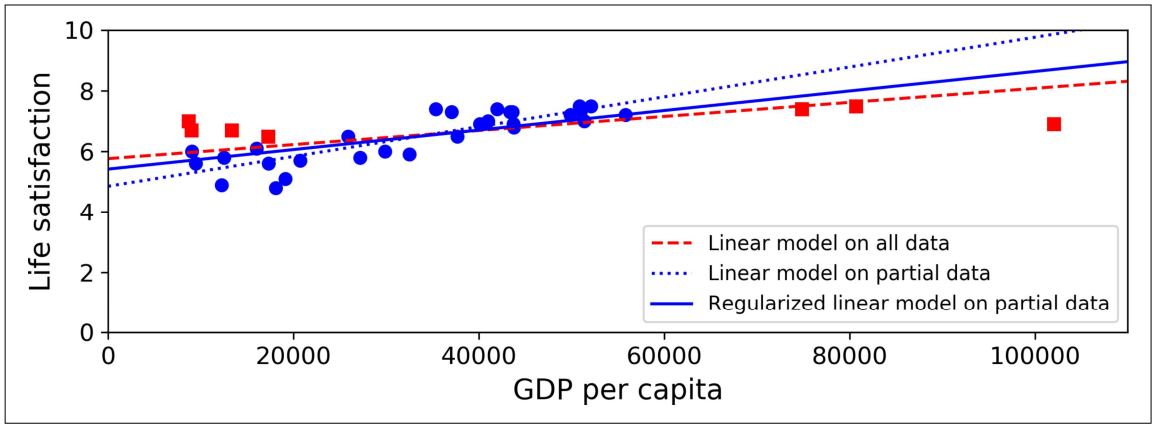


*Figure 1-23. Regularization reduces the risk of overfitting*

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.

The main options to fix this problem are:

- Selecting a more powerful model, with more parameters
- Feeding better features to the learning algorithm (feature engineering)
- Reducing the constraints on the model (e.g., reducing the regularization hyperparameter)

## Stepping Back

By now you already 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, and so on.
- In a 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 comparing them to the learned instances using a similarity measure.
- The system will not perform well if your training set is too small, or if the data is not representative, noisy, or 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.

## 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.

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: that's probably more than enough to get a good estimate of the generalization error.

## Hyperparameter Tuning and Model Selection

So evaluating a model is simple enough: just use a test set. Now suppose you are hesitating between two models (say a linear model and a polynomial model): how can you decide? 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.

So 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 heldout 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, we get a much more accurate measure of its performance. However, there is a drawback: the training time is multiplied by the number of validation sets.

## Data Mismatch

In some cases, it is easy to get a large amount of data for training, but it is not 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 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). After training your model on the web pictures, if you observe that the performance of your 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 part 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, so if performs poorly on the validation set, the problem must come from the data mismatch. You can try to tackle this problem by preprocessing 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 the model 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, as discussed earlier.

### 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. However, 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

11 "The Lack of A Priori Distinctions Between Learning Algorithms," D. Wolpert (1996).

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 you 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

In this chapter we have covered some of the most important concepts in Machine Learning. In the next chapters we will dive deeper and write more code, but before we do, make sure you know how to answer the following questions:

1. How would you define Machine Learning?
2. Can you name four types of problems where it shines?
3. What is a labeled training set?
4. What are the two most common supervised tasks?
5. Can you name four common unsupervised tasks?
6. What type of Machine Learning algorithm would you use to allow a robot to walk in various unknown terrains?
7. What type of algorithm would you use to segment your customers into multiple groups?
8. Would you frame the problem of spam detection as a supervised learning problem or an unsupervised learning problem?
9. What is an online learning system?
10. What is out-of-core learning?
11. What type of learning algorithm relies on a similarity measure to make predictions?
12. What is the difference between a model parameter and a learning algorithm's hyperparameter?
13. What do model-based learning algorithms search for? What is the most common strategy they use to succeed? How do they make predictions?
14. Can you name four of the main challenges in Machine Learning?
15. If your model performs great on the training data but generalizes poorly to new instances, what is happening? Can you name three possible solutions?
16. What is a test set and why would you want to use it?

