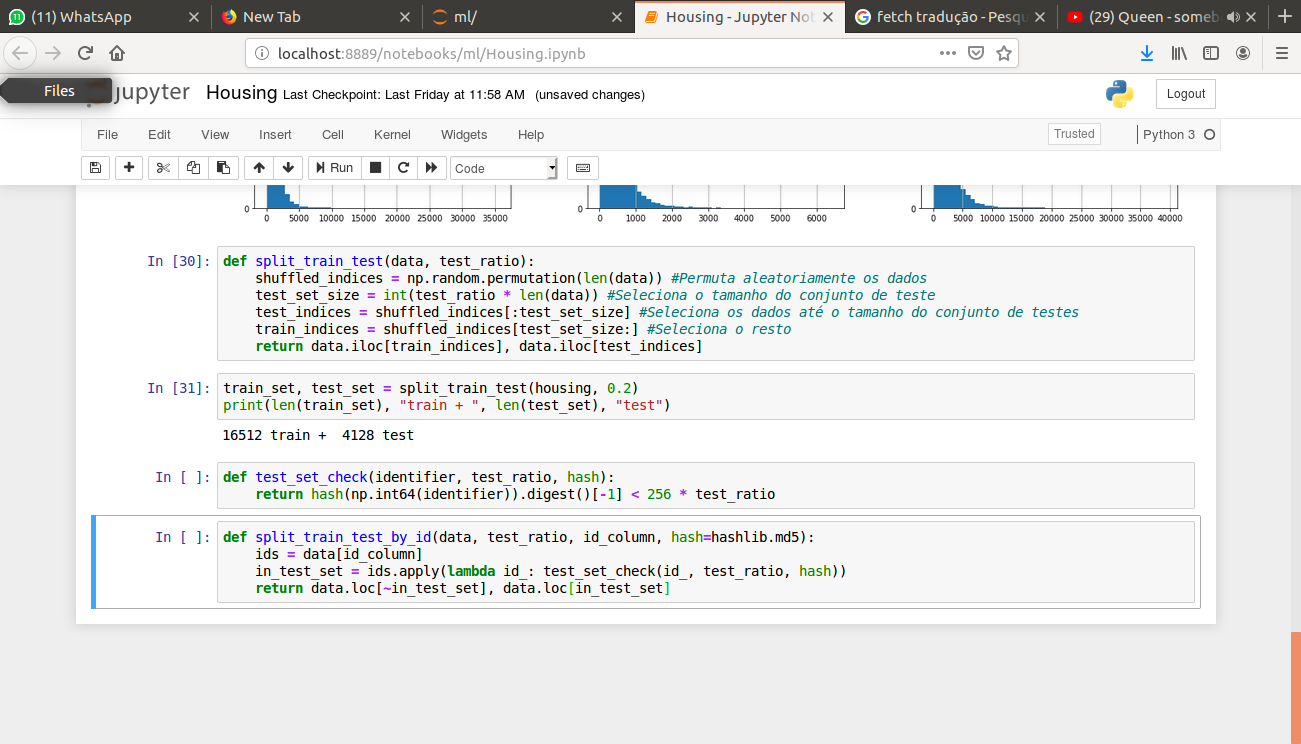
**Machine Learning**

**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")*

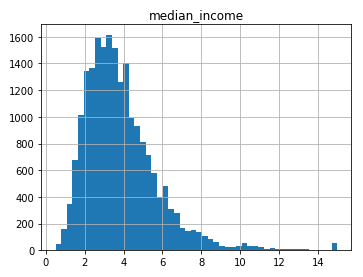
Scikit-Learn provê algumas funções para seprar datasets em múltiplos subconjuntos de várias formas. A função mais simples é train\_test\_split, a qual faz basicamente a mesma coisa que a função split\_train\_test definiu anteriormente, com alguns recursos adicionais. Primeiramente, existe um parâmetro 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.

