In 2006, Geoffrey Hinton(father of machine learning) et al. published a paper1 showing how to train a deep neural network capable of recognizing handwritten digits with state-of-the-art precision Tensorflow(developed by google) is a more complex library to perform distributed numerical computations.It was open sourced in November 2015, and version 2.0 was released in September 2019Keras is a high-level Deep Learning API that makes it very simple to train and run neural networks-Machine Learning is great for:Problems for which existing solutions require a lot of fine-tuning or long lists of rules: one Machine Learning algorithm can often simplify code and perform better than the traditional approach.Complex problems for which using a traditional approach yields no good solution: the best Machine Learning techniques can perhaps find a solution.Fluctuating environments: a Machine Learning system can adapt to new data.Getting insights about complex problems and large amounts of data.There are so many different types of Machine Learning systems that it is useful to classify them in broad categories, based on the following criteria: Whether or not they are trained with HUMAN SUPERVISION (supervised, unsupervised, semisupervised, and Reinforcement Learning) Whether or not they can learn INCREMENTALLY ON THE FLY (online versus batch learning) Whether they work by simply comparing new data points to known data points, or instead by detecting patterns in the training data and building a predictive model, much like scientists do (INSTANCE-BASED versus MODEL-BASED learning)LET’S LOOK AT EACH OF THESE CRITERIA A BIT MORE CLOSELY. Supervised/Unsupervised Learning Machine Learning systems can be classified according to the amount and type of supervision they get during training. There are four major categories: supervised learning, unsupervised learning, semisupervised learning, and Reinforcement Learning. Supervised learning In supervised learning, the training set you feed to the algorithm includes the desired solutions, called labels A typical supervised learning task is CLASSIFICATION. The spam filter is a good example of this: it is trained with many example emails along with their class (spam or ham), and it must learn how to classify new emails. Another typical task is to predict a target numeric value, such as the price of a car, given a set of features (mileage, age, brand, etc.) called predictors. This sort of task is called regression (Figure 1-6).1 To train the system, you need to give it many examples of cars, including both their predictors and their labels (i.e., their prices). NOTE In Machine Learning an ATTRIBUTE IS A DATA TYPE (E.G., “MILEAGE”), while a FEATURE HAS SEVERAL MEANINGS, DEPENDING ON THE CONTEXT, BUT GENERALLY MEANS AN ATTRIBUTE PLUS ITS VALUE (E.G., “MILEAGE = 15,000”). MANY PEOPLE USE THE WORDS ATTRIBUTE AND FEATURE INTERCHANGEABLY. Note that some regression algorithms can be used for CLASSIFICATION as well, and vice versa. For example, Logistic REGRESSION IS COMMONLY USED FOR CLASSIFICATION, as it can output a value that corresponds to the probability of belonging to a given class (e.g., 20% chance of being spam). Here are some of the most important supervised learning algorithms (covered in this book): k-Nearest Neighbors Linear Regression Logistic Regression Support Vector Machines (SVMs) Decision Trees and Random Forests Neural networks2 Unsupervised learning: In unsupervised learning, as you might guess, the training data is UNLABELED (FIGURE 1-7). THE SYSTEM TRIES TO LEARN WITHOUT A TEACHER. Here are some of the most important unsupervised learning algorithms (most of these are covered in Chapters 8 and 9):,Clustering,K-Means,DBSCAN,Hierarchical Cluster Analysis (HCA),Anomaly detection and novelty detection,One-class SVM,Isolation Forest,Visualization and dimensionality reduction,Principal Component Analysis (PCA),Kernel PCA,Locally Linear Embedding (LLE),t-Distributed Stochastic Neighbor Embedding (t-SNE),Association rule learning,Apriori,Eclat For example, say you have a lot of data about your blog’s visitors. You may want to run a CLUSTERING ALGORITHM to try to detect groups of similar visitors (Figure 1-8). At no point do you tell the algorithm which group a visitor belongs to: it finds those connections without your help. For example, it might notice that 40% of your visitors are males who love comic books and generally read your blog in the evening, while 20% are young sci-fi lovers who visit during the weekends. IF YOU USE A HIERARCHICAL CLUSTERING ALGORITHM, IT MAY ALSO SUBDIVIDE EACH GROUP INTO SMALLER GROUPS. THIS MAY HELP YOU TARGET YOUR POSTS FOR EACH GROUP. VISUALIZATION ALGORITHMS ARE ALSO GOOD EXAMPLES OF UNSUPERVISED LEARNING ALGORITHMS: you feed them a lot of complex and UNLABELED DATA, and they output a 2D or 3D representation of your data that can easily be plotted (Figure 1-9). These algorithms try to preserve as much structure as they can (e.g., trying to keep SEPARATE CLUSTERS in the input space from overlapping in the visualization) so that you can understand how the data is organized and perhaps identify unsuspected patterns. A related task is DIMENSIONALITY REDUCTION, IN WHICH THE GOAL IS TO SIMPLIFY THE DATA WITHOUT LOSING TOO MUCH INFORMATION. ONE WAY TO DO THIS IS TO MERGE SEVERAL CORRELATED FEATURES INTO ONE. FOR EXAMPLE, A CAR’S MILEAGE MAY BE STRONGLY CORRELATED WITH ITS AGE, SO THE DIMENSIONALITY REDUCTION ALGORITHM WILL MERGE THEM INTO ONE FEATURE THAT REPRESENTS THE CAR’S WEAR AND TEAR. This is called FEATURE EXTRACTION. anomaly detection,novelty detection( A very similar task is novelty detection: it aims to detect new instances that look different from all instances in the training set. This requires having a very “clean” training set, devoid of any instance that you would like the algorithm to detect. For example, if you have thousands of pictures of dogs, and 1% of these pictures represent Chihuahuas, then a novelty detection algorithm should not treat new pictures of Chihuahuas as novelties. On the other hand, anomaly detection algorithms may consider these dogs as so rare and so different from other dogs that they would likely classify them as anomalies.), **association learning** (the goal is to dig into large amounts of data and discover interesting relations between attributes. For example, suppose you own a supermarket. Running an association rule on your sales logs may reveal that people who purchase barbecue sauce and potato chips also tend to buy steak. Thus, you may want to place these items close to one another) Semisupervised learning: Since labeling data is usually time-consuming and costly, you will often have plenty of unlabeled instances, and few labeled instances. Some algorithms can deal with data that’s partially labeled. This is called semisupervised learning ex: google photos-Some photo-hosting services, such as Google Photos, are good examples of this. Once you upload all your family photos to the service, it automatically recognizes that the same person A shows up in photos 1, 5, and 11, while another person B shows up in photos 2, 5, and 7. This is the unsupervised part of the algorithm (clustering). Now all the system needs is for you to tell it who these people are. Just add one label per person4 and it is able to name everyone in every photo, which is useful for searching photos Most semisupervised learning algorithms are combinations of unsupervised and supervised algorithms. For example, deep belief networks (DBNs) are based on unsupervised components called restricted Boltzmann machines (RBMs) stacked on top of one another. RBMs are trained sequentially in an unsupervised manner, and then the whole system is fine-tuned using supervised learning techniques. Reinforcement Learning: "Reinforcement Learning is a very different beast. The learning system, called an AGENT in this context, can observe the environment, select and perform actions, and get rewards in return (or penalties in the form of negative rewards, as shown in Figure 1-12). It must then learn by itself what is the best strategy, called a POLICY, to get the most reward over time. A policy defines what action the agent should choose when it is in a given situation." For example, many robots implement Reinforcement Learning algorithms to learn how to walk. DeepMind’s AlphaGo program is also a good example of Reinforcement Learning: it made the headlines in May 2017 when it beat the world champion Ke Jie at the game of Go. It learned its winning policy by analyzing millions of games, and then playing many games against itself. Note that learning was turned off during the games against the champion; AlphaGo was just applying the policy it had learned.Batch and Online Learning: Another criterion used to classify Machine Learning systems is whether or not the system can learn incrementally from a stream of incoming data. Batch learning "IN BATCH LEARNING, THE SYSTEM IS INCAPABLE OF LEARNING INCREMENTALLY: it must be trained using all the available data. This will generally take a lot of time and computing resources, so it is typically done offline. First the system is trained, and then it is launched into production and runs without learning anymore; it just applies what it has learned. This is called OFFLINE LEARNING." "If you want a batch learning system to know about new data (such as a new type of spam), you need to train a new version of the system from scratch on the full dataset (not just the new data, but also the old data), then stop the old system and replace it with the new one." Fortunately, the whole process of training, evaluating, and launching a Machine Learning system can be automated fairly easily (as shown in Figure 1-3), so even a batch learning system can adapt to change. Simply update the data and train a new version of the system from scratch as often as needed This solution is simple and often works fine, but training using the full set of data can take many hours, so you would typically train a new system only every 24 hours or even just weekly. If your system needs to adapt to rapidly changing data (e.g., to predict stock prices), then you need a more reactive solution. Also, training on the full set of data requires a lot of computing resources (CPU, memory space, disk space, disk I/O, network I/O, etc.). If you have a lot of data and you automate your system to train from scratch every day, it will end up costing you a lot of money. If the amount of data is huge, it may even be impossible to use a batch learning algorithm. Finally, if your system needs to be able to learn autonomously and it has limited resources (e.g., a smartphone application or a rover on Mars), then carrying around large amounts of training data and taking up a lot of resources to train for hours every day is a showstopper. "Fortunately, a better option in all these cases is to use algorithms that are capable of learning incrementally." Online learning In online learning, you train the system incrementally by feeding it data instances sequentially, either individually or in small groups called mini-batches. Each learning step is fast and cheap, so the system can learn about new data on the fly, as it arrives (see Figure 1-13). Online learning is great for systems that receive data as a continuous flow (e.g., stock prices) and need to adapt to change rapidly or autonomously. "It is also a good option if you have limited computing resources: once an online learning system has learned about new data instances, it does not need them anymore, so you can discard them (unless you want to be able to roll back to a previous state and “replay” the data)." This can save a huge amount of space. Online learning algorithms can also be used to train systems on huge datasets that cannot fit in one machine’s main memory (this is called OUT-OF-CORE LEARNING). The algorithm loads part of the data, runs a training step on that data, and repeats the process until it has run on all of the data (see Figure 1-14). WARNING Out-of-core learning is usually done offline (i.e., not on the live system), so online learning can be a confusing name. Think of it as INCREMENTAL LEARNING. One important parameter of online learning systems is how fast they should adapt to changing data: this is called the LEARNING RATE. If you set a high learning rate, then your system will rapidly adapt to new data, but it will also tend to quickly forget the old data (you don’t want a spam filter to flag only the latest kinds of spam it was shown). Conversely, if you set a low learning rate, the system will have more inertia; that is, it will learn more slowly, but it will also be less sensitive to noise in the new data or to sequences of nonrepresentative data points (outliers). A big challenge with online learning is that if bad data is fed to the system, the system’s performance will gradually decline. If it’s a live system, your clients will notice. For example, bad data could come from a malfunctioning sensor on a robot, or from someone spamming a search engine to try to rank high in search results. To reduce this risk, you need to monitor your system closely and promptly switch learning off (and possibly revert to a previously working state) "if you detect a drop in performance. You may also want to monitor the input data and react to abnormal data (e.g., using an anomaly detection algorithm)."