17. What is the purpose of a validation set?
18. What can go wrong if you tune hyperparameters using the test set?
19. What is repeated cross-validation and why would you prefer it to using a single validation set?

Solutions to these exercises are available in ???.

CHAPTER 2

# End-to-End Machine Learning Project

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as he or she writes—so you can take advantage of these technologies long before the official release of these titles. The following will be Chapter 2 in the final release of the book.

In this chapter, you will go through an example project end to end, pretending to be a recently hired data scientist in a real estate company.[1] 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.

1 The example project is completely fictitious; the goal is just to illustrate the main steps of a Machine Learning project, not to learn anything about the real estate business.

## Working with Real Data

When you are learning about Machine Learning it is best to actually experiment with real-world data, not just 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:

- Popular open data repositories:
  - UC Irvine Machine Learning Repository
  - Kaggle datasets
  - Amazon's AWS datasets
- Meta portals (they list open data repositories):
  - *http://dataportals.org/*
  - *http://opendatamonitor.eu/*
  - *http://quandl.com/*
- Other pages listing many popular open data repositories:
  - Wikipedia's list of Machine Learning datasets
  - Quora.com question
  - Datasets subreddit

In this chapter we chose the California Housing Prices dataset from the StatLib repository[2] (see Figure 2-1). This dataset was based on data from the 1990 California census. It is not exactly recent (you could still afford a nice house in the Bay Area at the time), but it has many qualities for learning, so we will pretend it is recent data. We also added a categorical attribute and removed a few features for teaching purposes.

2 The original dataset appeared in R. Kelley Pace and Ronald Barry, "Sparse Spatial Autoregressions," *Statistics & Probability Letters* 33, no. 3 (1997): 291–297.

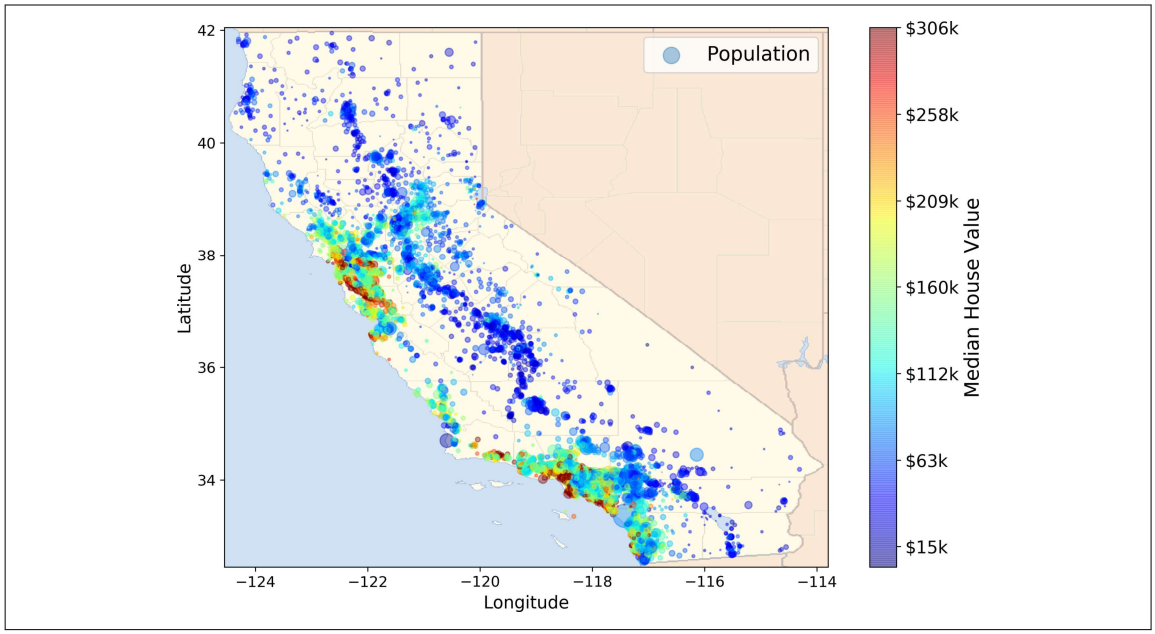


*Figure 2-1. California housing prices*

# Look at the Big Picture

Welcome to Machine Learning Housing Corporation! The first task you are asked to perform is to build a model of housing prices in California using the California census data. This data has metrics such as the population, median income, median housing price, and so on 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 just 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.