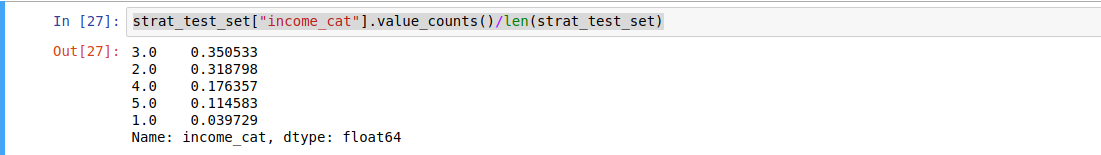
split = StratifiedShuffleSplit(n\_splits = 1, test\_size = 0.2, random\_state = 42)

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.

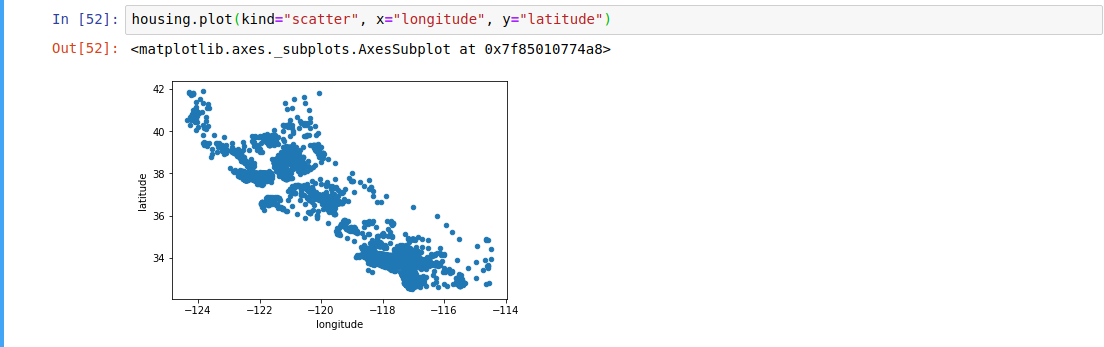
First, make sure you have put the test set aside and you are only exploring the training set. Also, if the training set is very large, you may wanto to sample an exploration set, to make manipulations easy and fast. In our case, the set is quite small so you can just work directly on the full set. Let’s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

housing = strat\_train\_set.copy()

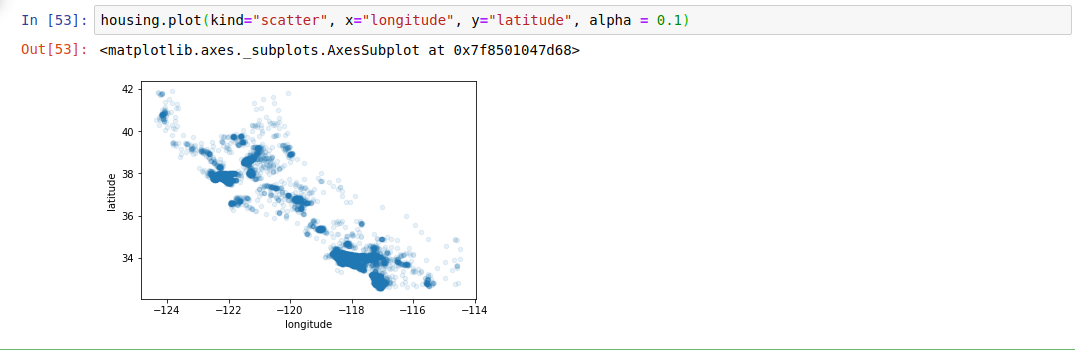
**Visualizing Geographical Data**

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”)



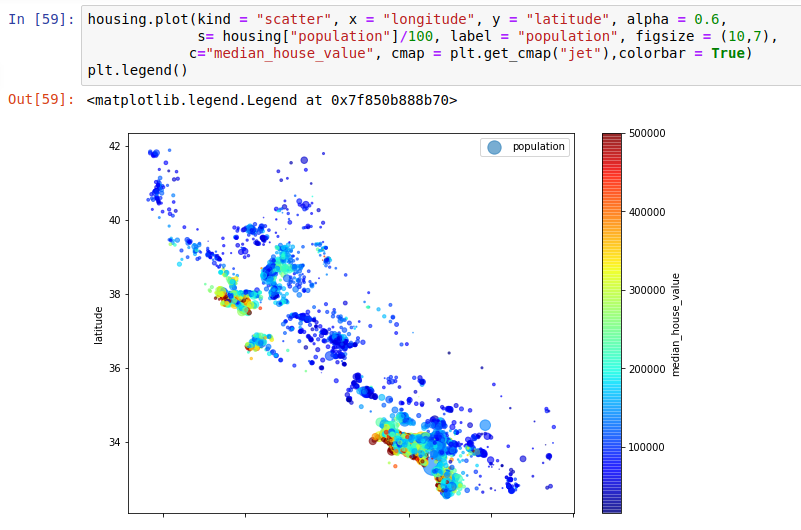
Ok, this looks like California all right, but other than that it is hard to see any particular pattern. Setting the alpha option to 0.1 makes it much easier to visualize the places where there is high density of points.