**Instance-Based Versus Model-Based Learning**: **One more way to categorize Machine Learning systems is by how they generalize(the predictions made by trained model). Most Machine Learning tasks are about making predictions. This means that given a number of training examples, the system needs to be able to make good predictions for (generalize to) examples it has never seen before**. Having a good performance measure on the training data is good, but insufficient; the true goal is to perform well on new instances. There are two main approaches to generalization: instance-based learning and model-based learning. Instance-based learning Possibly the most trivial form of learning is simply to learn by heart. If you were to create a spam filter this way, it would just flag all emails that are identical to emails that have already been flagged by users—not the worst solution, but certainly not the best. Instead of just flagging emails that are identical to known spam emails, your spam filter could be programmed to also flag emails that are very similar to known spam emails. This requires a measure of similarity between two emails. A (very basic) similarity measure between two emails could be to count the number of words they have in common. The system would flag an email as spam if it has many words in common with a known spam email. This is called instance-based learning: the system learns the examples by heart, then generalizes to new cases by using a SIMILARITY MEASURE to compare them to the learned examples (or a subset of them). For example, in Figure 1-15 the new instance would be classified as a triangle because the majority of the most similar instances belong to that class. Model-based learning Another way to generalize from a set of examples is to build a model of these examples and then use that model to make predictions. This is called model-based learning For example, suppose you want to know if money makes people happy, so you download the Better Life Index data from the OECD’s website and stats about gross domestic product (GDP) per capita from the IMF’s website. Then you join the tables and sort by GDP per capita. Table 1-1 shows an excerpt of what you get. check figure 1.17 and table above it. There does seem to be a trend here! Although the data is noisy (i.e., partly random), it looks like life satisfaction goes up more or less linearly as the country’s GDP per capita increases. So you decide to model life satisfaction as a linear function of GDP per capita. This step is called model selection: you selected a linear model of life satisfaction with just one attribute, GDP per capita WARNING Confusingly, the same word “model” can refer to a type of model (e.g., Linear Regression), to a fully specified model architecture (e.g., Linear Regression with one input and one output), or to the final trained model ready to be used for predictions (e.g., Linear Regression with one input and one output, using θ0 = 4.85 and θ1 = 4.91 × 10–5). Model selection consists in choosing the type of model and fully specifying its architecture. Training a model means running an algorithm to find the model parameters that will make it best fit the training data (and hopefully make good predictions on new data).We have covered a lot of ground so far: you now know what Machine Learning is really about, why it is useful, what some of the most common categories of ML systems are, and what a typical project workflow looks like. Now let’s look at what can go wrong in learning and prevent you from making accurate predictions.Main Challenges of Machine Learning In short, since your main task is to select a learning algorithm and train it on some data, the two things that can go wrong are “bad algorithm” and “bad data.” .MOREOVER DATA AND ALGORITHM ARE THE ONLY DETERMINING FACTORS OF SUCESSFULL SOLUTION AN PROBLEM CAN HAVE.Let’s start with examples of bad data. Insufficient Quantity of Training Data For a toddler to learn what an apple is, all it takes is for you to point to an apple and say “apple” (possibly repeating this procedure a few times). Now the child is able to recognize apples in all sorts of colors and shapes. Genius., Machine Learning is not quite there yet; "it takes a lot of data for most Machine Learning algorithms to work properly. Even for very simple problems you typically need thousands of examples, and for complex problems such as image or speech recognition you may need millions of examples (unless you can reuse parts of an existing model).," THE UNREASONABLE EFFECTIVENESS OF DATA IN A FAMOUS PAPER PUBLISHED IN 2001, MICROSOFT RESEARCHERS MICHELE BANKO AND ERIC BRILL SHOWED THAT VERY DIFFERENT MACHINE LEARNING ALGORITHMS, INCLUDING FAIRLY SIMPLE ONES, PERFORMED ALMOST IDENTICALLY WELL ON A COMPLEX PROBLEM OF NATURAL LANGUAGE DISAMBIGUATION8 ONCE THEY WERE GIVEN ENOUGH DATA (AS YOU CAN SEE IN FIGURE 1-20). As the authors put it, “these results suggest that we may want to reconsider the trade-off between spending time and money on algorithm development versus spending it on corpus development.” The idea that data matters more than algorithms for complex problems was further popularized by Peter Norvig et al. in a paper titled “The Unreasonable Effectiveness of Data”, published in 2009.10 It should be noted, however, that small- and medium-sized datasets are still very common, and it is not always easy or cheap to get extra training data⁠—so don’t abandon algorithms just yet. Nonrepresentative Training Data: refer example in book in chapter 1 of Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition in short words- if the data that you trained the model on and the new data that you want the machine to generalize on(i.e, the same code will not work on the new data that have a lot of diversion from the training data) By using a nonrepresentative training set, we trained a model that is unlikely to make accurate predictions, especially for very poor and very rich countries. It is crucial to use a training set that is representative of the cases you want to generalize to. This is often harder than it sounds: if the sample is too small, you will have sampling noise (i.e., nonrepresentative data as a result of chance), but even very large samples can be nonrepresentative if the sampling method is flawed. This is called SAMPLING BIAS.(sampling is the process of taking a part of the dataset) Perhaps the most famous example of sampling bias happened during the US presidential election in 1936, which pitted Landon against Roosevelt: the Literary Digest conducted a very large poll, sending mail to about 10 million people. It got 2.4 million answers, and predicted with high confidence that Landon would get 57% of the votes. Instead, Roosevelt won with 62% of the votes. The flaw was in the Literary Digest’s sampling method: First, to obtain the addresses to send the polls to, the Literary Digest used telephone directories, lists of magazine subscribers, club membership lists, and the like. All of these lists tended to favor wealthier people, who were more likely to vote Republican (hence Landon). Second, less than 25% of the people who were polled answered. Again this introduced a sampling bias, by potentially ruling out people who didn’t care much about politics, people who didn’t like the Literary Digest, and other key groups. This is a special type of sampling bias called NONRESPONSE BIAS. Poor-Quality Data Obviously, if your training data is full of errors, outliers, and noise (e.g., due to poor-quality measurements), it will make it harder for the system to detect the underlying patterns, so your system is less likely to perform well. It is often well worth the effort to spend time cleaning up your training data. The truth is, most data scientists spend a significant part of their time doing just that. The following are a couple of examples of when you’d want to clean up training data: If some instances are clearly outliers, it may help to simply discard them or try to fix the errors manually. If some instances are missing a few features (e.g., 5% of your customers did not specify their age), you must decide whether you want to ignore this attribute altogether, ignore these instances, fill in the missing values (e.g., with the median age), or train one model with the feature and one model without it. Irrelevant Features As the saying goes: garbage in, garbage out. Your system will only be capable of learning if the training data contains enough relevant features and not too many irrelevant ones. A critical part of the success of a Machine Learning project is coming up with a good set of features to train on. This process, called feature engineering, involves the following steps: Feature selection (selecting the most useful features to train on among existing features) Feature extraction (combining existing features to produce a more useful one⁠—as we saw earlier, dimensionality reduction algorithms can help) Creating new features by gathering new data Now that we have looked at many examples of bad data, let’s look at a couple of examples of bad algorithms Overfitting the Training Data Say you are visiting a foreign country and the taxi driver rips you off. You might be tempted to say that all taxi drivers in that country are thieves. Overgeneralizing is something that we humans do all too often, and unfortunately machines can fall into the same trap if we are not careful. In Machine Learning this is called overfitting: it means that the model performs well on the training data, but it does not generalize well when a model understands the data and noise too well that it negatively impacts the perfomance of the model on new data WARNING Overfitting happens when the model is too complex relative to the amount and noisiness of the training data. Here are possible solutions:, Simplify the model by selecting one with fewer parameters (e.g., a linear model rather than a high-degree polynomial model), by reducing the number of attributes in the training data, or by constraining the model., Gather more training data., Reduce the noise in the training data (e.g., fix data errors and remove outliers)., ***Constraining a model to make it simpler and reduce the risk of overfitting is called regularization***. For example, the linear model we defined earlier has two parameters, θ0 and θ1. This gives the learning algorithm two degrees of freedom to adapt the model to the training data: it can tweak both the height (θ0) and the slope (θ1) of the line. If we forced θ1 = 0, the algorithm would have only one degree of freedom and would have a much harder time fitting the data properly: all it could do is move the line up or down to get as close as possible to the training instances, so it would end up around the mean. A very simple model indeed! If we allow the algorithm to modify θ1 but we force it to keep it small, then the learning algorithm will effectively have somewhere in between one and two degrees of freedom. It will produce a model that’s simpler than one with two degrees of freedom, but more complex than one with just one. You want to find the right balance between fitting the training data perfectly and keeping the model simple enough to ensure that it will generalize well. THE AMOUNT OF REGULARIZATION TO APPLY DURING LEARNING CAN BE CONTROLLED BY A HYPERPARAMETER. A HYPERPARAMETER IS A PARAMETER OF A LEARNING ALGORITHM (NOT OF THE MODEL). AS SUCH, IT IS NOT AFFECTED BY THE LEARNING ALGORITHM ITSELF; "it must be set prior to training and remains constant during training. If you set the regularization hyperparameter to a very large value, you will get an almost flat model (a slope close to zero); the learning algorithm will almost certainly not overfit the training data, but it will be less likely to find a good solution. Tuning hyperparameters is an important part of building a Machine Learning system" (you will see a detailed example in the next chapter). Underfitting the Training Data As you might guess, underfitting is the opposite of overfitting: IT OCCURS WHEN YOUR MODEL IS TOO SIMPLE TO LEARN THE UNDERLYING STRUCTURE OF THE DATA. For example, a linear model of life satisfaction is prone to underfit; REALITY IS JUST MORE COMPLEX THAN THE MODEL, SO ITS PREDICTIONS ARE BOUND TO BE INACCURATE, EVEN ON THE TRAINING EXAMPLES., Here are the main options for fixing this problem:, Select a more powerful model, with more parameters., Feed better features to the learning algorithm (feature engineering)., Reduce the constraints on the model (e.g., reduce the regularization hyperparameter). Stepping Back BY NOW YOU KNOW A LOT ABOUT MACHINE LEARNING. HOWEVER, WE WENT THROUGH SO MANY CONCEPTS THAT YOU MAY BE FEELING A LITTLE LOST, SO LET’S STEP BACK AND LOOK AT THE BIG PICTURE:, Machine Learning is about making machines get better at some task by learning from data, instead of having to explicitly code rules., There are many different types of ML systems: supervised or not, batch or online, instance-based or model-based., In an ML project you gather data in a training set, and you feed the training set to a learning algorithm. If the algorithm is model-based, it tunes some parameters to fit the model to the training set (i.e., to make good predictions on the training set itself), and then hopefully it will be able to make good predictions on new cases as well. If the algorithm is instance-based, it just learns the examples by heart and generalizes to new instances by using a similarity measure to compare them to the learned instances., The system will not perform well if your training set is too small, or if the data is not representative, is noisy, or is polluted with irrelevant features (garbage in, garbage out). Lastly, your model needs to be neither too simple (in which case it will underfit) nor too complex (in which case it will overfit). THERE’S JUST ONE LAST IMPORTANT TOPIC TO COVER: ONCE YOU HAVE TRAINED A MODEL, YOU DON’T WANT TO JUST “HOPE” IT GENERALIZES TO NEW CASES. YOU WANT TO EVALUATE IT AND FINE-TUNE IT IF NECESSARY. LET’S SEE HOW TO DO THAT.