Since you are a well-organized data scientist, the first thing you do is to pull out your Machine Learning project checklist. You can start with the one in ???; 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 is the business objective; building a model is probably not the end goal. How does the company expect to use and benefit

from this model? This is important because it will determine how you frame the problem, what algorithms you will select, what performance measure you will use to evaluate your model, and how much effort you should 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), along with many other *signals*.[3] 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.

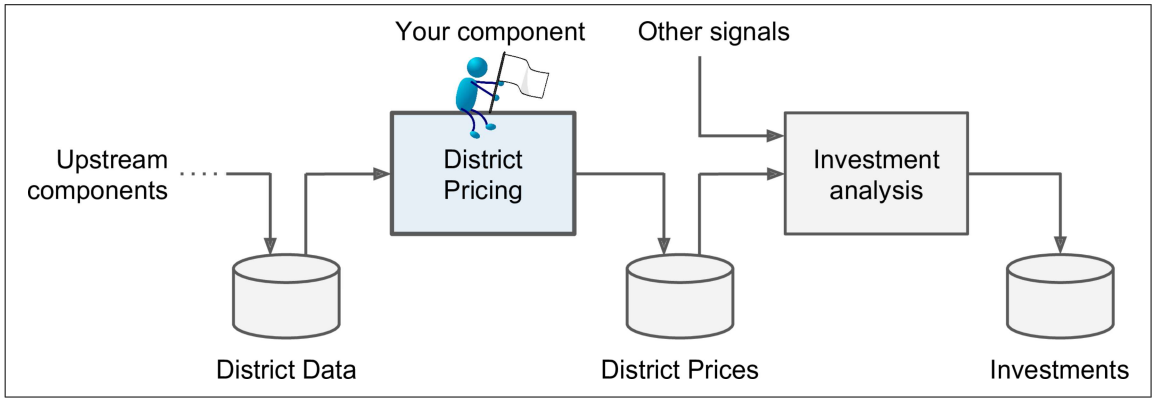


*Figure 2-2. A Machine Learning pipeline for real estate investments*

## 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, and then some time later the next component in the pipeline pulls this data and spits out its own output, and so on. Each component is fairly self-contained: the interface between components is simply the data store. This makes the system quite 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.

3 A piece of information fed to a Machine Learning system is often called a *signal* in reference to Shannon's information theory: you want a high signal/noise ratio.

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 is what the current solution looks like (if any). It will often give you a reference 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.

Okay, 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 *labeled* training examples (each instance comes with the expected output, i.e., the district's median housing price). Moreover, 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 in 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.

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

## 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 shows the mathematical formula to compute the RMSE.

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

$$\text{RMSE}(\mathbf{X}, h) = \sqrt{\frac{1}{m}\sum_{i=1}^{m}\left(h\left(\mathbf{x}^{(i)}\right) - y^{(i)}\right)^2}$$

## Notations

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

- $m$ is the number of instances in the dataset you are measuring the RMSE on.
  - For example, if you are evaluating the RMSE on a validation set of 2,000 districts, then $m = 2{,}000$.
- $\mathbf{x}^{(i)}$ 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).
  - 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:

$$\mathbf{x}^{(1)} = \begin{pmatrix} -118.29 \\ 33.91 \\ 1{,}416 \\ 38{,}372 \end{pmatrix}$$

  and:

$$y^{(1)} = 156{,}400$$

- $\mathbf{X}$ 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 $\mathbf{x}^{(i)}$, noted $(\mathbf{x}^{(i)})^T$.[4]
  - For example, if the first district is as just described, then the matrix $\mathbf{X}$ looks like this:

$$\mathbf{X} = \begin{pmatrix} \left(\mathbf{x}^{(1)}\right)^T \\ \left(\mathbf{x}^{(2)}\right)^T \\ \vdots \\ \left(\mathbf{x}^{(1999)}\right)^T \\ \left(\mathbf{x}^{(2000)}\right)^T \end{pmatrix} = \begin{pmatrix} -118.29 & 33.91 & 1{,}416 & 38{,}372 \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