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)

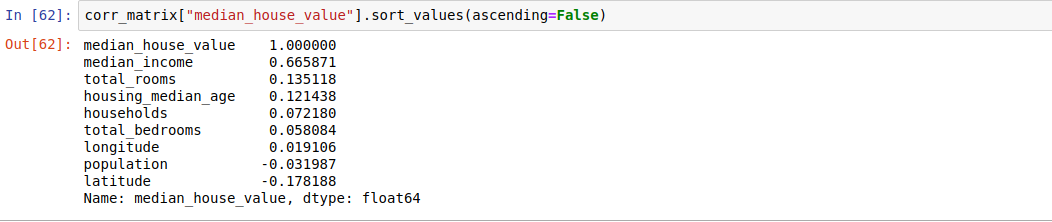


This image tells you that the housing prices are very much related to the location (e.g., close to the ocean) and to the population density, as you probably knew already. It will probably be useful to use a clustering algorithm to detect the main clusters, and add new features that measure the proximity 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:

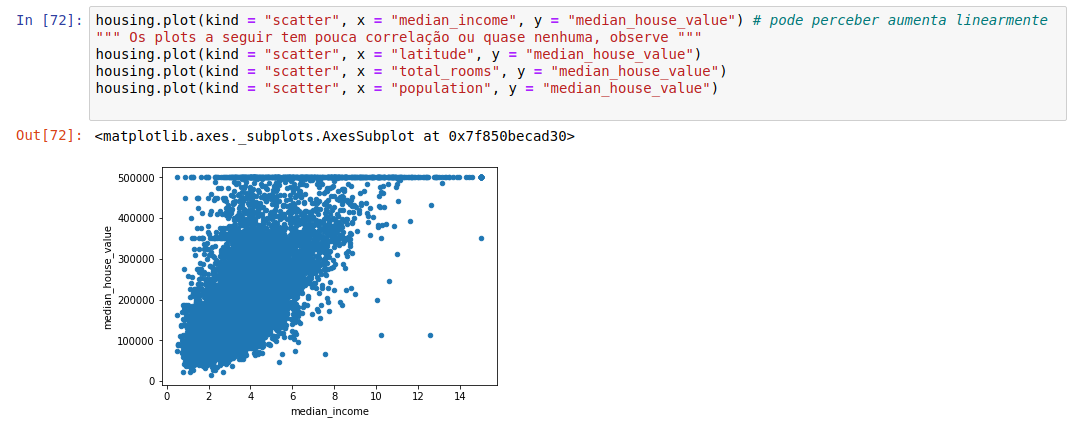
Now let’s look at how much each attribute correlates with the median house value:



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”

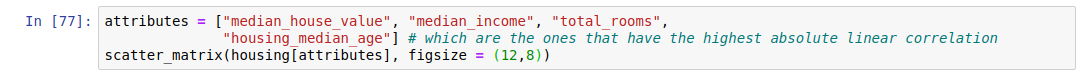
i.e = “In other words”