Testing and Validating

The only way to know how well a model will generalize to new cases is to actually try it out on new cases. One way to do that is to put your model in production and monitor how well it performs. This works well, but if your model is horribly bad, your users will complain—not the best idea.

A better option is to split your data into two sets: the *training set* and the *test set*. As these names imply, you train your model using the training set, and you test it using the test set. The error rate on new cases is called the *generalization error* (or *out-of-sample error*), and by evaluating your model on the test set, you get an estimate of this error. This value tells you how well your model will perform on instances it has never seen before.

If the training error is low (i.e., your model makes few mistakes on the training set) but the generalization error is high, it means that your model is overfitting the training data.

**TIP**

It is common to use 80% of the data for training and *hold out* 20% for testing. However, this depends on the size of the dataset: if it contains 10 million instances, then holding out 1% means your test set will contain 100,000 instances, probably more than enough to get a good estimate of the generalization error.

## Hyperparameter Tuning and Model Selection

Evaluating a model is simple enough: just use a test set. But suppose you are hesitating between two types of models (say, a linear model and a polynomial model): how can you decide between them? One option is to train both and compare how well they generalize using the test set.

Now suppose that the linear model generalizes better, but you want to apply some regularization to avoid overfitting. The question is, how do you choose the value of the regularization hyperparameter? One option is to train 100 different models using 100 different values for this hyperparameter. Suppose you find the best hyperparameter value that produces a model with the lowest generalization error⁠—say, just 5% error. You launch this model into production, but unfortunately it does not perform as well as expected and produces 15% errors. What just happened?

The problem is that you measured the generalization error multiple times on the test set, and you adapted the model and hyperparameters to produce the best model for that particular set. This means that the model is unlikely to perform as well on new data.

A common solution to this problem is called **holdout validation**: you simply hold out part of the training set to evaluate several candidate models and select the best one. The new held-out set is called the **validation set** (or sometimes the development set, or dev set). More specifically, you train multiple *models with various hyperparameters* on the ***reduced training set*** *(i.e., the full training set minus the validation set)*, and you select the model that performs best on the validation set. After this holdout validation process, you train the best model on the full training set (including the validation set), and this gives you the final model. Lastly, you evaluate this final model on the test set to get an estimate of the generalization error.

This solution usually works quite well. However, if the validation set is too small, then model evaluations will be imprecise: you may end up selecting a suboptimal model by mistake. Conversely, if the validation set is too large, then the remaining training set will be much smaller than the full training set. Why is this bad? Well, since the final model will be trained on the full training set, it is not ideal to compare candidate models trained on a much smaller training set. It would be like selecting the fastest sprinter to participate in a marathon. One way to solve this problem is to perform repeated **cross-validation**, using many small validation sets. **Each model is evaluated once per validation set after it is trained on the rest of the data**. By averaging out all the evaluations of a model, you get a much more accurate measure of its performance. There is a drawback, however: the training time is multiplied by the number of validation sets.

Data Mismatch

In some cases, it’s easy to get a large amount of data for training, but this data probably won’t be perfectly representative of the data that will be used in production. For example, suppose you want to create a mobile app to take pictures of flowers and automatically determine their species. You can easily download millions of pictures of flowers on the web, but they won’t be perfectly representative of the pictures that will actually be taken using the app on a mobile device. Perhaps you only have **10,000 representative pictures (i.e., actually taken with the app**). In this case**, the most important rule to remember is that the validation set and the test set must be as representative as possible of the data you expect to use in production**, *so they should be composed exclusively of representative pictures: you can shuffle them and put half in the validation set and half in the test set (****making sure that no duplicates or near-duplicates end up in both sets****). But after training your model on the web pictures, if you observe that the performance of the model on the validation set is disappointing, you will not know whether this is because your model has overfit the training set, or whether this is just due to the mismatch between the web pictures and the mobile app pictures*. One solution is to hold out some of the training pictures (from the web) in yet another set that Andrew Ng calls the train-dev set. After the model is trained (on the training set, not on the train-dev set), **you can evaluate it on the train-dev set. If it performs well, then the model is not overfitting the training set**. “If it performs poorly on the validation set, the problem must be coming from the **data mismatch(try understanding the scenario here, the web pictures that you used are trying to dominate the model generalization factor, as web pictures are different from the images that are taken from the app, it is better to apply some techniques to make web pictures look like the ones in the app)**. You can try to tackle this problem by pre-processing the web images to make them look more like the pictures that will be taken by the mobile app, and then retraining the model. Conversely, if the model performs poorly on the train-dev set, then it must have overfit the training set, so you should try to simplify or regularize the model, get more training data, and clean up the training data.”