4 Recall that the transpose operator flips a column vector into a row vector (and vice versa).

- $h$ is your system's prediction function, also called a *hypothesis*. When your system is given an instance's feature vector $\mathbf{x}^{(i)}$, it outputs a predicted value $\hat{y}^{(i)} = h(\mathbf{x}^{(i)})$ for that instance ($\hat{y}$ is pronounced "y-hat").
  - For example, if your system predicts that the median housing price in the first district is \$158,400, then $\hat{y}^{(1)} = h(\mathbf{x}^{(1)}) = 158{,}400$. The prediction error for this district is $\hat{y}^{(1)} - y^{(1)} = 2{,}000$.
- RMSE($\mathbf{X}$,$h$) 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 $\mathbf{x}^{(i)}$), and uppercase bold font for matrices (such as $\mathbf{X}$).

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* (also called the Average Absolute Deviation; see Equation 2-2):

*Equation 2-2. Mean Absolute Error*

$$\text{MAE}(\mathbf{X}, h) = \frac{1}{m}\sum_{i=1}^{m}\left|h\left(\mathbf{x}^{(i)}\right) - y^{(i)}\right|$$

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:

- Computing the root of a sum of squares (RMSE) corresponds to the *Euclidean norm*: it is the notion of distance you are familiar with. It is also called the $\ell_2$ *norm*, noted $\|\cdot\|_2$ (or just $\|\cdot\|$).
- Computing the sum of absolutes (MAE) corresponds to the $\ell_1$ *norm*, noted $\|\cdot\|_1$. It 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.
- More generally, the $\ell_k$ *norm* of a vector $\mathbf{v}$ containing $n$ elements is defined as $\|\mathbf{v}\|_k = \left(|v_0|^k + |v_1|^k + \cdots + |v_n|^k\right)^{\frac{1}{k}}$. $\ell_0$ just gives the number of non-zero elements in the vector, and $\ell_\infty$ gives the maximum absolute value in the vector.
- 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 were made so far (by you or others); this can 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 we assume that these prices are going to be used as such. But what if the downstream system actually 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!

## Get the Data

It's time to get your hands dirty. Don't hesitate to pick up your laptop and walk through the following code examples in a Jupyter notebook. The full Jupyter notebook is available at *https://github.com/ageron/handson-ml2*.

### Create the Workspace

First you will need to have Python installed. It is probably already installed on your system. If not, you can get it at *https://www.python.org/*.[5]

Next you need to create a workspace directory for your Machine Learning code and datasets. Open a terminal and type the following commands (after the `$` prompts):

```
$ export ML_PATH="$HOME/ml"      # You can change the path if you prefer
$ mkdir -p $ML_PATH
```

You will need a number of Python modules: Jupyter, NumPy, Pandas, Matplotlib, and Scikit-Learn. If you already have Jupyter running with all these modules installed, you can safely skip to "Download the Data" on page 49. If you don't have them yet, there are many ways to install them (and their dependencies). You can use your sys-

5 The latest version of Python 3 is recommended. Python 2.7+ may work too, but it is now deprecated, all major scientific libraries are dropping support for it, so you should migrate to Python 3 as soon as possible.

tem's packaging system (e.g., apt-get on Ubuntu, or MacPorts or HomeBrew on MacOS), install a Scientific Python distribution such as Anaconda and use its packaging system, or just use Python's own packaging system, pip, which is included by default with the Python binary installers (since Python 2.7.9).[6] You can check to see if pip is installed by typing the following command:

```
$ python3 -m pip --version
pip 19.0.2 from [...]/lib/python3.6/site-packages (python 3.6)
```