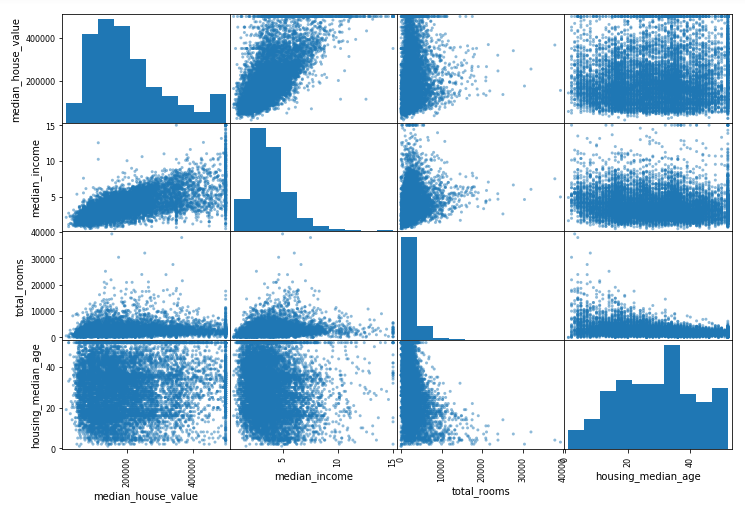


If we make a plot with the median\_house\_value in function of median\_income we can observe the linear pattern that when the median income increases, the median house value increases as well.

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.

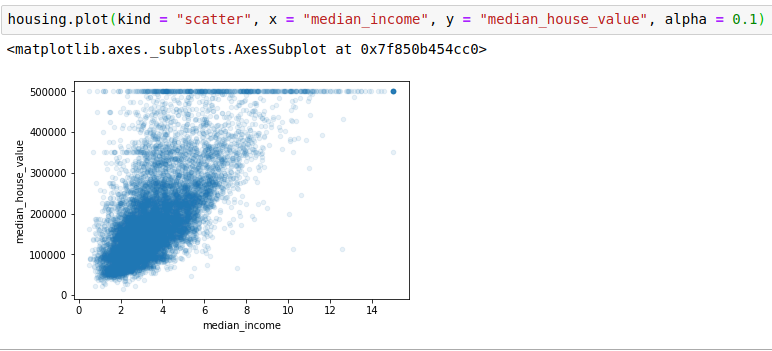
Another way to check for correlation between attributes is to use Panda’s scatter\_matrix function, which plots every numerical attribute against every other numerical attribute. Since there are now 11 numerical attributes, you wuold get 11² = 121 plots, which would not fit on a page, so let’s just focus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

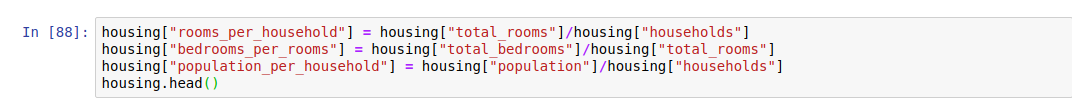


This plot reveals a few things. First, the correlation is indeed very strong; you can clearly see the upward trend and the points are not too dispersed. Second, the price cap that we noticed earlier is clearly visible as a horizontal line at $500,000. But other plot reveals other less obvious straight 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).

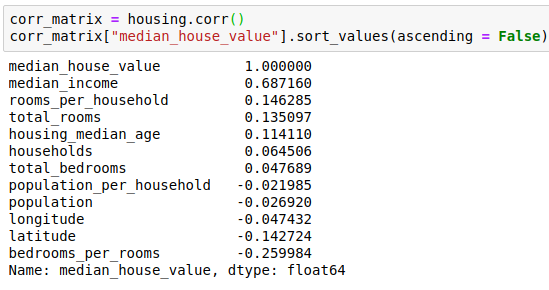
**Experimenting with Attribute Combinations**

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)

**Data Cleaning**

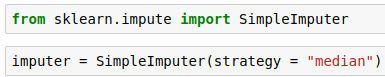
Most Machine Learning algorithms cannot work with missing feature, so let’s create a few functions to take care of them. You noticed earlier that the total\_bedrooms attribute has some missing values, so let’s fix this. You have three options:

1. Get rid of the corresponding districts
2. Get rid of the whole attribute
3. Set the values to some value (zero, the mean, he median, etc.)

You can accomplish these easily using DataFrame’s dropna(), drop(), and fillna().

If you choose option3, you should compute the median value on the training set, and use it to fill the missing values in the training set, but also don’t forget to save the median value that you have computed. You’ll need it later to replace missing values in test set when you want to evaluate your system, and also once the system goes live to replace missing values in new data.

