机器学习简介

机器学习概览

什么是机器学习？

机器学习是通过编程让计算机从数据中进行学习的科学（和艺术）。

更广义的概念：

机器学习是让计算机具有学习的能力，无需进行明确编程。 — 亚瑟·萨缪尔，1959

工程化的概念：

计算机程序利用经验 E 学习任务 T，性能是 P，如果针对任务 T 的性能 P 随着经验 E 不断增长，则称为机器学习。 — 汤姆·米切尔，1997

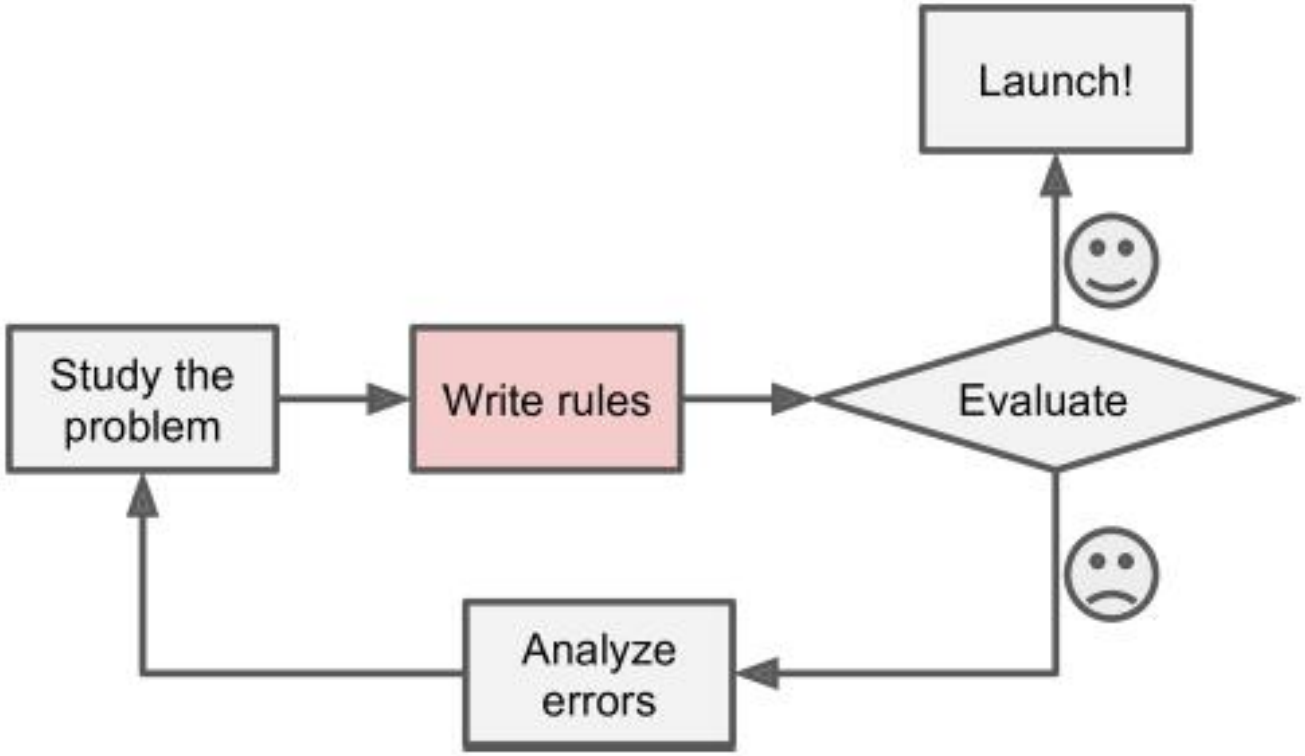
机器学习概览

例如，垃圾邮件过滤器就是一个机器学习程序，它可以根据垃圾邮件（比如，用户标记的垃圾邮件spam）和普通邮件（非垃圾邮件，也称作ham）学习标记垃圾邮件。

用来进行学习的样例称作**训练集**。每个训练样例称作训练实例（或样本）。在这个例子中，任务 T 就是标记新邮件是否是垃圾邮件，经验E是训练数据，性能P需要定义：例如，可以使用正确分类的比例。这个性能指标称为**准确率**，通常用在分类任务中。