You should make sure you have a recent version of pip installed. To upgrade the pip module, type:[7]

```
$ python3 -m pip install --user -U pip
Collecting pip
[...]
Successfully installed pip-19.0.2
```

## Creating an Isolated Environment

If you would like to work in an isolated environment (which is strongly recommended so you can work on different projects without having conflicting library versions), install virtualenv[8] by running the following pip command (again, if you want virtualenv to be installed for all users on your machine, remove `--user` and run this command with administrator rights):

```
$ python3 -m pip install --user -U virtualenv
Collecting virtualenv
[...]
Successfully installed virtualenv
```

Now you can create an isolated Python environment by typing:

```
$ cd $ML_PATH
$ virtualenv env
Using base prefix '[...]'
New python executable in [...]/ml/env/bin/python3.6
Also creating executable in [...]/ml/env/bin/python
Installing setuptools, pip, wheel...done.
```

6 We will show the installation steps using pip in a bash shell on a Linux or MacOS system. You may need to adapt these commands to your own system. On Windows, we recommend installing Anaconda instead.

7 If you want to upgrade pip for all users on your machine rather than just your own user, you should remove the `--user` option and make sure you have administrator rights (e.g., by adding `sudo` before the whole command on Linux or MacOSX).

8 Alternative tools include venv (very similar to virtualenv and included in the standard library), virtualenvwrapper (provides extra functionalities on top of virtualenv), pyenv (allows easy switching between Python versions), and pipenv (a great packaging tool by the same author as the popular `requests` library, built on top of pip, virtualenv and more).

Now every time you want to activate this environment, just open a terminal and type:

```
$ cd $ML_PATH
$ source env/bin/activate # on Linux or MacOSX
$ .\env\Scripts\activate  # on Windows
```

To deactivate this environment, just type `deactivate`. While the environment is active, any package you install using pip will be installed in this isolated environment, and Python will only have access to these packages (if you also want access to the system's packages, you should create the environment using virtualenv's `--system-site-packages` option). Check out virtualenv's documentation for more information.

Now you can install all the required modules and their dependencies using this simple pip command (if you are not using a virtualenv, you will need the `--user` option or administrator rights):

```
$ python3 -m pip install -U jupyter matplotlib numpy pandas scipy scikit-learn
Collecting jupyter
  Downloading jupyter-1.0.0-py2.py3-none-any.whl
Collecting matplotlib
  [...]
```

To check your installation, try to import every module like this:

```
$ python3 -c "import jupyter, matplotlib, numpy, pandas, scipy, sklearn"
```

There should be no output and no error. Now you can fire up Jupyter by typing:

```
$ jupyter notebook
[I 15:24 NotebookApp] Serving notebooks from local directory: [...]/ml
[I 15:24 NotebookApp] 0 active kernels
[I 15:24 NotebookApp] The Jupyter Notebook is running at: http://localhost:8888/
[I 15:24 NotebookApp] Use Control-C to stop this server and shut down all
kernels (twice to skip confirmation).
```

A Jupyter server is now running in your terminal, listening to port 8888. You can visit this server by opening your web browser to *http://localhost:8888/* (this usually happens automatically when the server starts). You should see your empty workspace directory (containing only the *env* directory if you followed the preceding virtualenv instructions).

Now create a new Python notebook by clicking on the New button and selecting the appropriate Python version[9] (see Figure 2-3).

This does three things: first, it creates a new notebook file called *Untitled.ipynb* in your workspace; second, it starts a Jupyter Python kernel to run this notebook; and

9 Note that Jupyter can handle multiple versions of Python, and even many other languages such as R or Octave.

third, it opens this notebook in a new tab. You should start by renaming this notebook to "Housing" (this will automatically rename the file to *Housing.ipynb*) by clicking Untitled and typing the new name.

