scatterplot of all districts to visualize the data:

housing.plot(kind=”scatter”, x=”longitude”, y=”latitude”)



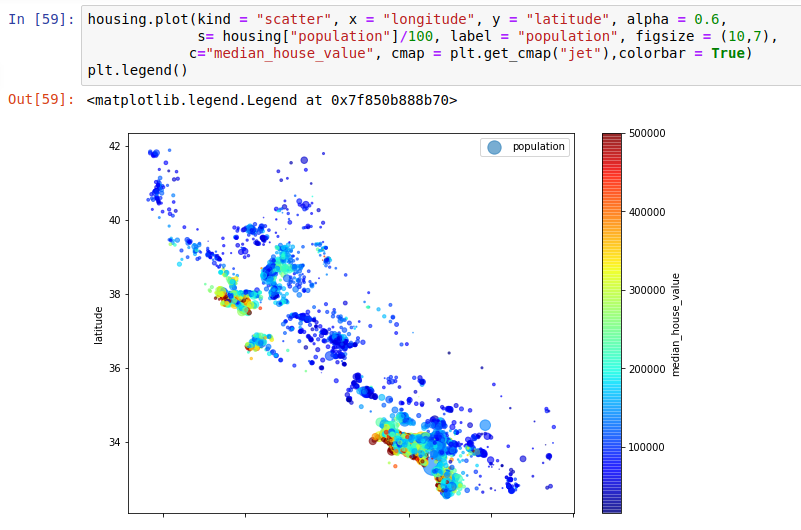
Ok, this looks like California all right, but other than that it is hard to see any particular pattern. Setting the alpha option to 0.1 makes it much easier to visualize the places where there is high density of points.



Now that’s much better: you can clearly see he high-density areas, namely the Bay Area and around Los Angeles and San Diego, plus a long line of fairly high density in the Central Valley, in particular around Sacramento and Fresno.

More generally, our brains are very good at spotting patterns on pictures, but you may need to play around visualization parameters to make the patterns stand out.

Now let’s look at the housing prices. The radius of each circle represents the district’s population (option s), and the color represents the price (option c). We will use a predefined color map (option cmap) called jet, which ranges from blue (low values) to red (high prices)

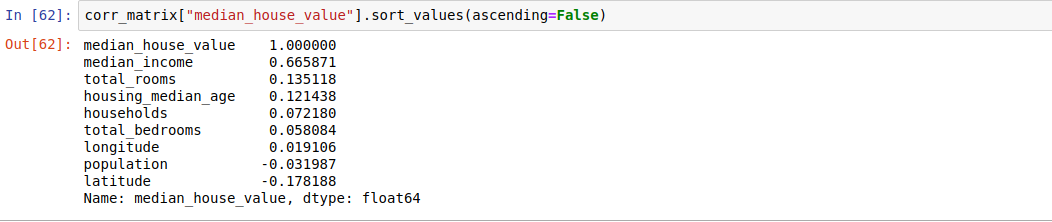


This image tells you that the housing prices are very much related to the location (e.g., close to the ocean) and to the population density, as you probably knew already. It will probably be useful to use a clustering algorithm to detect the main clusters, and add new features that measure the proximity to the cluster centers. The ocean proximity attribute may be useful as well, although in Northern California, the housing prices in coastal districts are not too high, so it’s not a simple rule.

**Looking for Correlations**

Since the dataset is not too large, you can easily compute the standard correlation coefficient (also called Pearson’s r) between every pair of attributes using the corr() method:

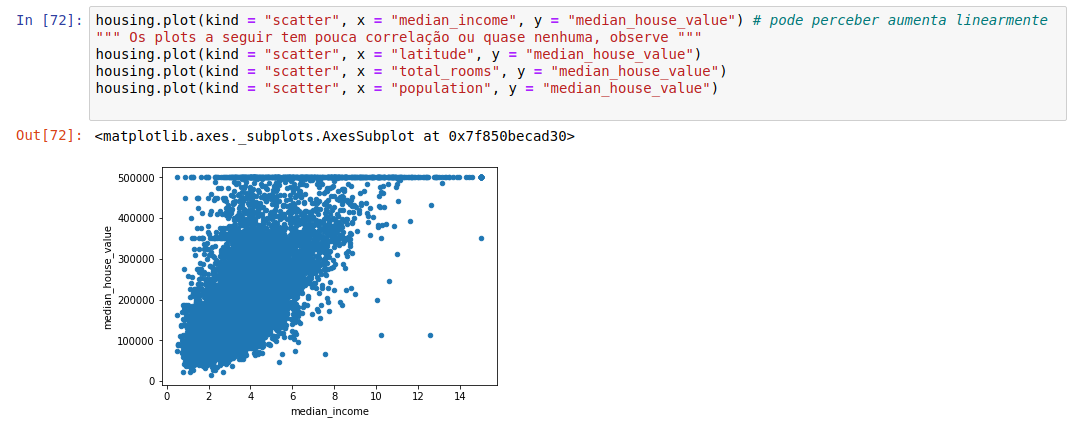
Now let’s look at how much each attribute correlates with the median house value:



The correlation coefficient ranges from -1 to 1. When it is close to 1, it means that there is a strong positive correlation; for example, the median house value tendsto go up when the median income goes up. When the coefficient is close to -1, it means that there is a strong negative correlation, you can see a small negative correlation between the latitude and the median house value (i.e, prices have a slight tendency to go down when you go north). Finally, coefficients close to zero mean that there is no linear correlation. Figure 2-14 (in the book) show various plots along with the correlation coefficent between their horizontal and vertical axes.

e.g = “exempli gratia”

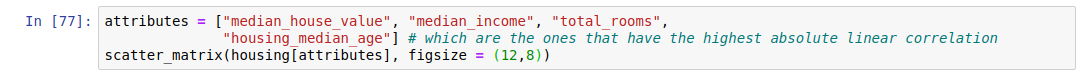
i.e = “In other words”

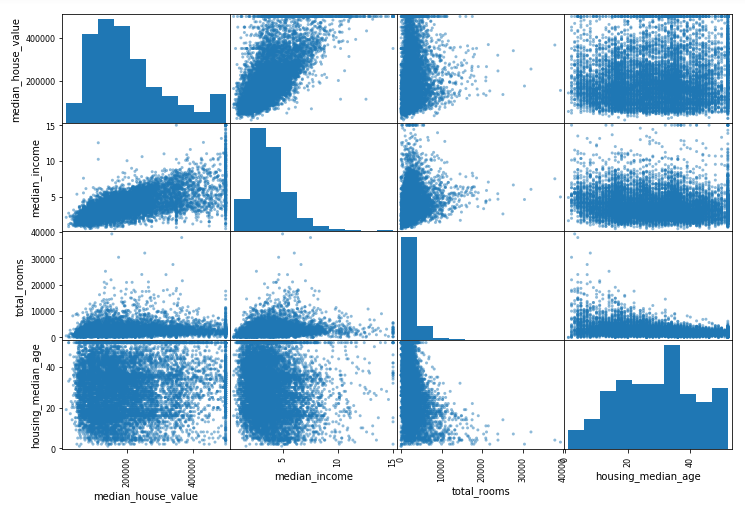


If we make a plot with the median\_house\_value in function of median\_income we can observe the linear pattern that when the median income increases, the median house value increases as well.

The correlation coefficient only measures linear correlations (“if x goes up, then y generally goes up/down”). It may completely miss out on nonlinear relationships (e.g “if x is close to zero then y generally goes up”). Note how all the plots of the bottom row have a correlation coefficient equal to zero despite the fact that their axes are clearly non independent: these are examples of nonlinear relationships. Also, the second row shows examples where the correlation coefficient is equal to -1 or -1; notice that this has nothing to do with the slope. For example, your height in inches has a correlation coefficient of 1 with your height in feet or nanometers.

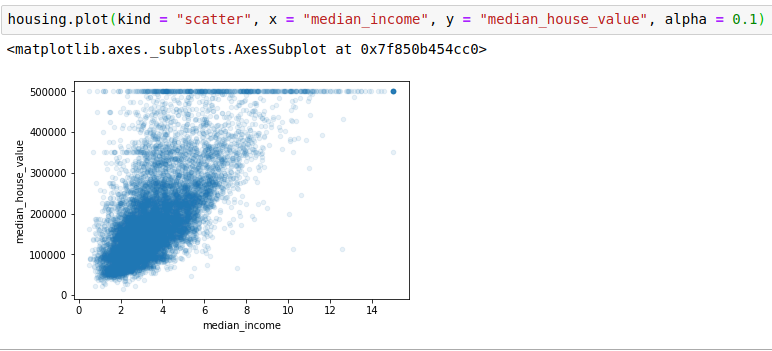
Another way to check for correlation between attributes is to use Panda’s scatter\_matrix function, which plots every numerical attribute against every other numerical attribute. Since there are now 11 numerical attributes, you wuold get 11² = 121 plots, which would not fit on a page, so let’s just focus on a few promising attributes that seem most correlated with the median housing value.