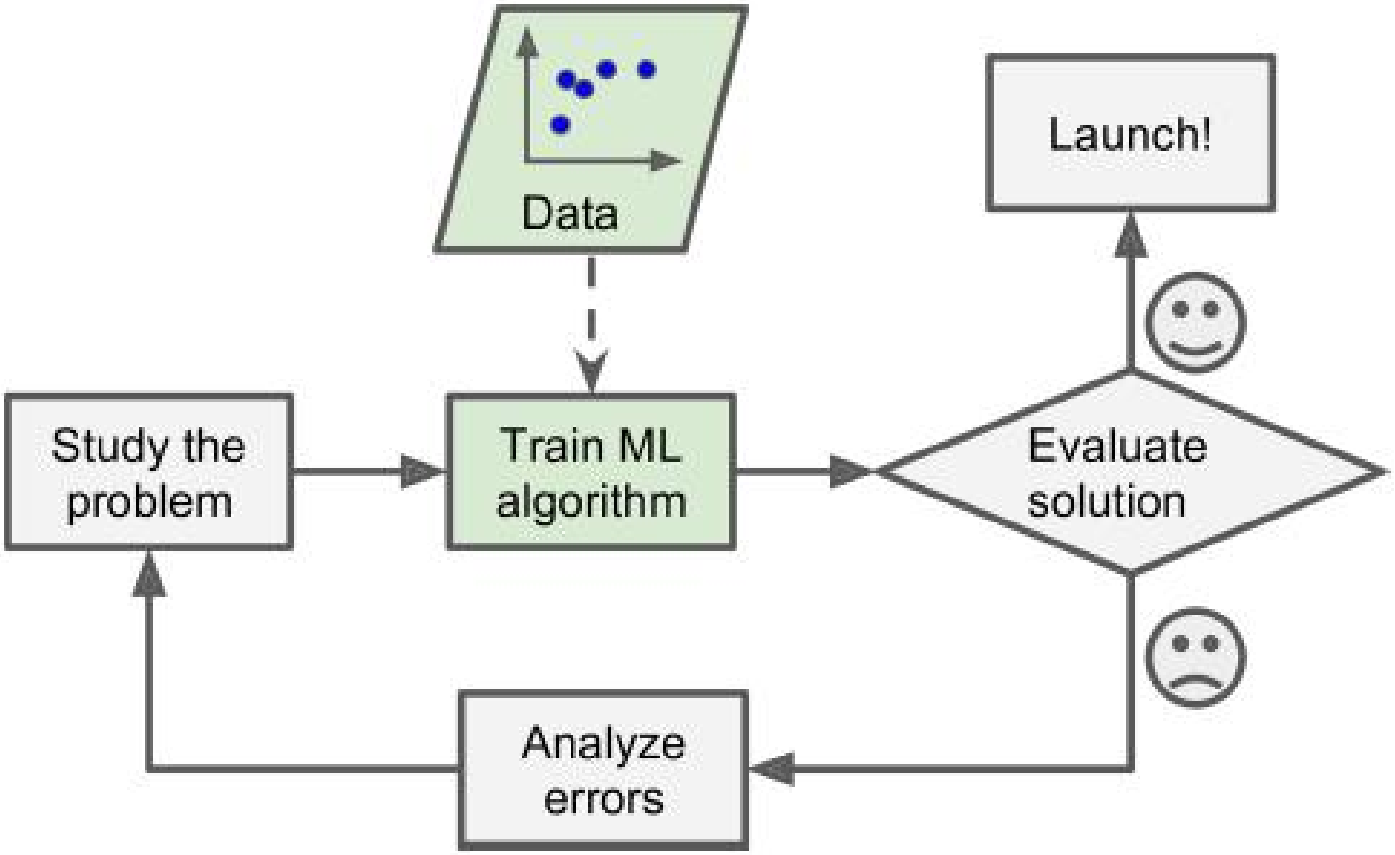
机器学习概览

传统方法


Launch!
Study the problem
Write rules
Evaluate
Analyze errors

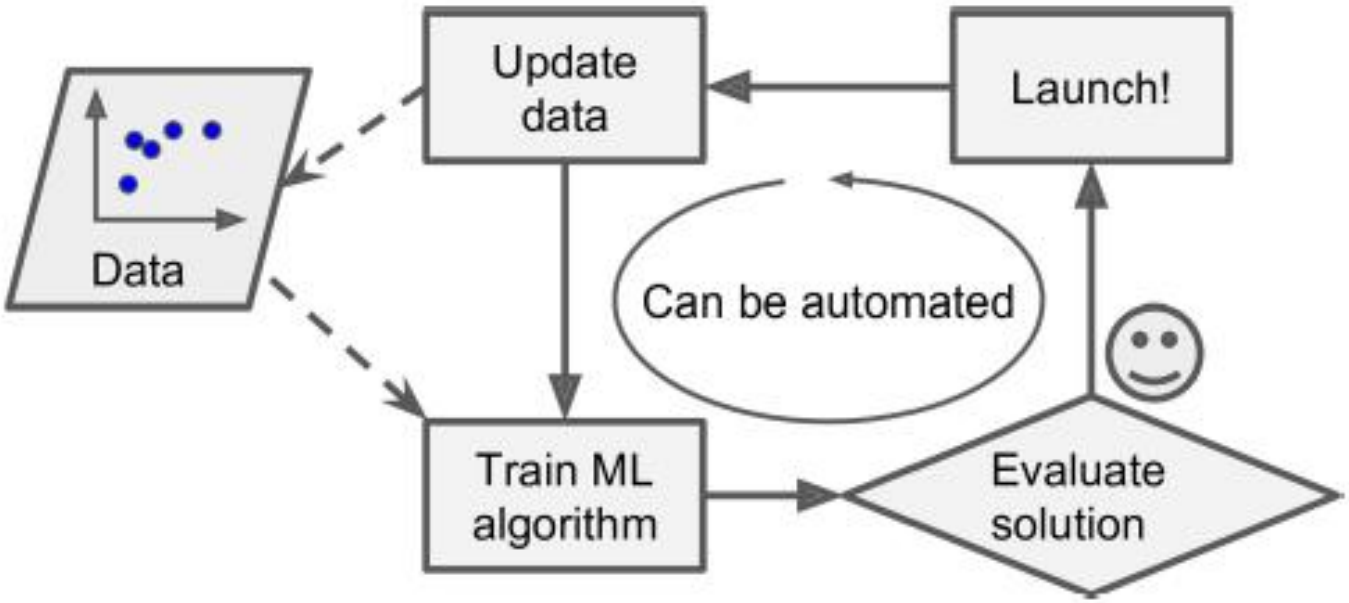
机器学习概览

机器学习方法



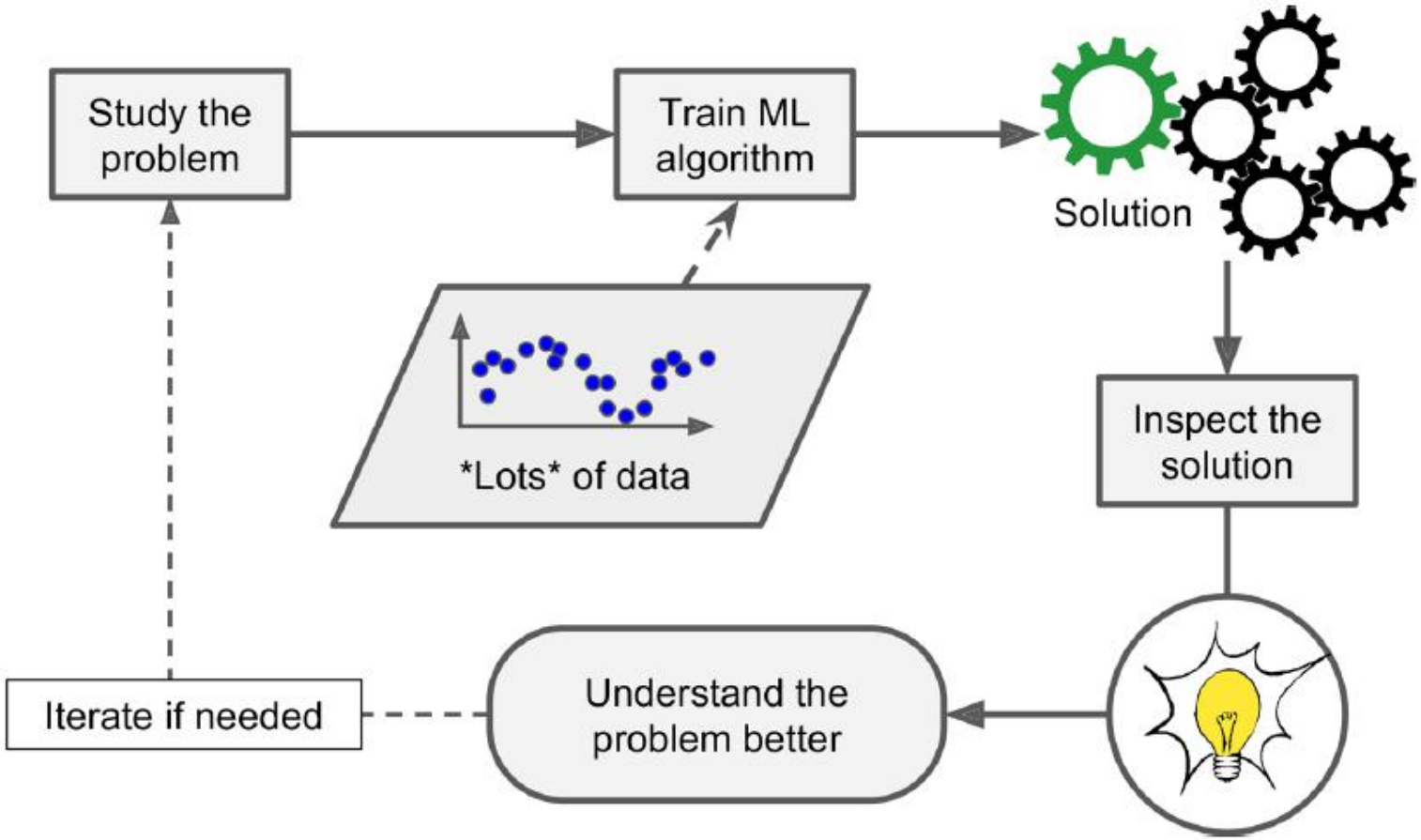
机器学习概览

机器学习方法 (适应性)



机器学习概览

机器学习方法 (与人类交互)



机器学习概览

机器学习可以用于：

- 需要进行大量手工调整或需要拥有长串规则才能解决的问题：机器学习算法通常可以简化代码、提高性能。
- 问题复杂，传统方法难以解决：最好的机器学习方法可以找到解决方案。
- 环境有变化：机器学习算法可以适应新数据
- 。
- 洞察复杂问题和大量数据。

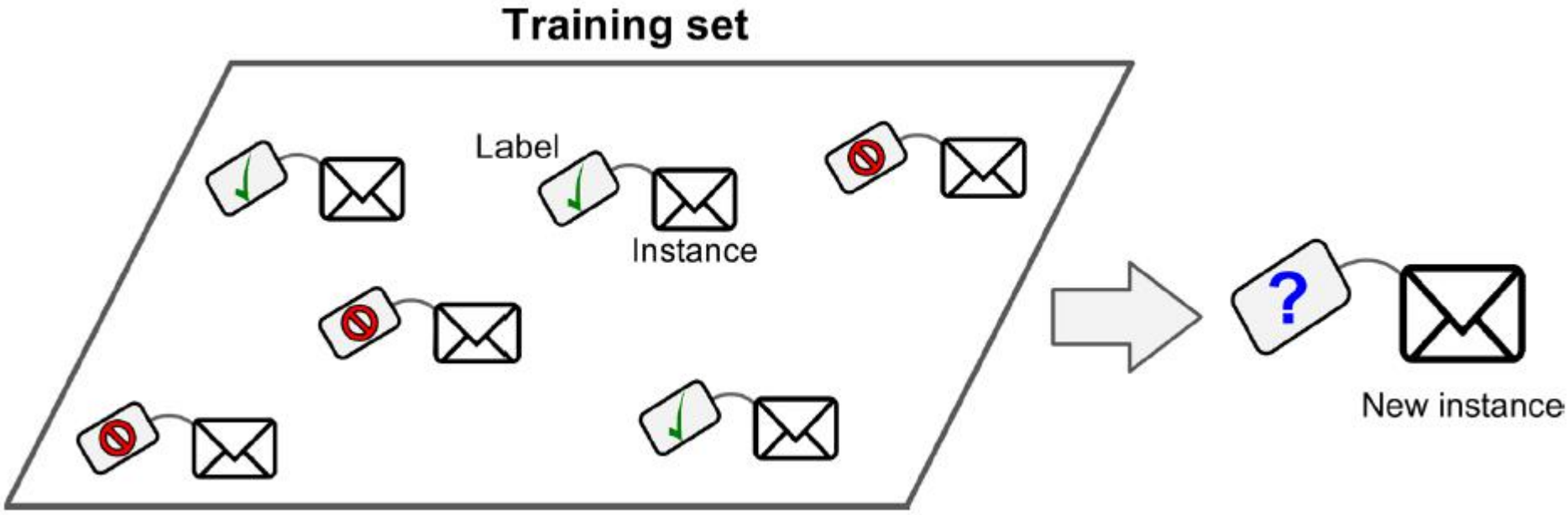
机器学习系统的类型

机器学习有多种类型，可以根据如下规则进行分类：

- 是否在人类监督下进行训练（监督，非监督，半监督和强化学习）
- 是否可以动态渐进学习（在线学习 vs 批量学习）
- 它们是否只是通过简单地比较新的数据点和已知的数据点，还是在训练数据中进行模式识别，以建立一个预测模型（基于实例学习 vs 基于模型学习）