NO FREE LUNCH THEOREM

A model is a simplified version of the observations. The simplifications are meant to discard the superfluous details that are unlikely to generalize to new instances. To decide what data to discard and what data to keep, you must make assumptions. For example, a linear model makes the assumption that the data is fundamentally linear and that the distance between the instances and the straight line is just noise, which can safely be ignored.

In a famous 1996 paper,11 David Wolpert demonstrated that if you make absolutely no assumption about the data, then there is no reason to prefer one model over any other. This is called the No Free Lunch (NFL) theorem. For some datasets the best model is a linear model, while for other datasets it is a neural network. There is no model that is a priori guaranteed to work better (hence the name of the theorem). The only way to know for sure which model is best is to evaluate them all. Since this is not possible, in practice you make some reasonable assumptions about the data and evaluate only a few reasonable models. For example, for simple tasks you may evaluate linear models with various levels of regularization, and for a complex problem you may evaluate various neural networks.

***Exercises:***

* How would you define Machine Learning?

Machine learning is about building systems that can learn form data, learning means get better at some task, given a performance measure

* Can you name four types of problems where it shines?

For complex problems where there is no algorithm, for a problem where to replace long lis of hand tuned rules, adapting to fluctuating environments, to help humans learn (getting insights from huge data)

1. What is a labeled training set?

A labelled training set contains label(a solution, a tag like in the example of spam filtering mails, where while training we will inform the model whether each particular mail is a spam or ham),

1. What are the two most common supervised tasks?

**Classification and regression**

1. Can you name four common unsupervised tasks?

Visualization, clustering, Dimensionality Reduction and Association Rule Learning

1. What type of Machine Learning algorithm would you use to allow a robot to walk in various unknown terrains?

Reinforcement Learning is likely to perform best if we want a robot to learn to walk in various unknown terrains, since this is typically the type of problem that Reinforcement Learning tackles. It might be possible to express the problem as a supervised or semisupervised learning problem, but it would be less natural.

1. What type of algorithm would you use to segment your customers into multiple groups?

If you don’t know how to define the groups, then you can use a clustering algorithm (unsupervised learning) to segment your customers into clusters of similar customers. However**, if you know what groups you would like to have, then you can feed many examples of each group to a classification algorithm (supervised learning), and it will classify all your customers into these groups.**

1. Would you frame the problem of spam detection as a supervised learning problem or an unsupervised learning problem?

Spam detection is a typical supervised learning problem: the algorithm is fed many emails along with their labels

1. What is an online learning system?

An online learning system can learn incrementally, as opposed to a batch learning system. This makes it capable of adapting rapidly to both changing data and autonomous systems, and of training on very large quantities of data.

1. What is out-of-core learning?

This learning can handle large amount of data that cannot fit in computer main memory, by partitioning it into small chunks called **minim-batches**  and uses **online learning techniques** to learn from mini batches

1. What type of learning algorithm relies on a similarity measure to make predictions?
2. An **instance-based learning system learns the training data by heart**; then, **when given a new instance, it uses a similarity measure to find the most similar learned instances and uses them to make predictions**.
3. What is the difference between a model parameter and a learning algorithm’s hyperparameter?

**A model has one or more model parameters that determine what it will predict given a new instance** (e.g., the slope of a linear model). A **learning algorithm tries to find optimal values for these parameters such that the model generalizes well to new instances. A hyperparameter is a parameter of the learning algorithm itself, not of the model (e.g., the amount of regularization to apply).**

1. What do model-based learning algorithms search for? What is the most common strategy they use to succeed? How do they make predictions?

Model-based learning algorithms search for an **optimal value** for the model parameters such that the model will generalize well to new instances. We usually train such systems by minimizing a **cost function** (The cost function is the technique of evaluating “the performance of our algorithm/model”. It takes both predicted outputs by the model and actual outputs and calculates how much wrong the model was in its prediction. It outputs a higher number if our predictions differ a lot from the actual values)that measures how bad the system is at making predictions on the training data, plus a penalty for model complexity if the model is regularized. To make predictions, we feed the new instance’s features into the model’s prediction function, using the parameter values found by the learning algorithm.

1. Can you name four of the main challenges in Machine Learning?

Lack of data, poor data Quality, Non representative data, Uninformative features, over fitting, under fitting

1. If your model performs great on the training data but generalizes poorly to new instances, what is happening? Can you name three possible solutions?

If a model performs great on the training data but generalizes poorly to new instances, the model is likely overfitting the training data (or we got extremely lucky on the training data). Possible solutions to overfitting are getting more data, simplifying the model (selecting a simpler algorithm, reducing the number of parameters or features used, or regularizing the model), or reducing the noise in the training data.

1. What is a test set, and why would you want to use it?

A test set is used **to estimate the generalization error that a model will make on new instances**, before the model is launched in production.

1. What is the purpose of a validation set?

A validation set is used to **compare models**, and **used to select the best model and tune the hyperparamters**

1. What is the train-dev set, when do you need it, and how do you use it?

The train-dev set is used when there is a risk of mismatch between the training data and the data used in the validation and test datasets (which should always be as close as possible to the data used once the model is in production). The train-dev set is a part of the training set that’s held out (the model is not trained on it). The model is trained on the rest of the training se t, and evaluated on both the train-dev set and the validation set. If the model performs well on the training set but not on the train-dev set, then the model is likely overfitting the training set. If it performs well on both the training set and the train-dev set, but not on the validation set, then there is probably a significant data mismatch between the training data and the validation + test data, and you should try to improve the training data to make it look more like the validation + test data

1. What can go wrong if you tune hyperparameters using the test set?

If you tune hyperparameters using the test set, you risk **overfitting** the test set, and the generalization error you measure will be optimistic (**you may launch a model that performs worse than you expect**).

Chapter 2. End-to-End Machine Learning Project

In this chapter you will work through an example project end to end, pretending to be a recently hired data scientist at a real estate company.[**1**](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch02.html#idm45022192822104) Here are the main steps you will go through:

1. Look at the big picture.
2. Get the data.
3. Discover and visualize the data to gain insights.
4. Prepare the data for Machine Learning algorithms.
5. Select a model and train it.
6. Fine-tune your model.
7. Present your solution.
8. Launch, monitor, and maintain your system.

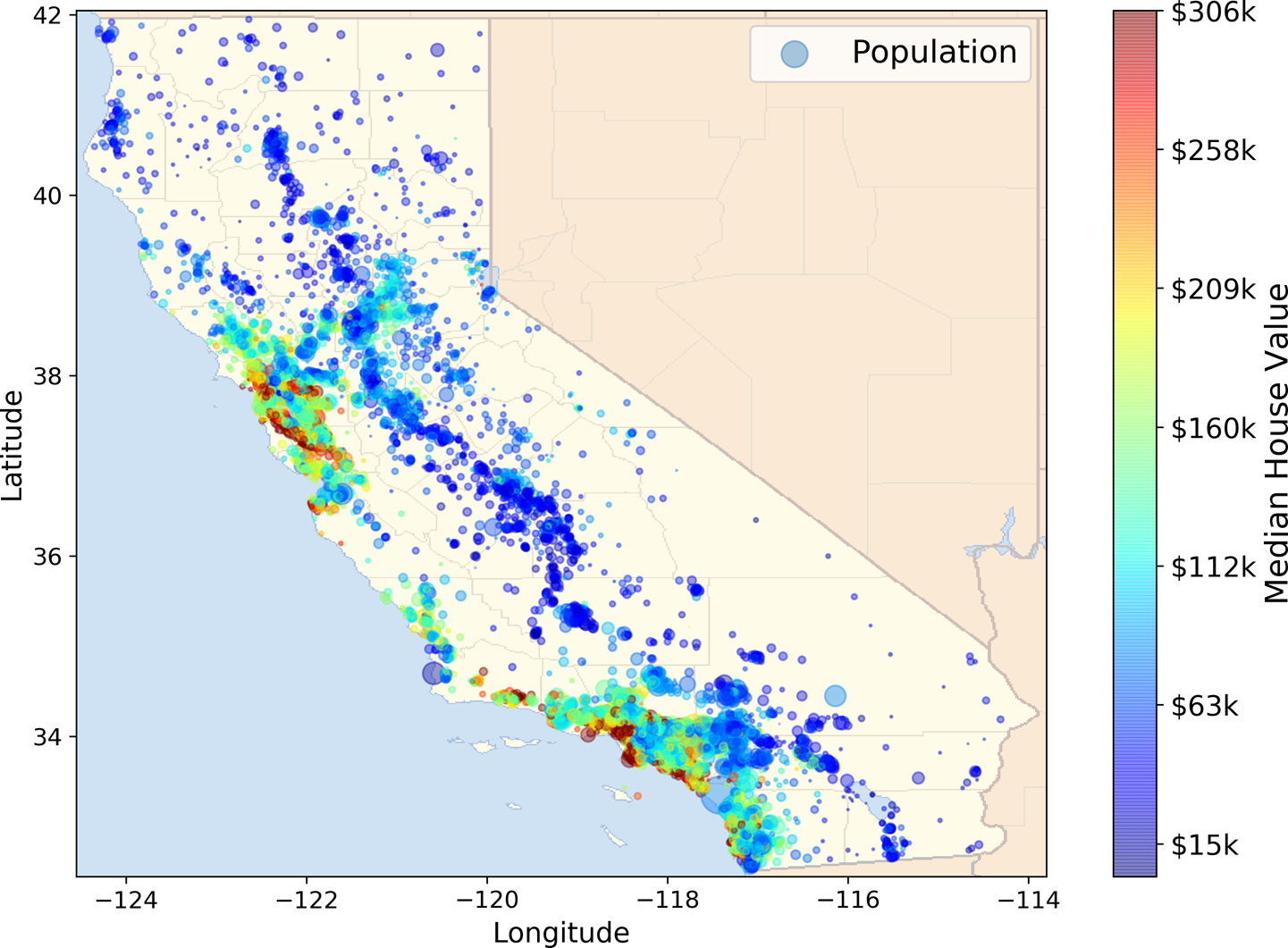
**The above steps are the building blocks of creating an ML model**