Scikit-Learn provides a handy class to take care of missing values: Imputer. Here is how to use it. First, you need to create an Imputer instance, specifying that you want to replace each attribute’s missing values with the median of that attribute:



Since the median can only be computed on numerical attributes, we need to create a copy of the data without the TEXT ATTRIBUTE ocean\_proximity

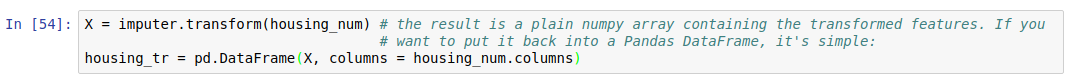
housing\_num = housing.drop("ocean\_proximity", axis = 1)

Now you can fit the imputer instance to the training data using the fit() method

imputer.fit(housing\_num)

The imputer has simply computed the median of each attribute and stored the result in its statistics\_ instance variable. Only the total\_bedrooms attribute had missing values, but we cannot be sure that there won’t be any missing values in new data after the system goes live. So it’s safer to apply the imputer to all the numerical attributes.

Now you can use this “trained” imputer to transform the training set by replacing missing values by the learned medians



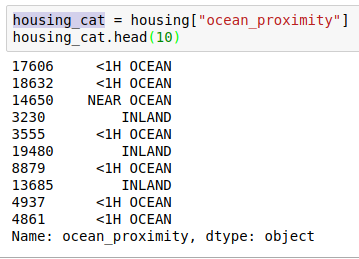
**Scikit-Learn Design**

Scikit-Learn’s API (Application Programming Interface) is remarkably well designed. The main design principles are:

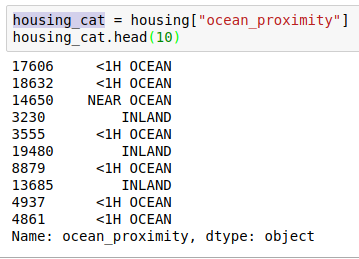
1. **Consistency**: All objects share a consistent and simple interface:
   1. **Estimators**. Any object that can estimate some parameters based on a dataset is called an estimator (e.g, an imputer is an estimator). The estimation itself is performed by the fit() method, and it takes only a dataset as a parameter (or two for supervised learning algorithms; the second dataset contains the labels). Any other parameter need to guide the esimation process is considered a hyperparameter (such as an imputer’s strategy), and it must be set as an instance variable (generaly via a constructor parameter)
   2. **Transformers**: Some estimators (such as an imputer) can also transform the dataset; these are called transformers. Once again, the API is quite simple: the transformation is perfomed by the transform() method with the dataset to transform as a parameter. It returns the transformed dataset. This transformation generally relies on the learned parameters, as in the case for an imputer, as is the case for an imputer. All transformers also have a convenience method called fit\_transform() that is equivalent to calling fit() and then transform() (but sometimes fit\_transform() is optimized and runs much faster).
   3. **Predictors**. Finally, some estimators are capable of making predictions given a dataset; they are called predictor. For example, the LinearRegression model in the previous chapter was a predictor: it predicted life satisfaction given a country’s GDP per capita. A predictor has a predict() method that takes a dataset of new instances and returns a dataset of corresponding predictions. It also has a score() method that measures the quality of the predictions given a test set (and the corresponding labels in the case of supervised learning algorithms - some predictors also provide methods to measure the confidence of their predictions).
2. **Inspection.** All the estimator’s hyperparameters are accessible directly via public instance variables (e.g., imputer.strategy), and all the estimator’s learned parameters are also accessible via public instance variables with an underscore suffix (e.g, imputer.statistics\_).
3. **Nonproliferation of classes.** Datasets are represented as NumPy arrays of SciPy sparse matrices, instead of homemade classes. Hyperparameters are just regular Python strings or numbers.
4. **Composition.** Existing building blocks are reused as much as possible. For example, it is easy to create a Pipeline estimator from an arbitraty sequence of transformers followed by an final estimator, as we will se.
5. **Sensible defaults.** Scikit-Learn provides reasonable default values for most parameters, making it easy to create a baseline working system quickly.