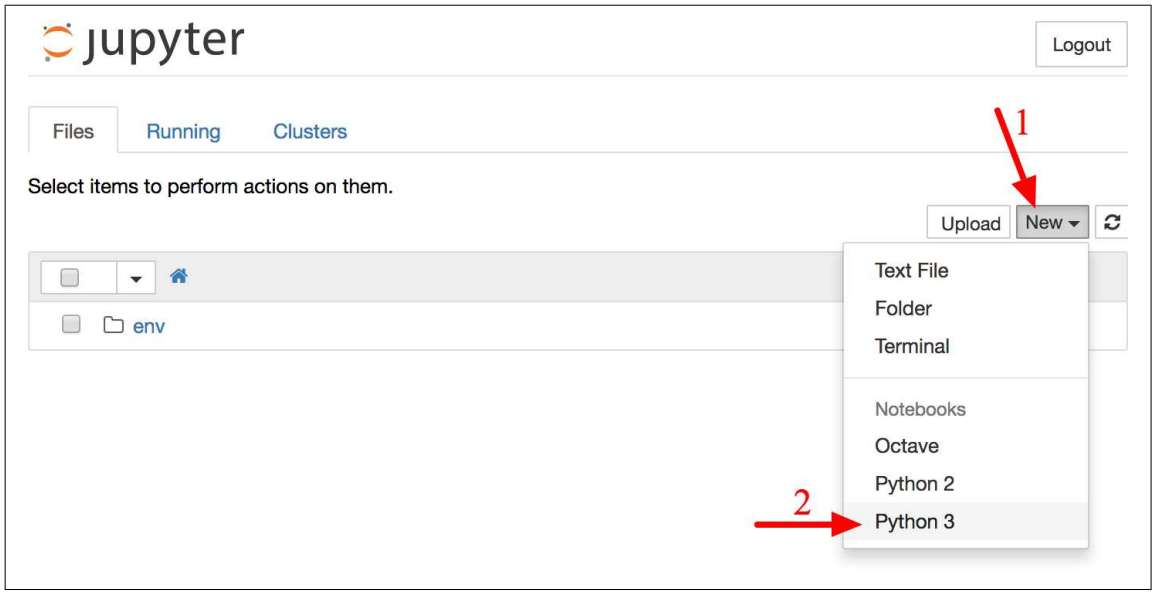


*Figure 2-3. Your workspace in Jupyter*

A notebook contains a list of cells. Each cell can contain executable code or formatted text. Right now the notebook contains only one empty code cell, labeled "In [1]:". Try typing **`print("Hello world!")`** in the cell, and click on the play button (see Figure 2-4) or press Shift-Enter. This sends the current cell to this notebook's Python kernel, which runs it and returns the output. The result is displayed below the cell, and since we reached the end of the notebook, a new cell is automatically created. Go through the User Interface Tour from Jupyter's Help menu to learn the basics.