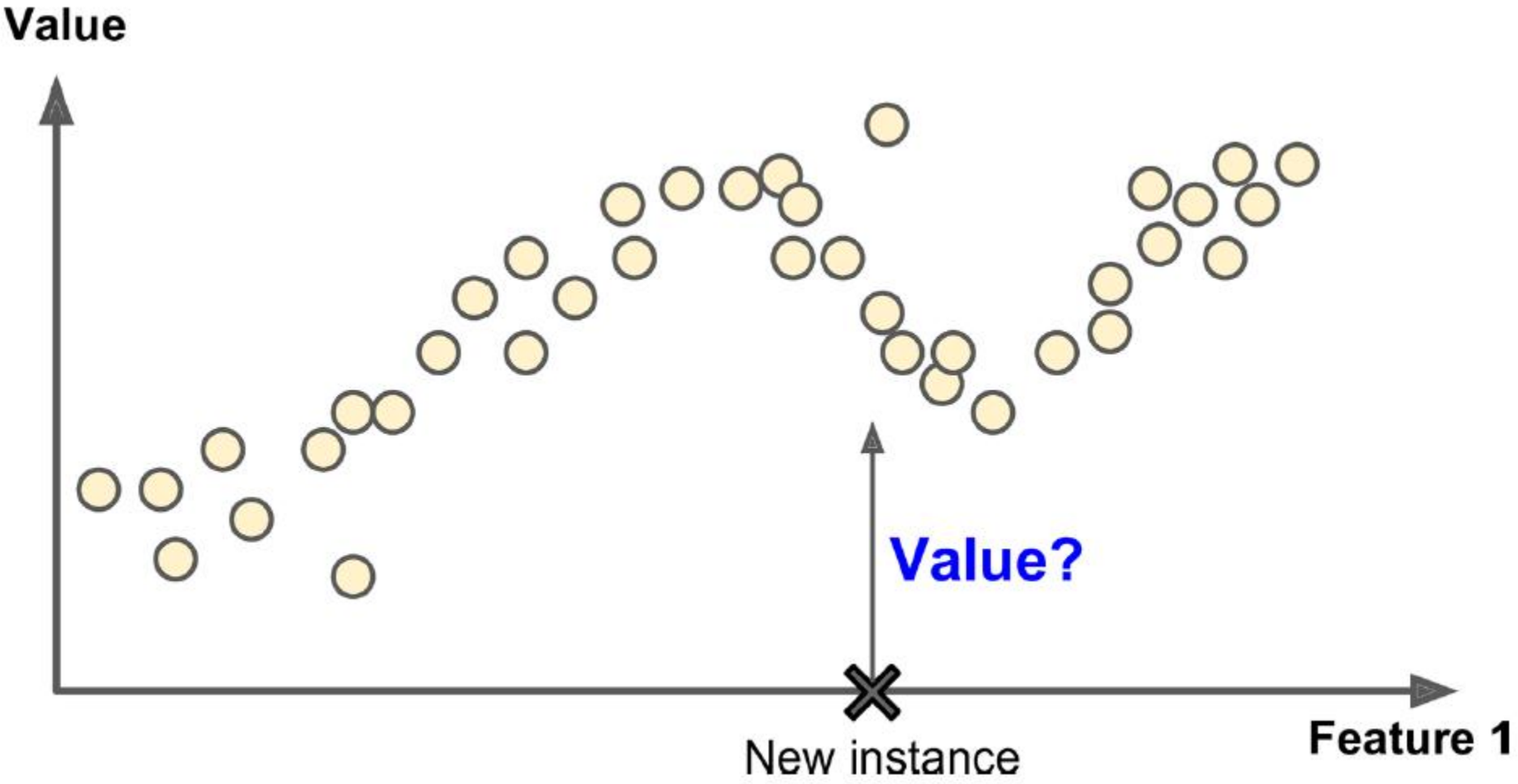
监督/非监督学习

- 监督学习 (classification)



监督/非监督学习

- 监督学习 (regression)



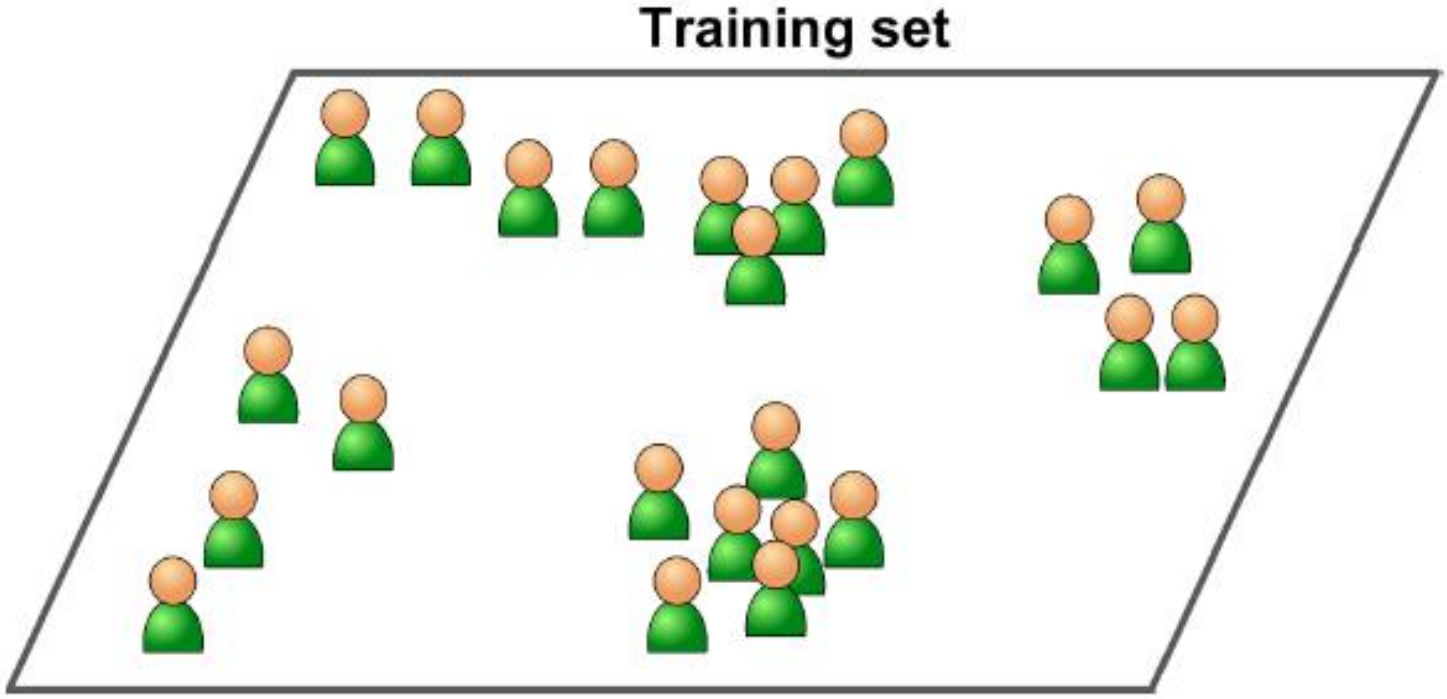
监督/非监督学习

下面是一些重要的监督学习算法：

- K近邻算法
- 线性回归
- 逻辑回归
- 支持向量机（SVM）
- 决策树和随机森林
- 神经网络

监督/非监督学习

- 非监督学习 (Clustering)



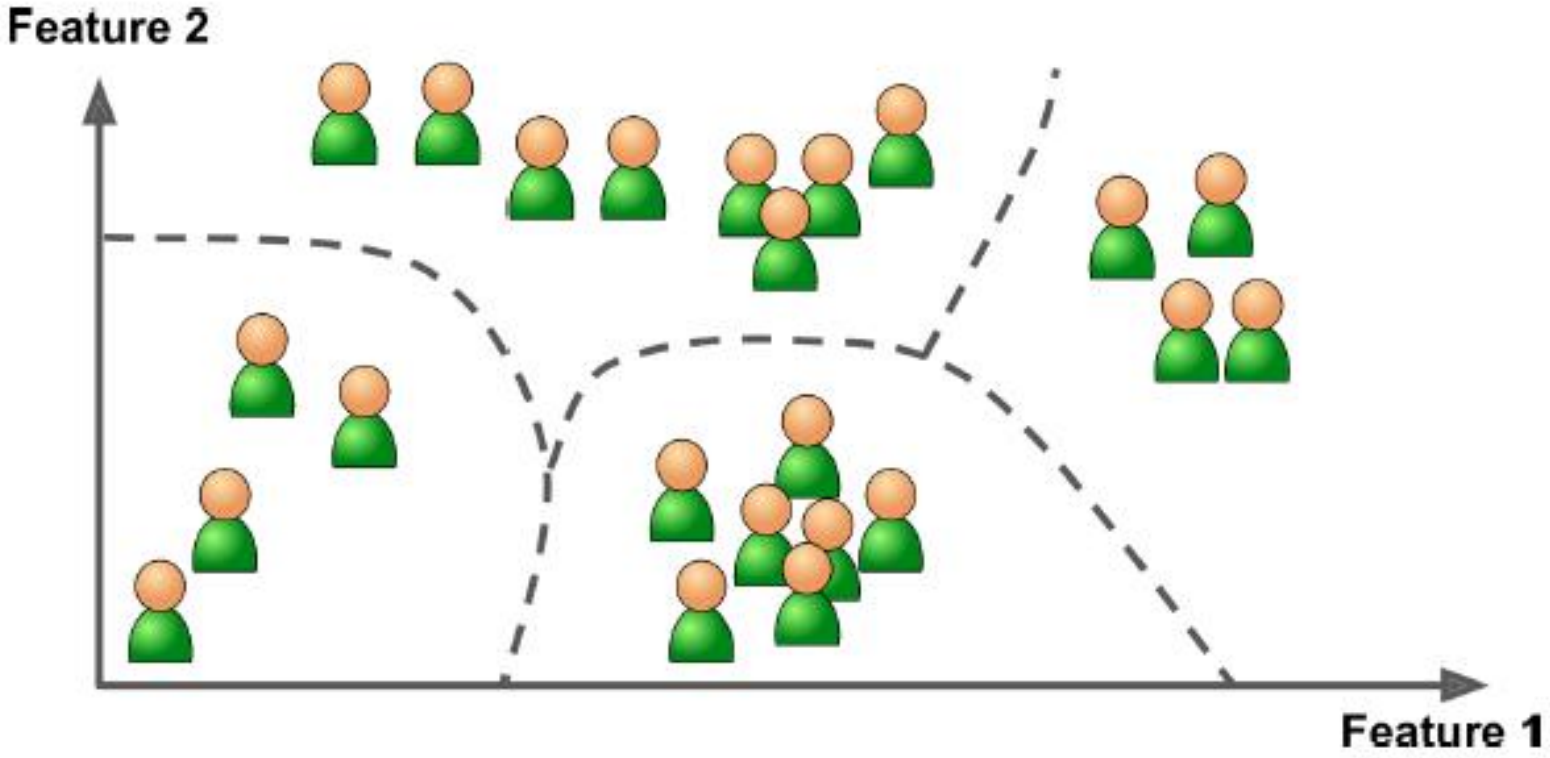
监督/非监督学习

下面是一些最重要的非监督学习算法：

- 聚类
  - K 均值
  - 层次聚类分析（Hierarchical Cluster Analysis，HCA）
  - 最大期望法
- 可视化和降维
  - 主成分分析（Principal Component Analysis，PCA）
  - 核主成分分析
  - 局部线性嵌入（Locally-Linear Embedding，LLE）
  - t-分布邻域嵌入算法（t-distributed Stochastic Neighbor Embedding，t-SNE）
- 关联性规则学习
  - Apriori 算法
  - Eclat 算法