**Handling Text and Categorical Attributes**

Earlier we left out the categorical attribute ocean\_proximity because it is a text attribute so we cannot compute its median:



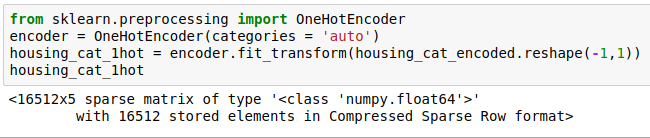
Most Machine Learning algorithms prefer to work with numbers anyway, so let’s convert these categories from text to numbers. For this, we can use Pandas’ factorize() method which maps each category to a different integer:



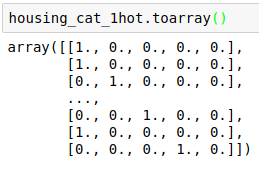
This is better: housing\_cat\_encoded is now purely numerical. The factorize() method also returns the list of categories(“<1H OCEAN” was mapped to 0, “NEAR OCEAN” was mapped to 1, etc.)

One issue with this representation is that ML algorithms will assume that two nearby values are more similar than two distant values. Obviously that is not the case (for example, categories 0 and 4 are more similar than categories 0 and 2). To fix this issue, a common solution is to create one binary attribute per category: one attribute equal to 1 when the category is “<1H OCEAN” (and 0 otherwise), another attribute equal to 1 when the category is “NEAR OCEAN” (and 0 otherwise) and so on. This is called one-hot encoding, because only one attribute will be equal to 1 (hot), while the others will be 0 (cold).

Scikit-Learn provides a OneHotEncoder encoder to convert integer categorical values into one-hot vectors. Let’s encode the categories as one-hot vectors:



Note that fit\_transofrm(0 expects a 2d array, but housing\_cat\_encoded is a 1D array, so we need to reshape it (NumPy’s reshape() function allows one dimension to be -1, which means “unspecified”: the value is inferred from the length of the array and the remaining dimensions). Also notice that the output is a SciPy sparse matrix, instead of a NumPy array. This is very useful when you have categorical attributes with thousands of categories. After one-hot encoding we get a matrix with thousands of columns, and the matrix is full of zeros except for a single 1 per row. Using up tons of memory mostly to sore zeros would be very wasteful, so instead a sparse matrix only sores the location of the nonzero elements. You can use it mostly like a normal 2D array, but if you really want yo convert it to a (dense) NumPy array, just call toarray() method.

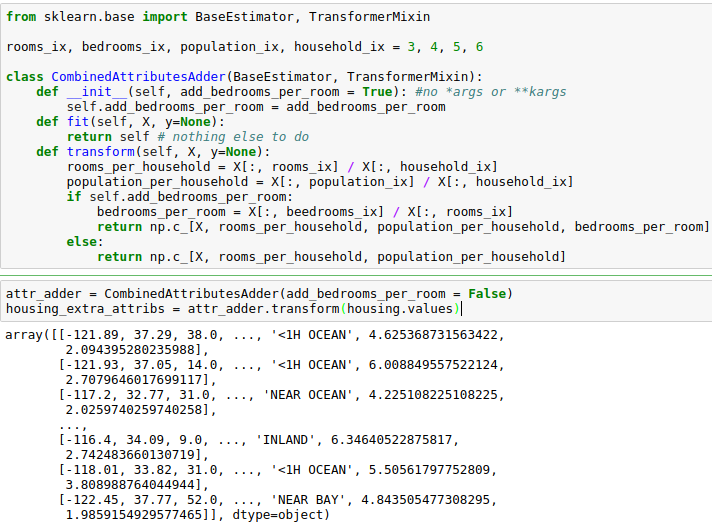


You can apply both transformations( from text categories to integer categories, then from integer cattegories to one-hot vectors) in one shot using the CategoricalEncoder class. It’s not pat of Scikit-Learn 0.19.0 and earliers, but it will be added shortly.