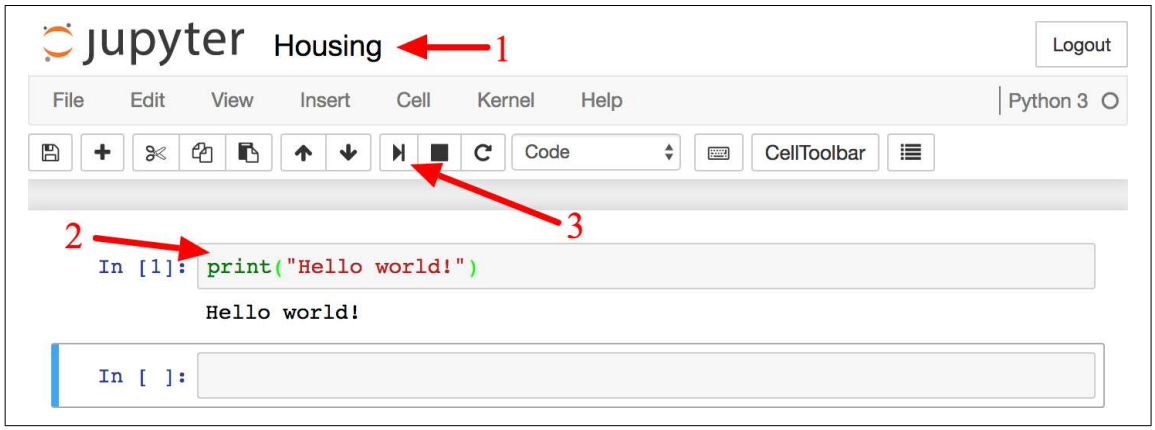


*Figure 2-4. Hello world Python notebook*

## Download the Data

In typical environments your data would be available in a relational database (or some other common datastore) and spread across multiple tables/documents/files. To access it, you would first need to get your credentials and access authorizations,[10] and familiarize yourself with the data schema. In this project, however, things are much simpler: you will just download a single compressed file, *housing.tgz*, which contains a comma-separated value (CSV) file called *housing.csv* with all the data.

You could use your web browser to download it, and run `tar xzf housing.tgz` to decompress the file and extract the CSV file, but it is preferable to create a small function to do that. It is useful in particular if data changes regularly, as it allows you to write a small script that you can run whenever you need to fetch the latest data (or you can set up a scheduled job to do that automatically at regular intervals). Automating the process of fetching the data is also useful if you need to install the dataset on multiple machines.

Here is the function to fetch the data:[11]

```
import os
import tarfile
from six.moves import urllib

DOWNLOAD_ROOT = "https://raw.githubusercontent.com/ageron/handson-ml2/master/"
HOUSING_PATH = os.path.join("datasets", "housing")
HOUSING_URL = DOWNLOAD_ROOT + "datasets/housing/housing.tgz"

def fetch_housing_data(housing_url=HOUSING_URL, housing_path=HOUSING_PATH):
    if not os.path.isdir(housing_path):
        os.makedirs(housing_path)
    tgz_path = os.path.join(housing_path, "housing.tgz")
    urllib.request.urlretrieve(housing_url, tgz_path)
    housing_tgz = tarfile.open(tgz_path)
    housing_tgz.extractall(path=housing_path)
    housing_tgz.close()
```

Now when you call `fetch_housing_data()`, it creates a *datasets/housing* directory in your workspace, downloads the *housing.tgz* file, and extracts the *housing.csv* from it in this directory.

Now let's load the data using Pandas. Once again you should write a small function to load the data:

10 You might also need to check legal constraints, such as private fields that should never be copied to unsafe datastores.

11 In a real project you would save this code in a Python file, but for now you can just write it in your Jupyter notebook.

```
import pandas as pd

def load_housing_data(housing_path=HOUSING_PATH):
    csv_path = os.path.join(housing_path, "housing.csv")
    return pd.read_csv(csv_path)
```

This function returns a Pandas DataFrame object containing all the data.

## Take a Quick Look at the Data Structure

Let's take a look at the top five rows using the DataFrame's `head()` method (see Figure 2-5).

```
In [5]: housing = load_housing_data()
        housing.head()
```

Out[5]:

| | longitude | latitude | housing_median_age | total_rooms | total_bedrooms | populatio |
|---|---|---|---|---|---|---|
| 0 | -122.23 | 37.88 | 41.0 | 880.0 | 129.0 | 322.0 |
| 1 | -122.22 | 37.86 | 21.0 | 7099.0 | 1106.0 | 2401.0 |
| 2 | -122.24 | 37.85 | 52.0 | 1467.0 | 190.0 | 496.0 |
| 3 | -122.25 | 37.85 | 52.0 | 1274.0 | 235.0 | 558.0 |
| 4 | -122.25 | 37.85 | 52.0 | 1627.0 | 280.0 | 565.0 |

*Figure 2-5. Top five rows in the dataset*

Each row represents one district. There are 10 attributes (you can see the first 6 in the screenshot): `longitude`, `latitude`, `housing_median_age`, `total_rooms`, `total_bedrooms`, `population`, `households`, `median_income`, `median_house_value`, and `ocean_proximity`.

The `info()` method is useful to get a quick description of the data, in particular the total number of rows, and each attribute's type and number of non-null values (see Figure 2-6).