# Working with Real Data

**When you are learning about Machine Learning, it is best to experiment with real-world data**, not artificial datasets. Fortunately, there are thousands of open datasets to choose from, ranging across all sorts of domains. Here are a few places you can look to get data:

1. Popular open data repositories
   1. [UC Irvine Machine Learning Repository](http://archive.ics.uci.edu/ml/)
   2. [Kaggle datasets](https://www.kaggle.com/datasets)
   3. [Amazon’s AWS datasets](https://registry.opendata.aws/)
2. Meta portals (they list open data repositories)
   1. [Data Portals](http://dataportals.org/)
   2. [OpenDataMonitor](http://opendatamonitor.eu/)
   3. [Quandl](http://quandl.com/)
3. Other pages listing many popular open data repositories
   1. [Wikipedia’s list of Machine Learning datasets](https://homl.info/9)
   2. [Quora.com](https://homl.info/10)
   3. [The datasets subreddit](https://www.reddit.com/r/datasets)

In this chapter we’ll use the California Housing Prices dataset from the StatLib repository[**2**](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch02.html#idm45022192789784) (see [Figure 2-1](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch02.html#california_housing_prices_plot)). This dataset is based on data from the 1990 California census. It is not exactly recent (a nice house in the Bay Area was still affordable at the time), but it has many qualities for learning, so we will pretend it is recent data. For teaching purposes I’ve added a categorical attribute and removed a few features.



###### Figure 2-1. California housing prices

# Look at the Big Picture

Welcome to the Machine Learning Housing Corporation! Your first task is to use California census data to build a model of housing prices in the state. This data includes metrics such as the population, median income, and median housing price for each block group in California. Block groups are the smallest geographical unit for which the US Census Bureau publishes sample data (a block group typically has a population of 600 to 3,000 people). We will call them “districts” for short.

Your model should learn from this data and be able to predict the median housing price in any district, given all the other metrics.

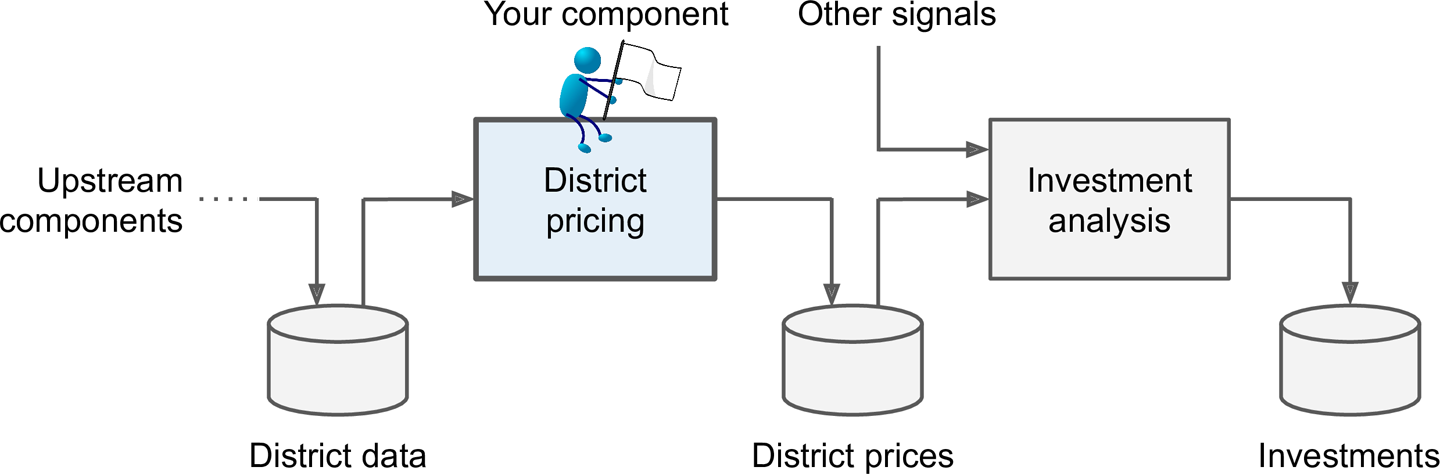
###### TIP

Since you are a well-organized data scientist, the first thing you should do is pull out your Machine Learning project checklist. You can start with the one in [Appendix B](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/app02.html#project_checklist_appendix); it should work reasonably well for most Machine Learning projects, but make sure to adapt it to your needs. In this chapter we will go through many checklist items, but **we will also skip a few, either because they are self-explanatory or because they will be discussed in later chapters**.

## Frame the Problem

“**The first question to ask your boss is what exactly the business objective is**. Building a model is probably not the end goal. How does the company expect to use and benefit from this model? Knowing the objective is important because it will determine how you frame the problem, which algorithms you will select, which performance measure you will use to evaluate your model, and how much effort you will spend tweaking it.”

Your boss answers that your model’s output (a prediction of a district’s median housing price) will be fed to another Machine Learning system (see [Figure 2-2](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch02.html#house_pricing_pipeline_diagram)), along with many other signals(A piece of information fed to a Machine Learning system is often called a signal). This downstream system will determine whether it is worth investing in a given area or not. Getting this right is critical, as it directly affects revenue.



###### Figure 2-2. A Machine Learning pipeline for real estate investments

##### PIPELINES

**A sequence of data processing components is called a data pipeline**. Pipelines are very common in Machine Learning systems, since there is a lot of data to manipulate and many data transformations to apply.

Components typically run asynchronously. Each component pulls in a large amount of data, processes it, and spits out the result in another data store. Then, some time later, the next component in the pipeline pulls this data and spits out its own output. Each component is fairly self-contained: the interface between components is simply the data store. This makes the system simple to grasp (with the help of a data flow graph), and different teams can focus on different components. Moreover, if a component breaks down, the downstream components can often continue to run normally (at least for a while) by just using the last output from the broken component. This makes the architecture quite robust.

On the other hand, a broken component can go unnoticed for some time if proper monitoring is not implemented. The data gets stale and the overall system’s performance drops.

**The next question to ask your boss is what the current solution looks like (if any). The current situation will often give you a reference for performance, as well as insights on how to solve the problem**. Your boss answers that the district housing prices are currently estimated manually by experts: a team gathers up-to-date information about a district, and when they cannot get the median housing price, they estimate it using complex rules.

This is costly and time-consuming, and their estimates are not great; in cases where they manage to find out the actual median housing price, they often realize that their estimates were off by more than 20%. This is why the company thinks that it would be useful to train a model to predict a district’s median housing price, given other data about that district. The census data looks like a great dataset to exploit for this purpose, since it includes the median housing prices of thousands of districts, as well as other data.

With all this information, you are now ready to start designing your system. First, you need to frame the problem: is it supervised, unsupervised, or Reinforcement Learning? Is it a classification task, a regression task, or something else? Should you use batch learning or online learning techniques? Before you read on, pause and try to answer these questions for yourself.

Have you found the answers? Let’s see: it is clearly a **typical supervised learning task**, since you are given labelled training examples (each instance comes with the expected output, i.e., the district’s median housing price). **It is also a typical regression task, since you are asked to predict a value**. More specifically, this is a multiple regression problem, since the system will use multiple features to make a prediction (it will use the district’s population, the median income, etc.). It is also a **univariate regression** problem, since we are only trying to predict a single value for each district. If we were trying to predict multiple values per district, it would be a **multivariate regression** problem. Finally, *there is no continuous flow of data coming into the system*, there is no particular need to adjust to changing data rapidly, and **the data is small enough to fit in memory, so plain batch learning should do just fine.**

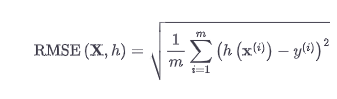
###### TIP

If the data were huge, you could either split your batch learning work across multiple servers (using the MapReduce technique) or use an online learning technique

## Select a Performance Measure

Your next step is to select a **performance measure. A typical performance measure for regression problems is the Root Mean Square Error** (RMSE). It gives an idea of how much error the system typically makes in its predictions, with a higher weight for large errors. [Equation 2-1](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch02.html#rmse_equation) shows the mathematical formula to compute the RMSE.