**Custom Transformers**

Although Scikit-Learn provides many useful transformers, you will need to write your own for tasks such as custom cleanup operations or combining specific attributes. You will want your transformer to work seamlessly with Scikit-Learn functionalities (such as pipelines), and since sk.learn relies on duck typing (not inheritance), all you need is to create a class and implement three methods: fit() (returning self), transform(), and fit\_transform(). You can get the last one for free by simply adding TransformerMixin as a base class. Also, if you add BaseEstimator as a base class (and avoid \*args and \*\*kargs in your constructor) you will get two extra methods (get\_params() and set\_params()) that will be useful of automatic hyperparameter tuning. For example, here is a small transformer class that adds the combined attributes we discussed earlier:

In this example the transformer has one hyperparameter, add\_bedrooms\_per\_room, set to True by default (it is often helpful to provide sensible defaults). This hyper-parameter will allow you to easily find out whether adding this attribute helps the Machine Learning algorithms or not. More generally, you can add a hyperparameter to get any data preparation step that you are not 100% sure about. The more you automate these data preparation steps, the more combinations you can automatically try out, making it much more likely to find a great combination (and saving you a lot of time).



**Feature Scaling**

One of the most important transformations you need to apply to your data is feature scaling. With few exceptions, Machine Learning algorithms don’t perform well when the input numerical attributes have very different scales. This is the case for the housing data: the total number of rooms rangers from about 6 to 39,320, while the median incomes only range from 0 to 15. Note that scaling the target values is generally not required.

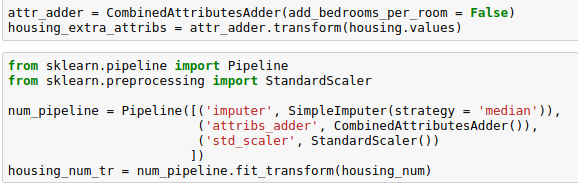
There are two common ways to get all attributes to have the same scale: min-max scaling and standardization.

Min-max scaling (many people call this normalization) is quite simple: values are shifted and re-scaled so that they end up ranging from 0 to 1. We do this by subtracting the min value and diving by the max minus the min. Sckit-Learn provides a transformer called MinMaxScaler for this. It has a feature\_range hyperparameter that lets you change the range if you don’t want 0-1 for some reason.

Standardization is quite different: first it subtracts the mean value (so strandardized values always have a zero mean), and then it divides by the variance so that the resulting distribution has unit variance. Unlike min-max scaling, standardization does not bound values to a specific range, which may be a problem for some algorithms (e.g, neural networks often expect an input vale ranging from 0 to 1). However, standardization is musch less affected by outliers. For example, suppose a district had a median income equal to 100 (by mistake). Min-max scalling would then crush all the other values from 0-15 to 0-0.15, whereas standardization would not be much affected. Scikit-Learn provides a transformer called StandardScaler for standardization.

**Transformation Pipelines**

As you can see, there are many data transformation steps that need to be executed in the right order. Fortunately, Scikit-Learn provides the Pipeline class to help with such sequences of transformations. Here is a small pipelines for the numerical attributes:

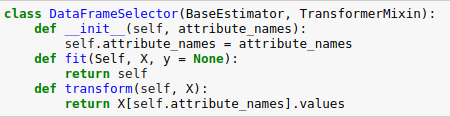


The pipeline constructor takes a list of name/estimator pairs defining a sequence of steps. All but the last estimator must be transformers (i.e, they must have a fit\_transform() method). The names can be anything you like (as long ad they don’t contain double underscores “\_\_”).

When you call the pipeline’s fit() method, it calls fit\_transform() sequentially on all transformers, passing the output of each call as the parameter to the next call, until it reaches the final estimator, for which it jsut calls the fit() method.

The pipeline exposes the same methods as the final estimator. In this example, the last estimator is a StandardScaler, which is a transformer, so the pipeline has a transform() method that applies all the transforms to the data in sequence (it also has a fit\_transform method that we could have used instead of calling fit() and then transform().

Now it would be nice if we could feed a Pandas DataFrame contaiing non-numerical columns directly into our pipeline, instead of having to first manually exctract the numerical columns into a Numpy array. There is nothing in Scikit-Learn to handle Pandas DataFrames, but we can write a custom transformer for this task.



Our DataFrameSelector will transform the data by selecting the desired attributes, dropping the rest, and converting the resulting DataFrame to a Numpy array. With this, you can easily write a pipeline that will talke a Pandas DataFrame and handle only the numerical values: the pipline wod just start with a DataFrameSelector to pick only the numerical attributes, followed by the other preprocessing steps we discussed earlier. And you can just as easily write another pipeline for the categorical attributes as well by simple selecting the categorical attributes using a DataFrameSelector and then applying a CategoricalEnconder