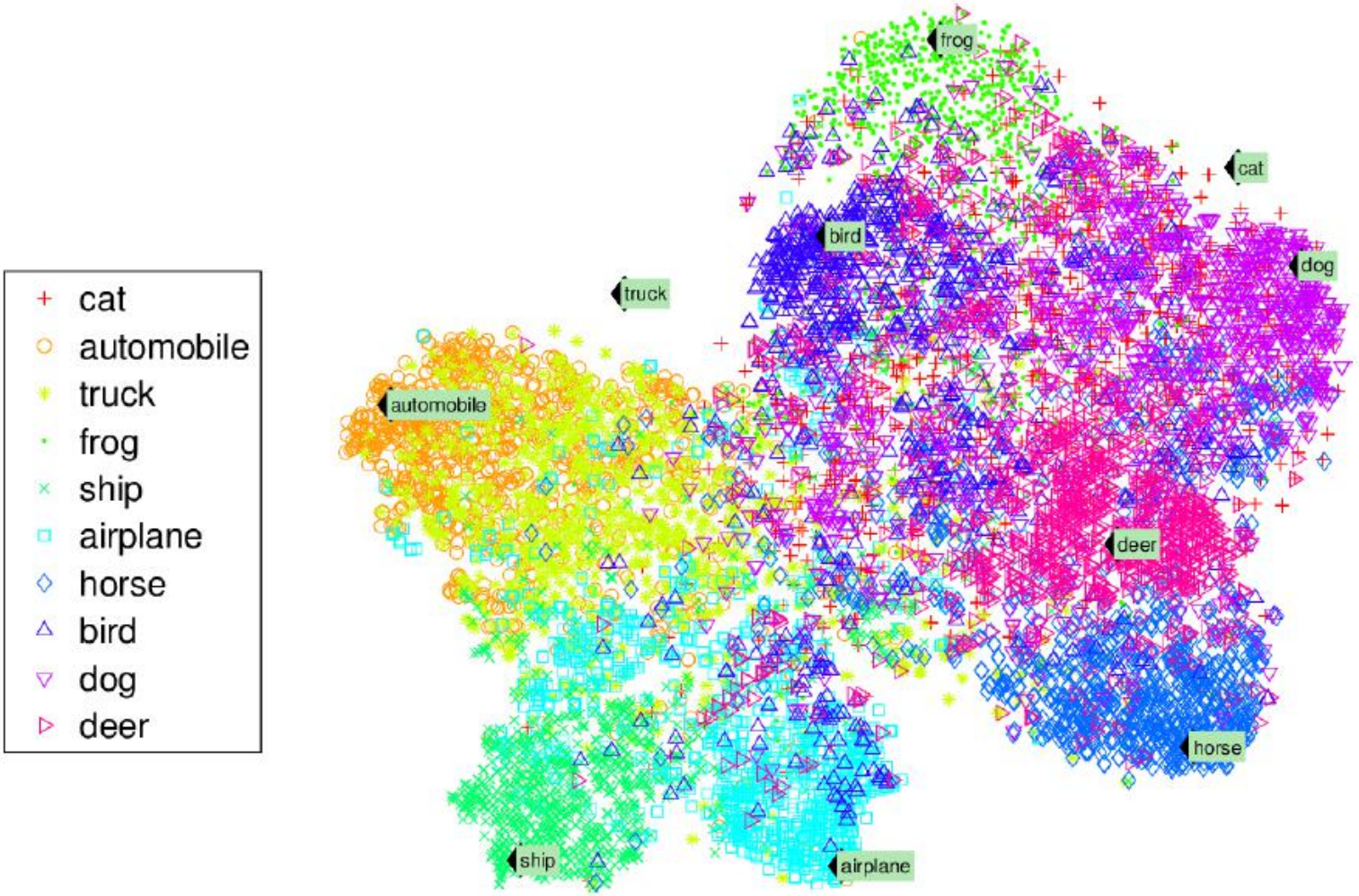
监督/非监督学习

- Clustering

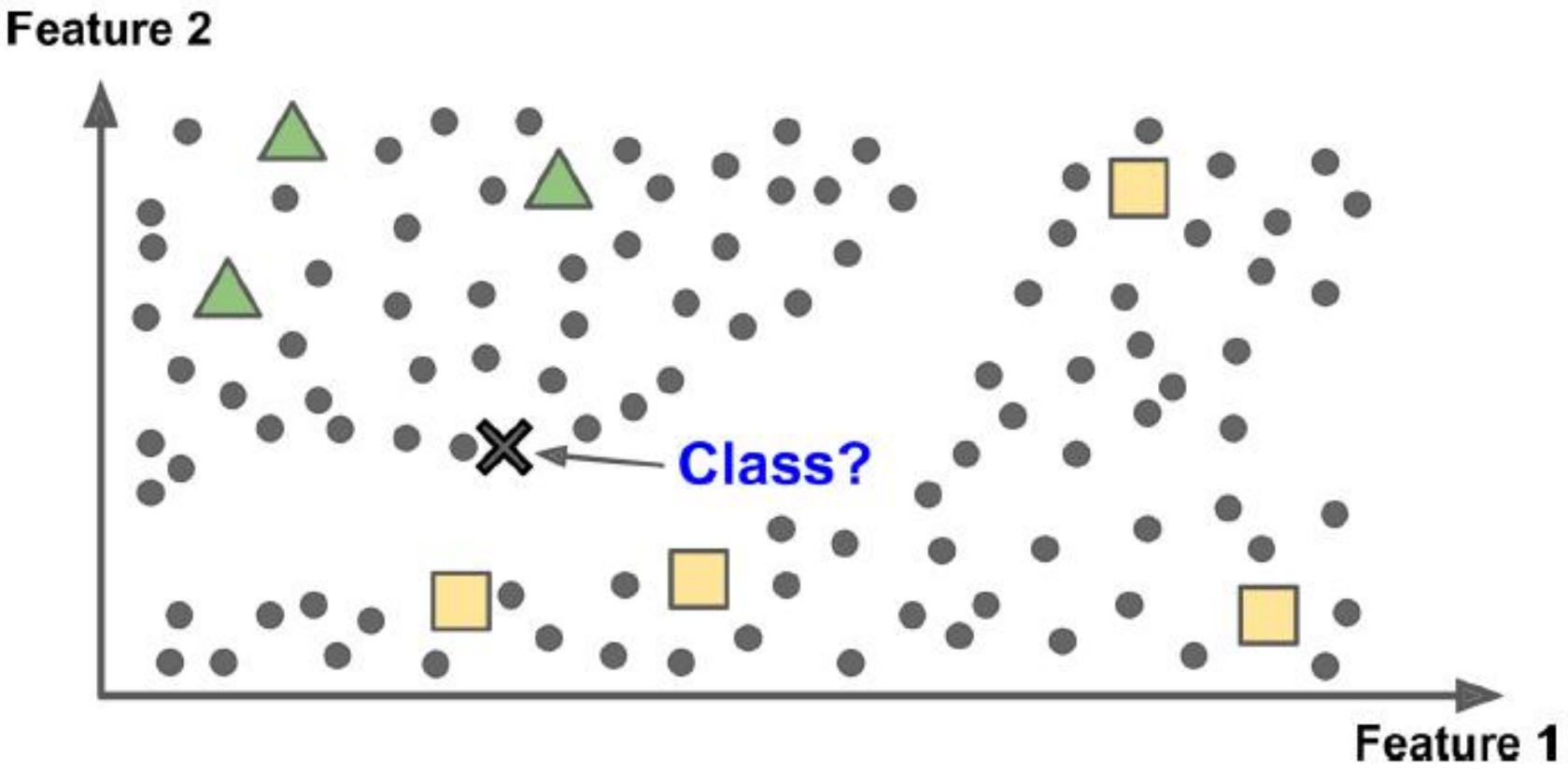


监督/非监督学习

- Visualization



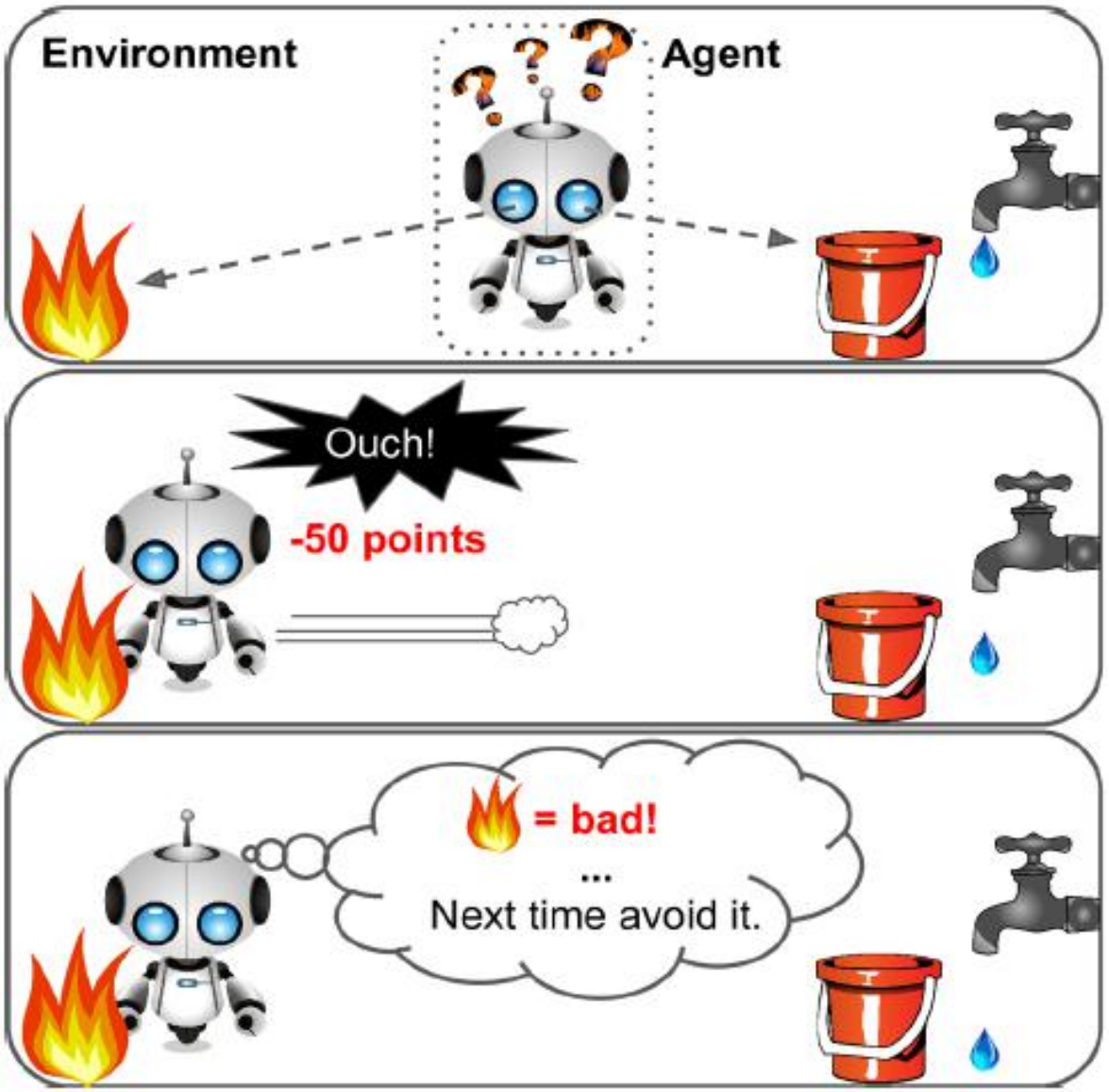
半监督学习



强化学习

- 强化学习与之前的分类非常不同。学习系统在这里被称为智能体（agent），可以对环境进行观察、选择和执行动作，并获得奖励作为回报（负的奖励是惩罚）。然后它必须自己学习哪个是最佳动作序列（称为策略，policy），以得到长久的最大奖励。策略决定了智能体在给定情况下应该采取的行动。

强化学习



1. Observe
2. Select action using policy
3. Action!