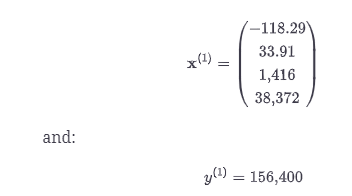
##### Equation 2-1. Root Mean Square Error (RMSE)



##### NOTATIONS

This equation introduces several very common Machine Learning notations that we will use throughout this book:

1. m is the number of instances in the dataset you are measuring the RMSE on.
   1. For example, if you are evaluating the RMSE on a validation set of 2,000 districts, then m = 2,000.
2. **x(i)** is a vector of all the feature values (excluding the label) of the i**th** instance in the dataset, and y**(i)** is its label (the desired output value for that instance).
   1. For example, if the first district in the dataset is located at longitude –118.29°, latitude 33.91°, and it has 1,416 inhabitants with a median income of $38,372, and the median house value is $156,400 (ignoring the other features for now), then:



1. **X** is a matrix containing all the feature values (excluding labels) of all instances in the dataset. There is one row per instance, and the i**th** row is equal to the transpose of **x(i)**, noted .[**4**](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch02.html#idm45022192690152)
   1. For example, if the first district is as just described, then the matrix **X** looks like this:

A picture containing graphical user interface

Description automatically generated

* + 1. h is your **system’s prediction function**, also called a **hypothesis**. When your system is given an instance’s feature vector **x(i)**, it outputs a predicted value for that instance (ŷ is pronounced “y-hat”).
  1. For example, if your system predicts that the median housing price in the first district is $158,400, then ŷ**(i)** = h(**x(i)**) = 158,400. The prediction error for this district is ŷ**(i)** – y**(i)** = 2,000.

1. RMSE(**X**,h) is the cost function measured on the set of examples using your hypothesis h.

We use lowercase italic font for scalar values (such as m or y**(i)**) and function names (such as h), lowercase bold font for vectors (such as **x(i)**), and uppercase bold font for matrices (such as **X**).

Even though the RMSE is generally the preferred performance measure for regression tasks, in some contexts you may prefer to use another function. For example, suppose that there are many outlier districts. In that case, you may consider using the mean absolute error (MAE, also called the average absolute deviation; see [Equation 2-2](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch02.html#mae_equation)):

##### Equation 2-2. Mean absolute error (MAE)

Diagram, schematic

Description automatically generated

Both the RMSE and the MAE are ways to measure the **distance between two vectors: the vector of predictions and the vector of target values**. Various distance measures, or **norms**, are possible:

1. Computing the root of a sum of squares (RMSE) corresponds to the Euclidean norm: this is the notion of distance you are familiar with. It is also called the ℓ**2** norm, noted ∥ · ∥**2** (or just ∥ · ∥).
2. Computing the sum of absolutes (MAE) corresponds to the ℓ**1** norm, noted ∥ · ∥**1**. This is sometimes called the Manhattan norm because it measures the distance between two points in a city if you can only travel along orthogonal city blocks.
3. More generally, the ℓ**k** norm of a vector **v** containing n elements is defined as ∥**v**∥**k** = (|v**0**|**k** + |v**1**|**k** + ... + |v**n**|**k**)**1/k**. ℓ**0** gives the number of nonzero elements in the vector, and ℓ**∞** gives the maximum absolute value in the vector.
4. The higher the norm index, the more it focuses on large values and neglects small ones. This is why the **RMSE is more sensitive to outliers than the MAE**. But when outliers are exponentially rare (like in a bell-shaped curve), the RMSE performs very well and is generally preferred.

## **Check the Assumptions**

Lastly, it is good practice to list and verify the assumptions that have been made so far (by you or others); this can help you catch serious issues early on. For example, the district prices that your system outputs are going to be fed into a downstream Machine Learning system, and you assume that these prices are going to be used as such. But what if the downstream system converts the prices into categories (e.g., “cheap,” “medium,” or “expensive”) and then uses those categories instead of the prices themselves? In this case, getting the price perfectly right is not important at all; your system just needs to get the category right. If that’s so, then the problem should have been framed as a classification task, not a regression task. You don’t want to find this out after working on a regression system for months.

Fortunately, after talking with the team in charge of the downstream system, you are confident that they do indeed need the actual prices, not just categories. Great! You’re all set, the lights are green, and you can start coding now!

## Create a Test Set

suppose you chatted with experts who told you that the median income is a very important attribute to predict median housing prices. You may want to ensure that the test set is representative of the various categories of incomes in the whole dataset. Since the median income is a continuous numerical attribute, you first need to create an income category attribute. Let’s look at the median income histogram more closely (back in [Figure 2-8](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch02.html#attribute_histogram_plots)): most median income values are clustered around 1.5 to 6 (i.e., $15,000–$60,000), but some median incomes go far beyond 6. It is important to have a sufficient number of instances in your dataset for each stratum, or else the estimate of a stratum’s importance may be biased. This means that you should not have too many strata, and each stratum should be large enough. The following code uses the pd.cut() function to create an income category attribute with five categories (labeled from 1 to 5): category 1 ranges from 0 to 1.5 (i.e., less than $15,000), category 2 from 1.5 to 3, and so on:

housing["income\_cat"] = pd.cut(housing["median\_income"],

bins=[0., 1.5, 3.0, 4.5, 6., np.inf],

labels=[1, 2, 3, 4, 5])

These income categories are represented in [Figure 2-9](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch02.html#housing_income_cat_hist_screenshot):

housing["income\_cat"].hist()

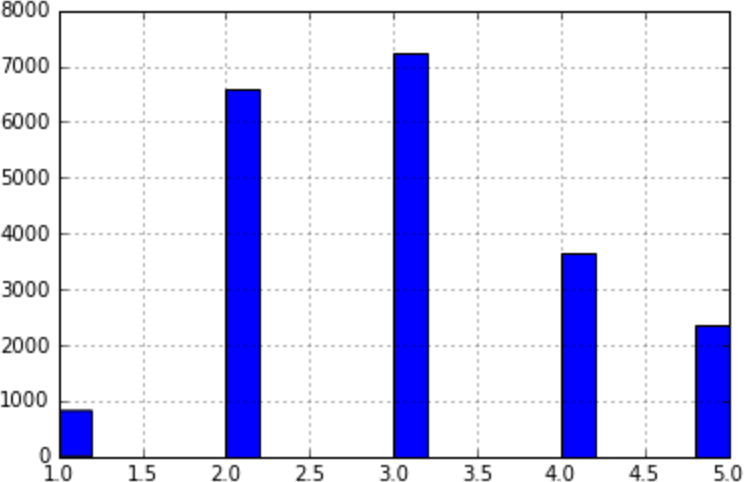


Figure 2-9. Histogram of income categories

**Now you are ready to do stratified sampling based on the income category. For this you can use Scikit-Learn’s StratifiedShuffleSplit class:**

from sklearn.model\_selection import StratifiedShuffleSplit

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

for train\_index, 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:

>>> strat\_test\_set["income\_cat"].value\_counts() / len(strat\_test\_set)

3 0.350533

2 0.318798

4 0.176357

5 0.114583

1 0.039729

Name: income\_cat, dtype: float64

With similar code you can measure the income category proportions in the full dataset. [Figure 2-10](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch02.html#compare_sampling_errors_screenshot) compares the income category proportions in the overall dataset, in the test set generated with stratified sampling, and in a 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 skewed.

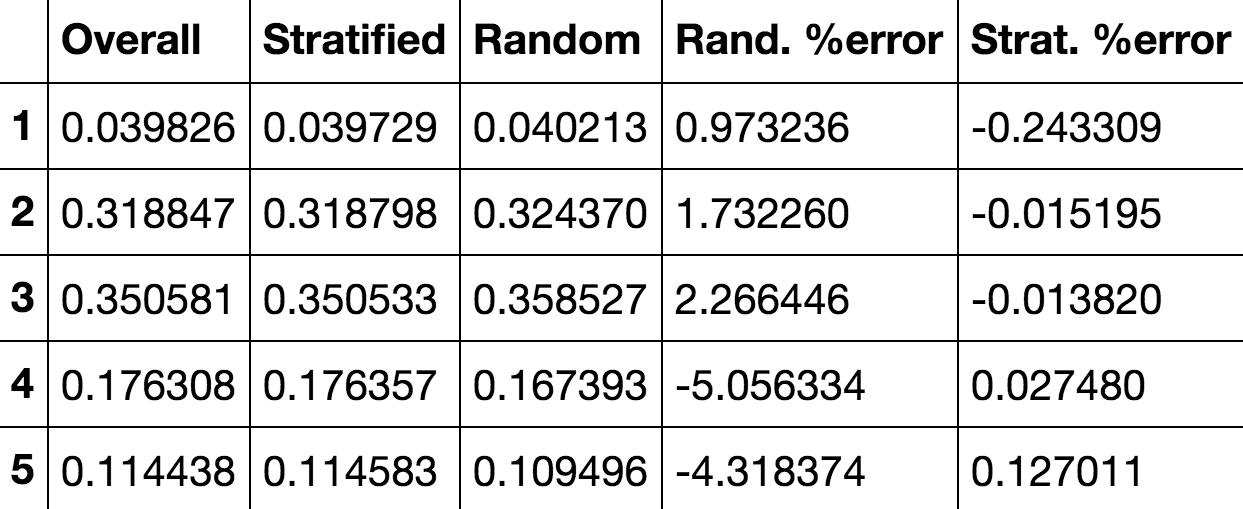


Figure 2-10. Sampling bias comparison of stratified versus purely random sampling

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

for set\_ in (strat\_train\_set, strat\_test\_set):

set\_.drop("income\_cat", axis=1, inplace=True)

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 a general understanding of the kind of data you are manipulating. Now the goal is to go into a little more 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 want 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 that 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 (Figure 2-11):

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

Figure 2-11. A geographical scatterplot of the data

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 a high density of data points (Figure 2-12):

housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.1)

Figure 2-12. A better visualization that highlights high-density areas

Now that’s much better: you can clearly see the 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.