The main diagonal (top left to bottom right) would be full of straight lines if Pandas plotted each variable against itself, which would not be very useful. SO instead Pandas display a histogram of each attribute (other options are available see Panda’s documentation for more details).

The most promising attribute to predict the median house value is the median income, so let’s zoom on their correlation, scatter-plot.

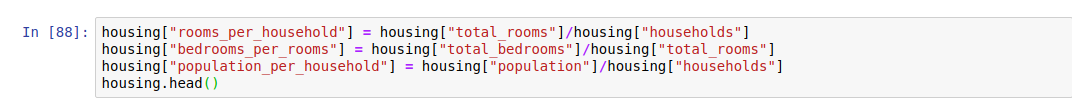


This plot reveals a few things. First, the correlation is indeed very strong; you can clearly see the upward trend and the points are not too dispersed. Second, the price cap that we noticed earlier is clearly visible as a horizontal line at $500,000. But other plot reveals other less obvious straight lines: a horizontal line at $450,000, another around $350,000, perhaps one around $280,000, and a few more bellow that. You may want to try removing the corresponding districts to prevent your algorithm from learning to reproduce these data quirks (peculiaridades).

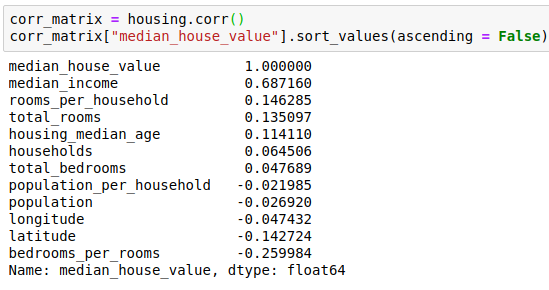
**Experimenting with Attribute Combinations**

Hopefully the previous sections gave you an idea of a few ways you can explore the data and gain insights. You identified a few data quirks that you may want to clean up before feeding the data to a Machine Learning Algorithm, and you found interesting correlations between attributes, in particular with the target attribute. You also noticed that some attributes have a tail-heavy algorithm, so you may want to transform them (e.g, by computing their logarithm). Of course, your mileage will vary considerably witch each project, but the general ideas are similar.

One last thing you may want to do before actually preparing the data for Machine Learning algorithms is to try out various attribute combinations. For example, the total numbers of rooms in a district is not very useful if you don’t know how many households (número de residências) there are. Wht you really want is the number of rooms per household. Similarly, the total number of bedrooms by itself is not very useful, you probably want to compare it to the number of rooms. And the population per household also seems like an interesting attribute combination to look at. Let’s create these new attributes.



And now let’s look at the correlation matrix again:



Hey, not bad! The new beedroms\_per\_room attribute is much more correlated with the median house value than the total number of rooms or bedrooms. Apparently houses with a lower bedroom/room ratio tend to be more expensive. The number of rooms per household is also more informative than the total number of rooms in a district - obviously the larger the houses, the more expensive they are.

This round of exploration does not have to be absolutely thorough (completo); the point is to start off on the right foot and quickly gain insights that will help you get a first reasonably good prototype. But this is an iterative process: once you get a prototype up and running, you can analyze its output to gain more insights and come back to this exploration step.

**Prepare the Data for Machine Learning Algorithms**

It’s time to prepare the data for your Machine Learning algorithms. Instead of just doing this manually, you should write functions to do that, for several good reasons:

1. This will allow you to reproduce these transformations easily on any dataset (e.g the next time you get a fresh dataset).
2. You will gradually build a library of transformation functions that you can reuse in future projects.
3. You can use these functions in your live system to transform the new data before feeding it to your algorithms.
4. This will make it possible for you to easily try various transformations and see which combination of transformations works best.

But first let’s revert to a clean training set (by copying strat\_train\_set once again, and let’s separe the predictors and the labels since we don’t necessarily want to apply the same transformations to the predictors and the target values (note that drop() creates a copy of the data and does not affect strat\_train\_set)