4. Get reward or penalty
5. Update policy (learning step)
6. Iterate until an optimal policy is found

n

批量和在线学习

批量学习

- 在批量学习中，系统不能进行增量学习：每次必须用所有可用数据进行训练。这通常会占用大量时间和计算资源，所以一般是线下做的。首先是进行训练，然后再部署到生产环境，再生产环境中它只是使用已经学到的策略。这称为**离线学习**。

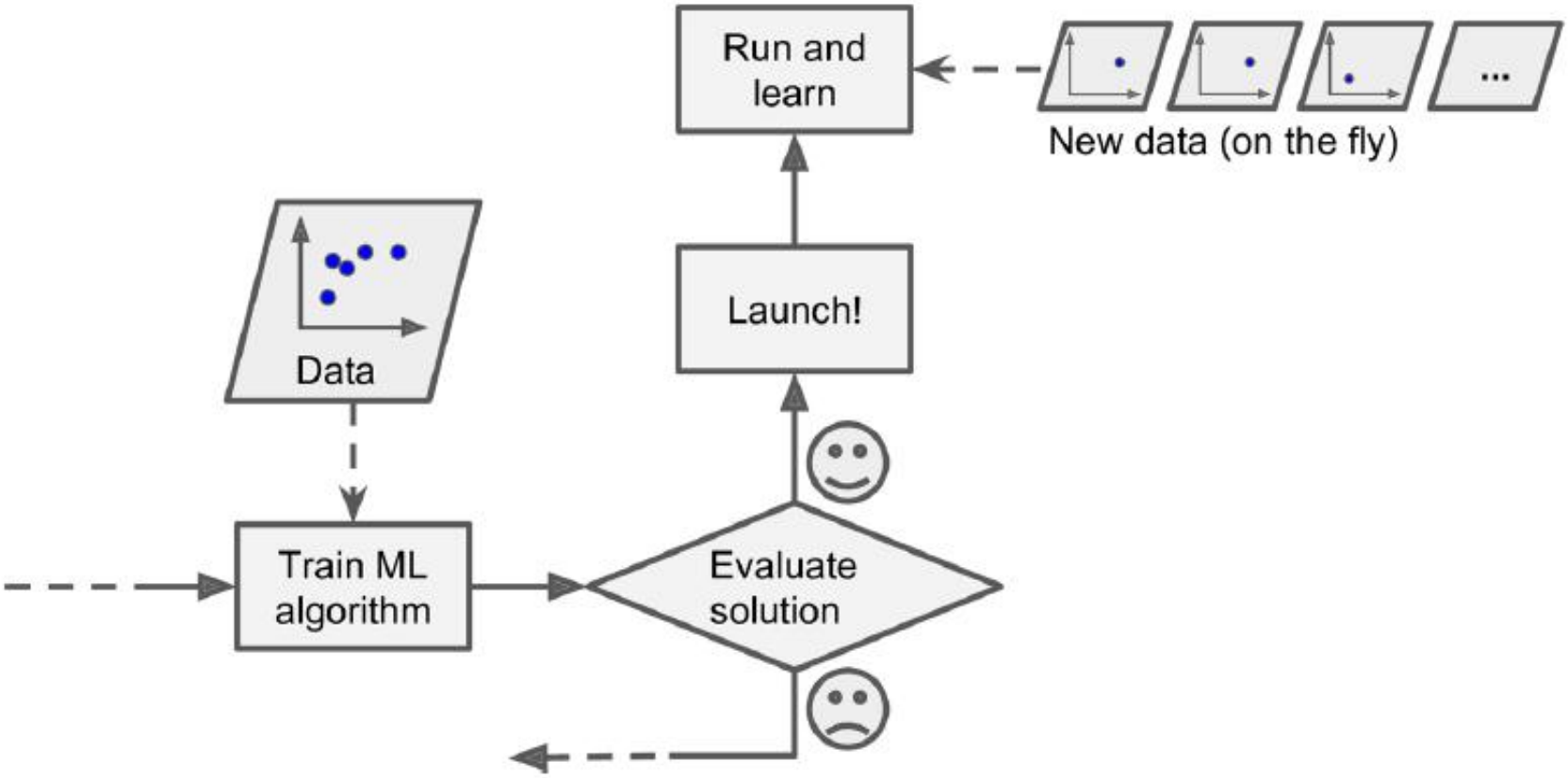
Batch and Online Learning

在线学习

- 在线学习是用数据实例持续地进行训练，可以一次一个或一次几个实例（称为小批量 mini-batch）。每个学习步骤都很快且廉价，所以系统可以动态地学习收到的最新数据。