Our brains are very good at spotting patterns in pictures, but you may need to play around with visualization parameters to make the patterns stand out.

Now let’s look at the housing prices (Figure 2-13). 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):16

housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.4,

s=housing["population"]/100, label="population", figsize=(10,7),

c="median\_house\_value", cmap=plt.get\_cmap("jet"), colorbar=True,

)

plt.legend()

Figure 2-13. California housing prices: red is expensive, blue is cheap, larger circles indicate areas with a larger population

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. A clustering algorithm should be useful for detecting the main cluster and for adding 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 is not a simple rule.

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 (Figure 2-11):

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

Figure 2-11. A geographical scatterplot of the data

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 a high density of data points (Figure 2-12):

housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.1)

Figure 2-12. A better visualization that highlights high-density areas

Now that’s much better: you can clearly see the 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.

Our brains are very good at spotting patterns in pictures, but you may need to play around with visualization parameters to make the patterns stand out.

Now let’s look at the housing prices (Figure 2-13). 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):16

housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.4,

s=housing["population"]/100, label="population", figsize=(10,7),

c="median\_house\_value", cmap=plt.get\_cmap("jet"), colorbar=True,

)

plt.legend()

Figure 2-13. California housing prices: red is expensive, blue is cheap, larger circles indicate areas with a larger population

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. A clustering algorithm should be useful for detecting the main cluster and for adding 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 is 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:

corr\_matrix = housing.corr()

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

>>> corr\_matrix["median\_house\_value"].sort\_values(ascending=False)

median\_house\_value 1.000000

median\_income 0.687170

total\_rooms 0.135231

housing\_median\_age 0.114220

households 0.064702

total\_bedrooms 0.047865

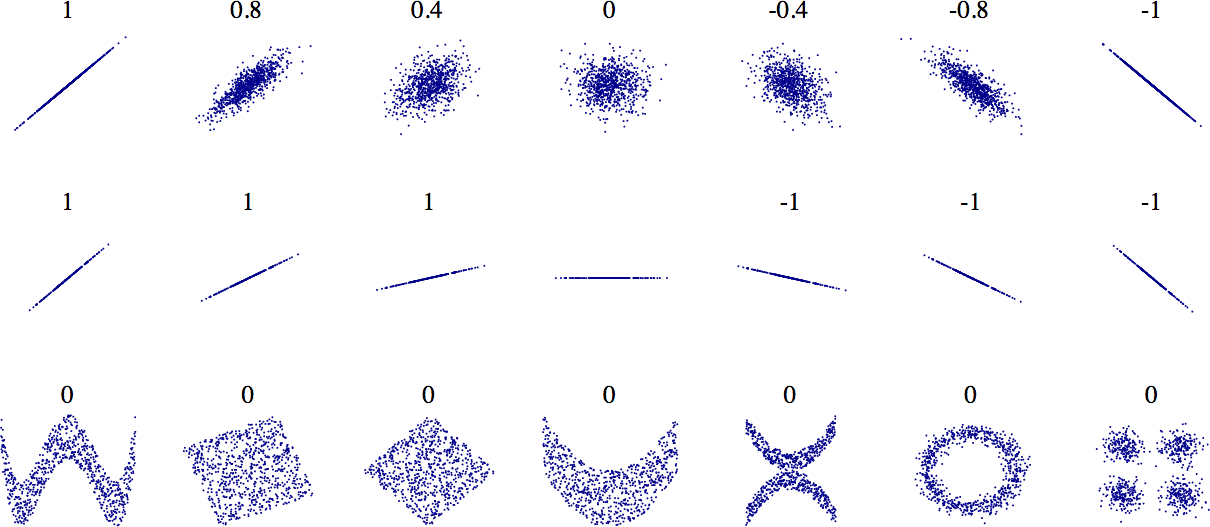
population -0.026699

longitude -0.047279

latitude -0.142826

Name: median\_house\_value, dtype: float64

**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 tends to 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 0 mean that there is no linear correlation.** [Figure 2-14](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch02.html#correlation_coefficient_plots) shows various plots along with the correlation coefficient between their horizontal and vertical axes.



###### Figure 2-14. Standard correlation coefficient of various datasets (source: Wikipedia; public domain image)

###### WARNING

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 0, then y generally goes up”). Note how all the plots of the bottom row have a correlation coefficient equal to 0, despite the fact that their axes are clearly not 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 in nanometres.

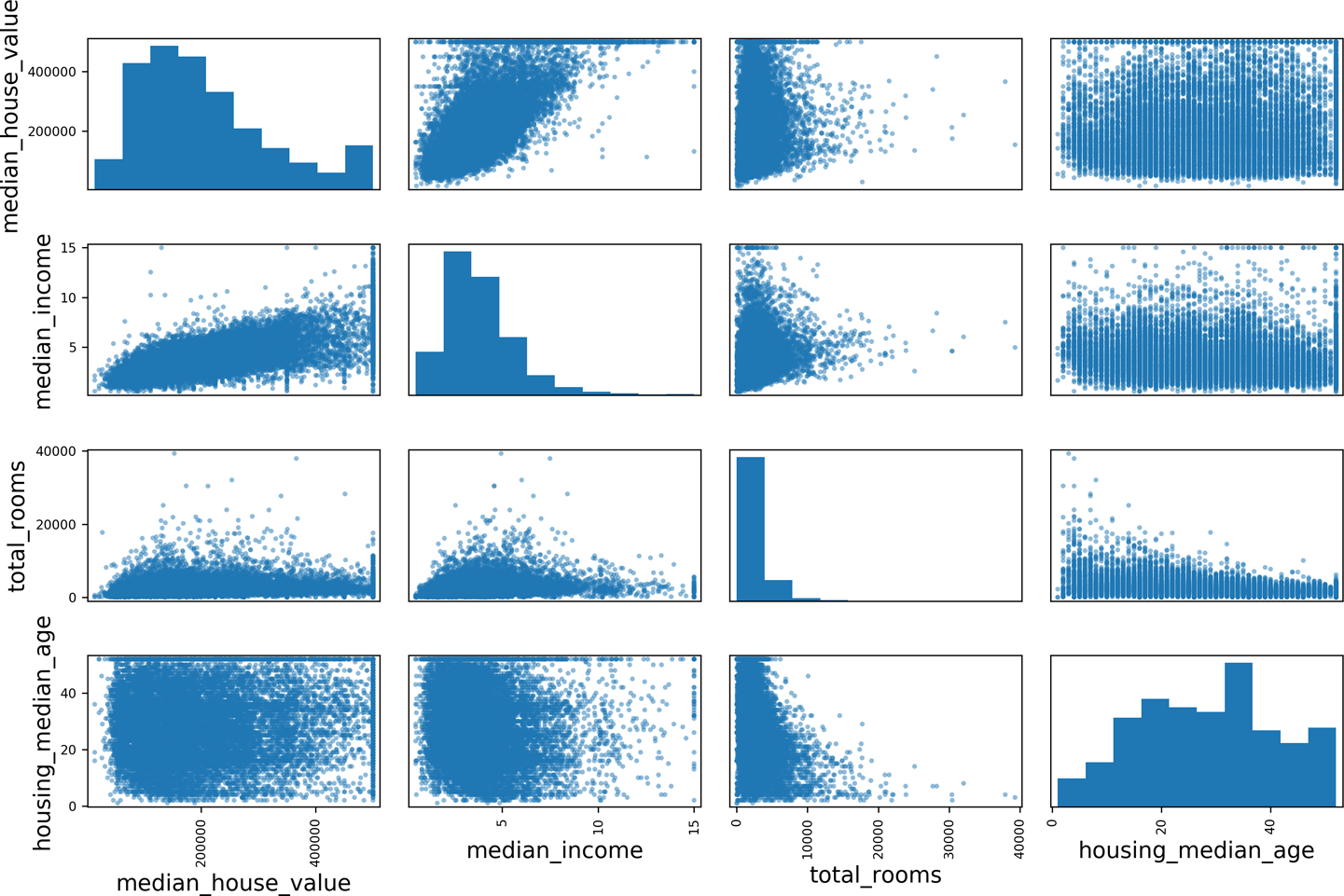
Another way to check for correlation between attributes is to use the **pandas scatter\_matrix() function, which plots every numerical attribute against every other numerical attribute**. Since there are now 11 numerical attributes, you would get 11**2** = 121 plots, which would not fit on a page—so let’s just focus on a few promising attributes that seem most correlated with the median housing value ([Figure 2-15](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch02.html#scatter_matrix_plot)):

from pandas.plotting import scatter\_matrix

attributes = ["median\_house\_value", "median\_income", "total\_rooms",

"housing\_median\_age"]

scatter\_matrix(housing[attributes], figsize=(12, 8))



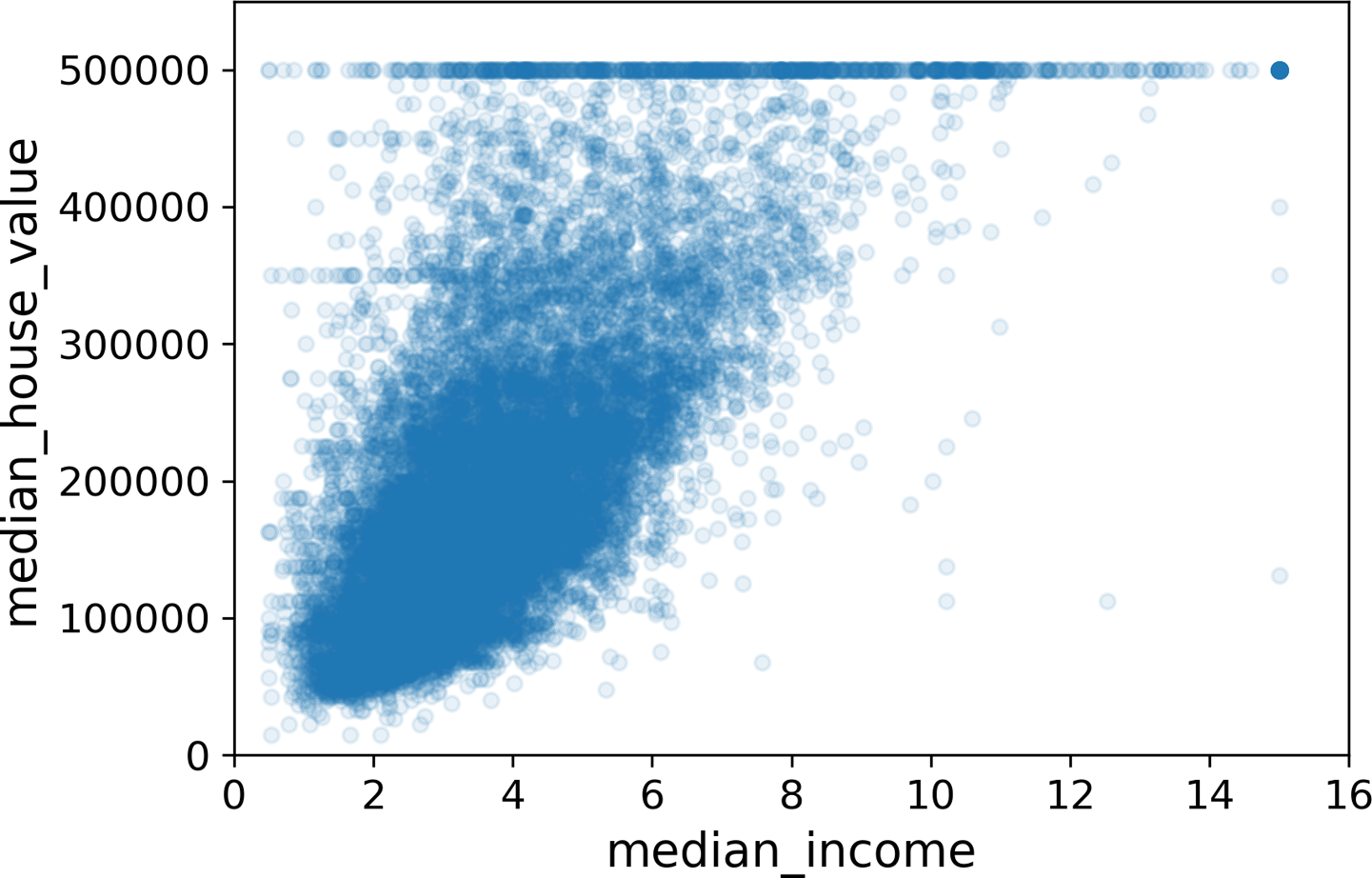
###### Figure 2-15. This scatter matrix plots every numerical attribute against every other numerical attribute, plus a histogram of each numerical attribute

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 displays a histogram of each attribute (other options are available; see the pandas documentation for more details).

The most promising attribute to predict the median house value is the median income, so let’s zoom in on their correlation scatterplot ([Figure 2-16](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch02.html#income_vs_house_value_scatterplot)):

housing.plot(kind="scatter", x="median\_income", y="median\_house\_value",

alpha=0.1)



###### Figure 2-16. Median income versus median house value

**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 this plot reveals other less obvious straight lines: a horizontal line around $450,000, another around $350,000, perhaps one around $280,000, and a few more below that**. You may want to try removing the corresponding districts to prevent your algorithms from learning to reproduce these data quirks.

## 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 distribution, so you may want to transform them (e.g., by computing their logarithm).** Of course, your mileage will vary considerably with each project, but the general ideas are similar.

One last thing you may want to do before preparing the data for Machine Learning algorithms is to try out various attribute combinations. For example, the total number of rooms in a district is not very useful if you don’t know how many households there are. What 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:

housing["rooms\_per\_household"] = housing["total\_rooms"]/housing["households"]

housing["bedrooms\_per\_room"] = housing["total\_bedrooms"]/housing["total\_rooms"]

housing["population\_per\_household"]=housing["population"]/housing["households"]

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

>>> corr\_matrix = housing.corr()

>>> corr\_matrix["median\_house\_value"].sort\_values(ascending=False)

median\_house\_value 1.000000

median\_income 0.687160

rooms\_per\_household 0.146285

total\_rooms 0.135097

housing\_median\_age 0.114110

households 0.064506

total\_bedrooms 0.047689

population\_per\_household -0.021985

population -0.026920

longitude -0.047432

latitude -0.142724

bedrooms\_per\_room -0.259984

Name: median\_house\_value, dtype: float64

**Hey, not bad! The new bedrooms\_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; 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 doing this manually, **you should write functions** for this purpose, 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). Let’s also separate 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):

housing = strat\_train\_set.drop("median\_house\_value", axis=1)

housing\_labels = strat\_train\_set["median\_house\_value"].copy()

## Data Cleaning

Most Machine Learning algorithms cannot work with missing features, so let’s create a few functions to take care of them. We saw 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, the median, etc.).

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

housing.dropna(subset=["total\_bedrooms"]) # option 1

housing.drop("total\_bedrooms", axis=1) # option 2

median = housing["total\_bedrooms"].median() # option 3

housing["total\_bedrooms"].fillna(median, inplace=True)

If you choose option 3, you should compute the median value on the training set and use it to fill the missing values in the training set. Don’t forget to save the median value that you have computed. You will need it later to replace missing values in the 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: SimpleImputer. Here is how to use it. First, you need to create a SimpleImputer instance, specifying that you want to replace each attribute’s missing values with the median of that attribute:

from sklearn.impute import SimpleImputer

imputer = SimpleImputer(strategy="median")

Since the median can only be computed on numerical attributes, you 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 is safer to apply the imputer to all the numerical attributes:

>>> imputer.statistics\_

array([ -118.51 , 34.26 , 29. , 2119.5 , 433. , 1164. , 408. , 3.5409])

>>> housing\_num.median().values

array([ -118.51 , 34.26 , 29. , 2119.5 , 433. , 1164. , 408. , 3.5409])

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

X = imputer.transform(housing\_num)

**The result is a plain NumPy array containing the transformed features. If you want to put it back into a pandas DataFrame, it’s simple**:

housing\_tr = pd.DataFrame(X, columns=housing\_num.columns,

index=housing\_num.index)

##### SCIKIT-LEARN DESIGN

Scikit-Learn’s API is remarkably well designed. These are the [main design principles](https://homl.info/11):[**17**](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch02.html#idm45022182960760)

*Consistency*

All objects share a consistent and simple interface:

*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 needed to guide the estimation process is considered a hyperparameter (such as an imputer’s strategy), and it must be set as an instance variable (generally via a constructor parameter).

*Transformers*

Some estimators (such as an imputer) can also transform a dataset; these are called transformers. Once again, the API is simple: the transformation is performed 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 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).

*Predictors*

Finally, some estimators, given a dataset, are capable of making predictions; they are called predictors. For example, the LinearRegression model in the previous chapter was a predictor: given a country’s GDP per capita, it predicted life satisfaction. 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).[**18**](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch02.html#idm45022182940232)

*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 accessible via public instance variables with an underscore suffix (e.g., imputer.statistics\_).

*Nonproliferation of classes*

Datasets are represented as NumPy arrays or SciPy sparse matrices, instead of homemade classes. Hyperparameters are just regular Python strings or numbers.

*Composition*

Existing building blocks are reused as much as possible. For example, it is easy to create a Pipeline estimator from an arbitrary sequence of transformers followed by a final estimator, as we will see.

*Sensible defaults*

Scikit-Learn provides reasonable default values for most parameters, making it easy to quickly create a baseline working system.