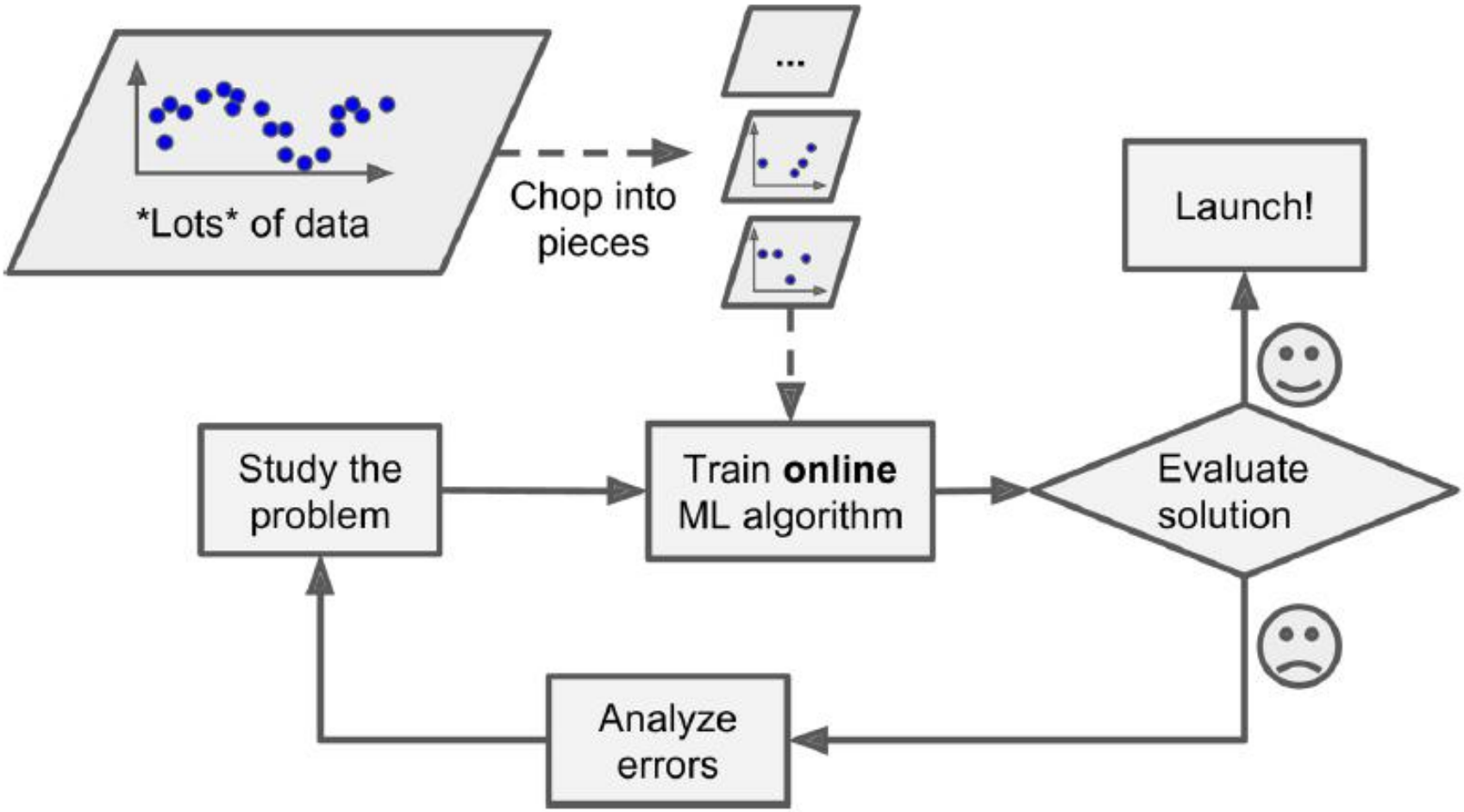
Batch and Online Learning

在线学习



Batch and Online Learning

在线学习（核外学习，out-of-core learning）



基于实例 vs 基于模型学习

基于实例学习

- 也许最简单的学习形式就是用记忆学习。如果用这种方法做一个垃圾邮件检测器，只需标记所有和用户标记的垃圾邮件相同的邮件 — 这个方法可行，但肯定不是最好的。

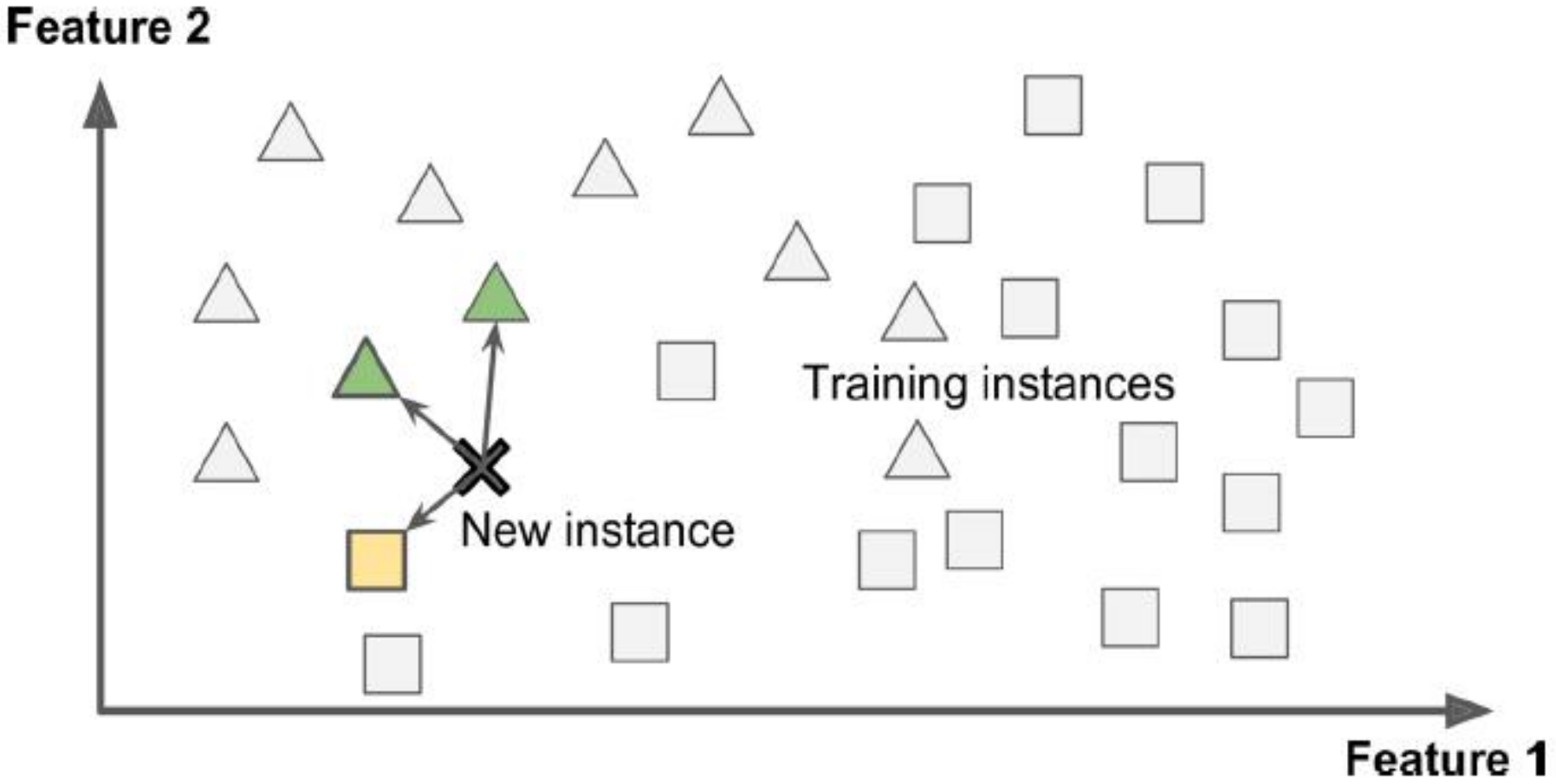
基于实例 vs 基于模型学习

基于实例学习

- 不仅能标记和已知的垃圾邮件相同的邮件，你的垃圾邮件过滤器也要能标记类似垃圾邮件的邮件。这就需要测量两封邮件的相似性。一个（简单的）相似度测量方法是统计两封邮件包含的相同单词的数量。如果一封邮件含有许多垃圾邮件中的词，就会被标记为垃圾邮件。
- 基于实例学习特点：系统先用记忆学习案例，然后使用相似度测量推广到新的例子

基于实例 vs 基于模型学习

- 基于实例学习



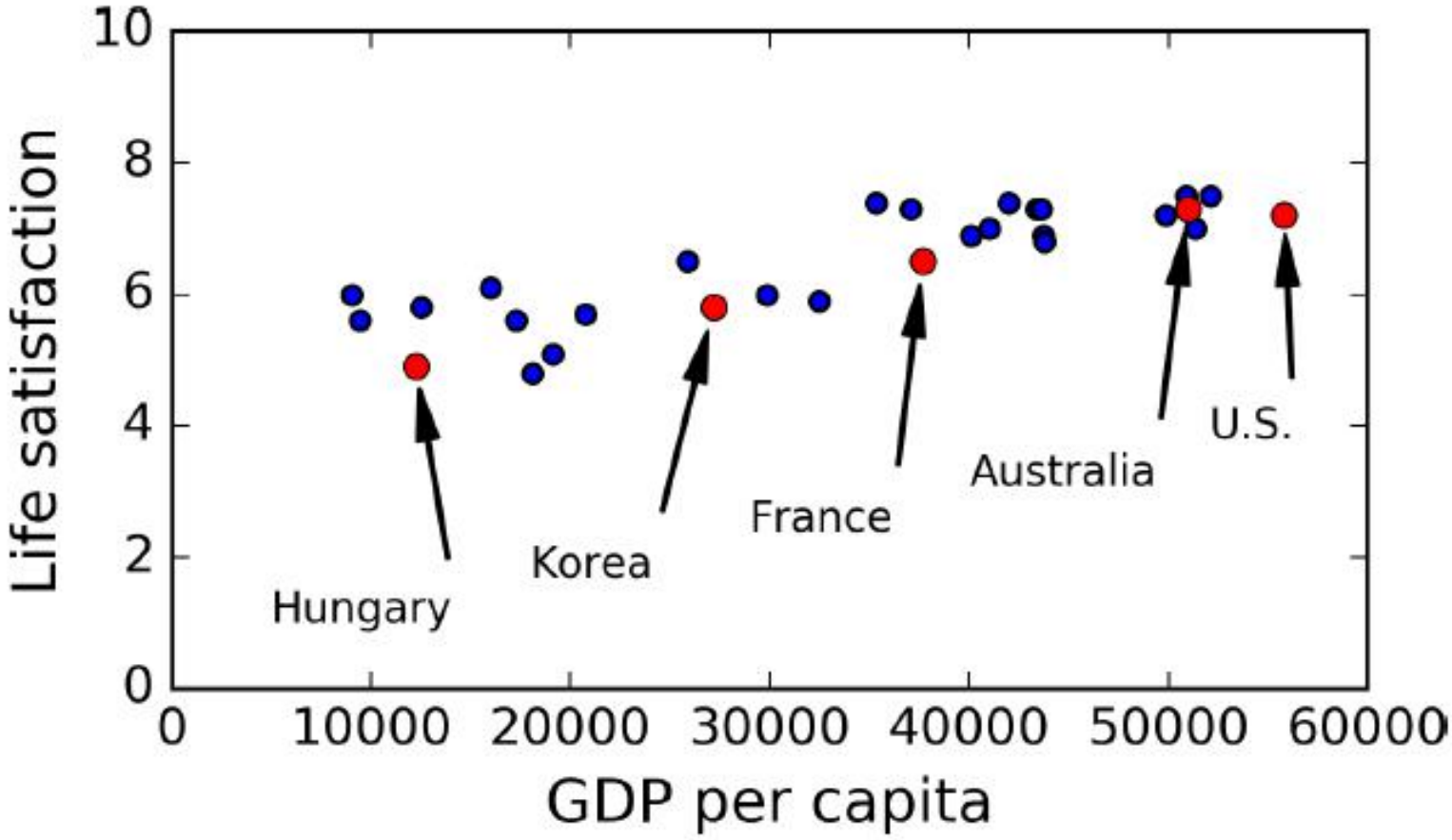
基于实例 vs 基于模型学习

基于模型学习（有钱和快乐的关系）

国家	人均 GDP（美元）	生活满意度
匈牙利	12240	4.9
韩国	27195	5.8
法国	37675	6.5
澳大利亚	50962	7.3
美国	55805	7.2

基于实例 vs 基于模型学习

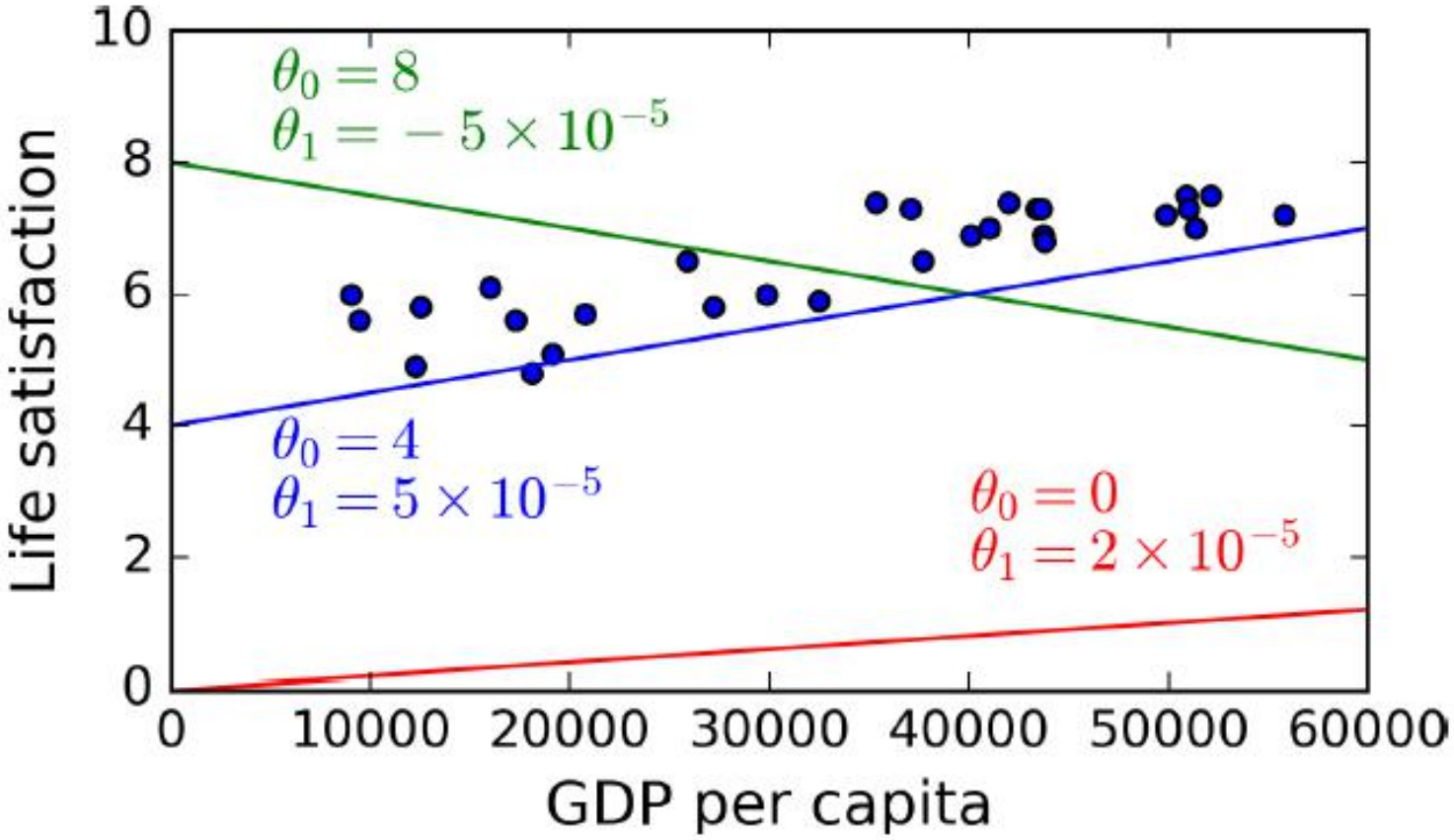
基于模型学习



基于实例 vs 基于模型学习

Equation 1-1. A simple linear model

$$life\_satisfaction = \theta_0 + \theta_1 \times GDP\_per\_capita$$

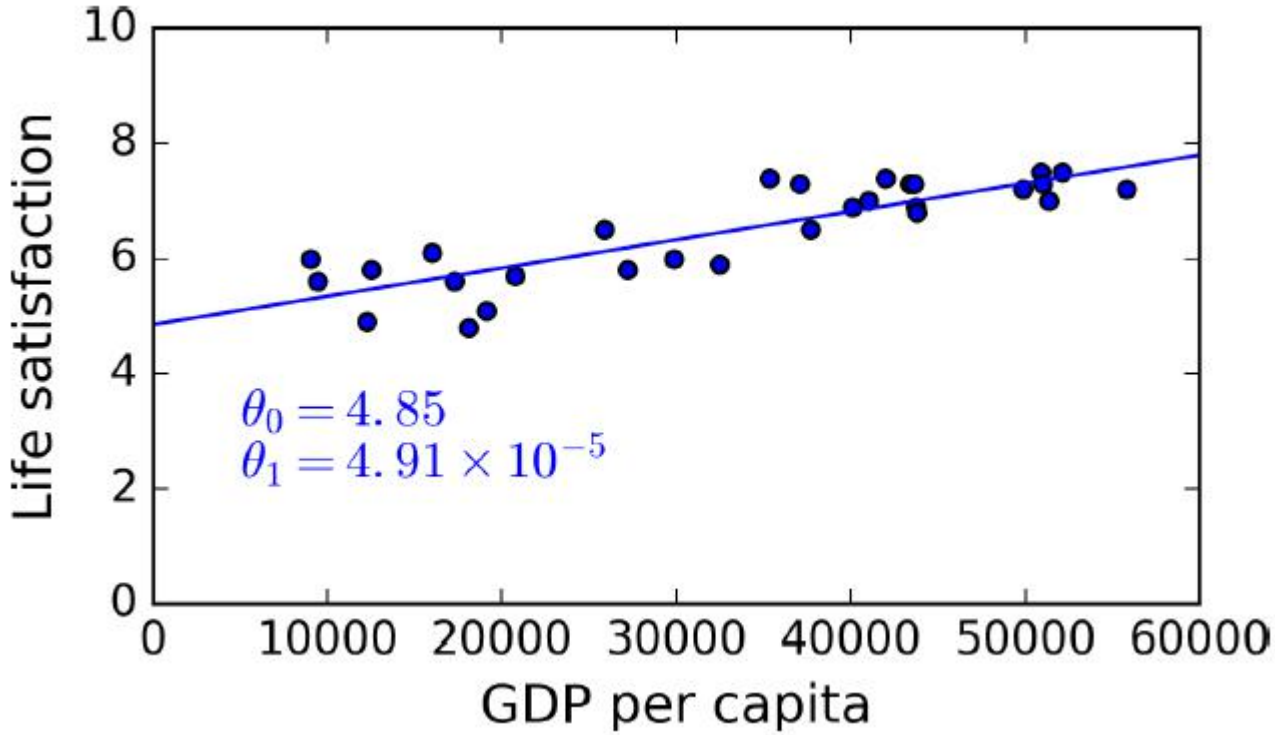


基于实例 vs 基于模型学习

- 在使用模型之前，你需要确定θ_0和θ_1。如何能知道哪个值可以使模型的性能最佳呢？要回答这个问题，你需要指定性能的量度。
- 你可以定义一个实用函数（或**拟合函数**）用来测量模型是否够好，或者你可以定义一个代价函数来测量模型有多差。对于线性回归问题，人们一般是测量线性模型的预测值和训练样本之间的距离差，目标是使距离差最小。

基于实例 vs 基于模型学习

- 这称作模型训练。在我们的例子中，算法得到的参数值是θ_0=4.85和θ_1=4.91×10^{-5}。



```
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import sklearn

oecd_bli = pd.read_csv("oecd_bli_2015.csv", thousands=',')
gdp_per_capita = pd.read_csv("gdp_per_capita.csv",thousands=',',delimiter='\t',
                                                encoding='latin1', na_values="n/a")

country_stats = prepare_country_stats(oecd_bli, gdp_per_capita)
X = np.c_[country_stats["GDP per capita"]]
y = np.c_[country_stats["Life satisfaction"]]


country_stats.plot(kind='scatter', x="GDP per capita", y='Life satisfaction')
plt.show()

lin_reg_model = sklearn.linear_model.LinearRegression()

lin_reg_model.fit(X, y)

X_new = [[22587]] # Cyprus' GDP per capita
print(lin_reg_model.predict(X_new)) # outputs [[ 5.96242338]]
```

典型的机器学习项目

- 研究数据
- 选择模型
- 用训练数据进行训练（即，学习算法搜寻模型参数值，使代价函数最小）
- 最后，使用模型对新案例进行预测（这称作**推断**），希望这个模型泛化效果不差

机器学习的主要挑战

- 会导致机器学习出错的两件事就是“错误的算法”和“错误的数据”。

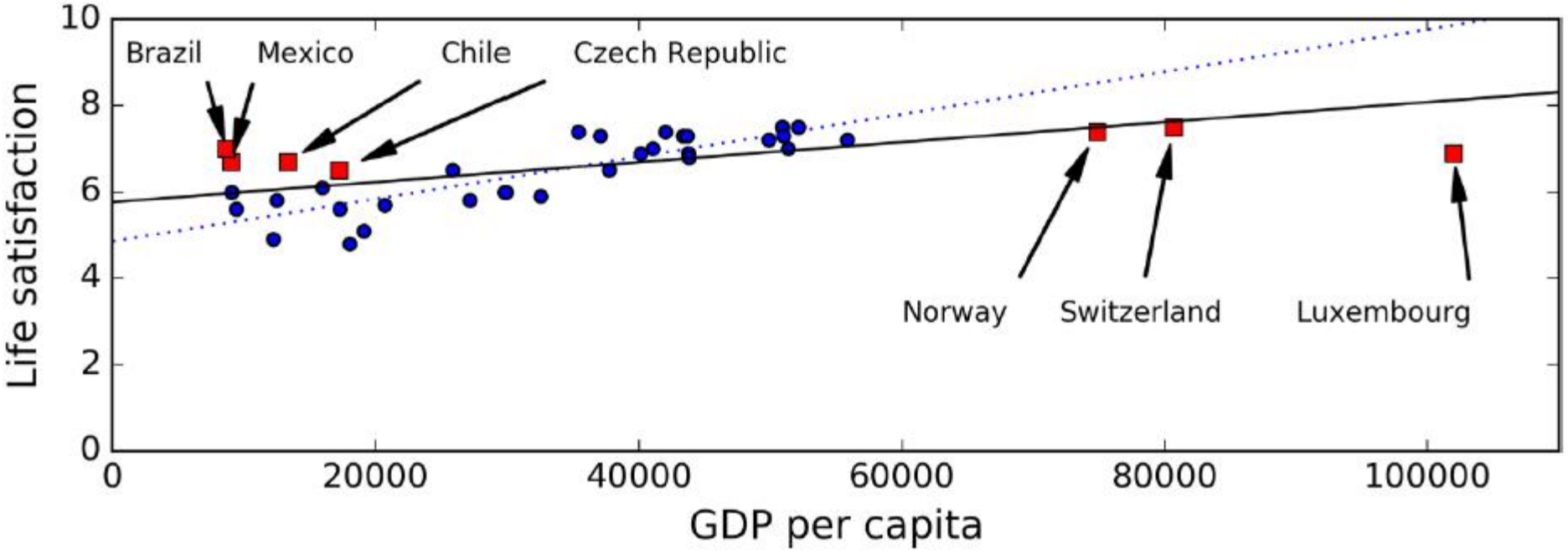
错误的数据

训练数据量不足

- 机器学习往往需要大量数据，才能让多数算法正常工作。即便对于非常简单的问题，一般也需要数千的样本，对于复杂的问题，比如图像或语音识别，你可能需要数百万的样本（除非你有预训练好的模型）。

错误的数据

没有代表性的训练数据



错误的数据

不相关的特征

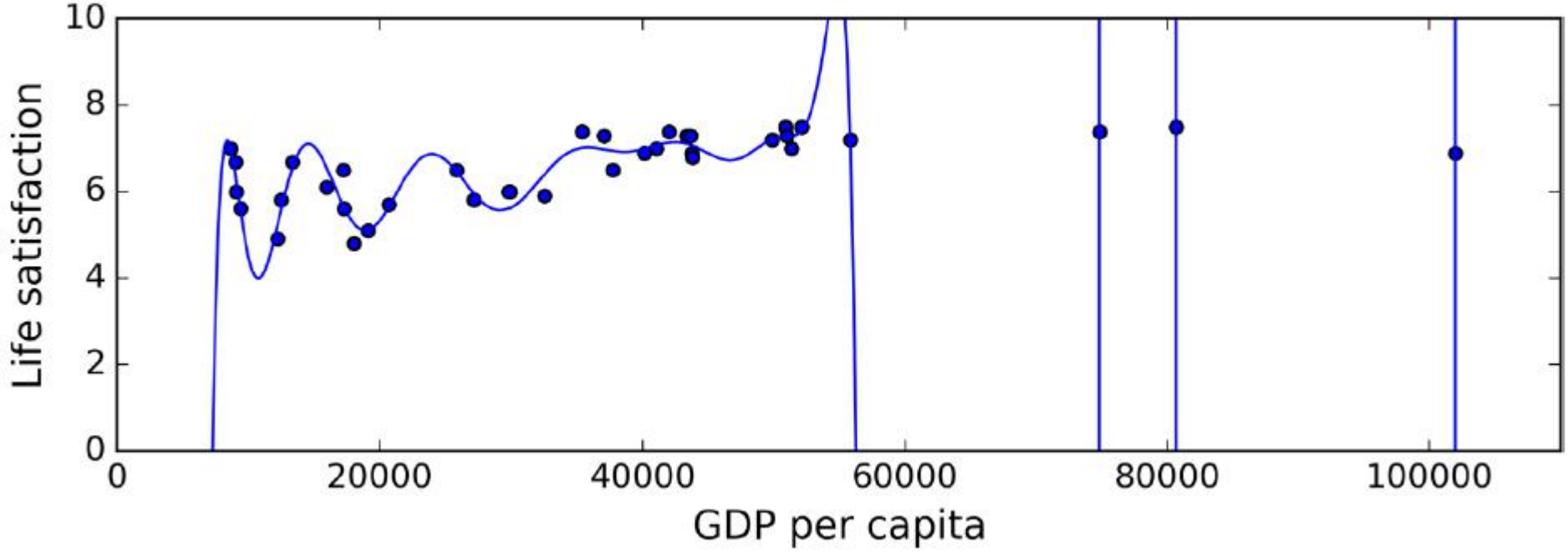
俗语说：进来的是垃圾，出去的也是垃圾。你的系统只有在训练数据包含足够相关特征的情况下，才能进行学习。机器学习项目成功的关键之一是用好的特征进行训练。这个过程称作特征工程，包括：

- 特征选择：在所有存在的特征中选取最有用的特征进行训练。
- 特征提取：组合存在的特征，生成一个更有用的特征（如前面看到的，可以使用降维算法）。
- 收集新数据创建新特征。

错误的算法

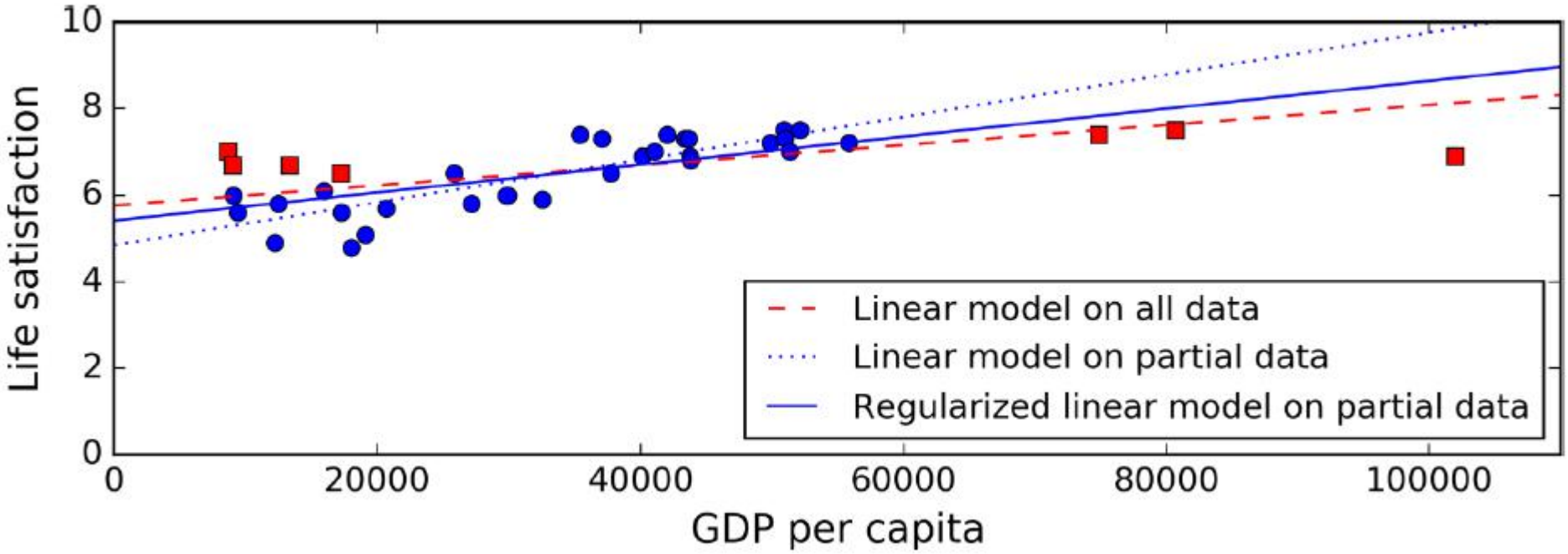
过拟合训练数据

- 模型在训练数据上表现很好，但是在新的数据上效果不好。在机器学习中，这称作过拟合。



错误的算法

- 限定一个模型以让它更简单并且降低过拟合的风险被称作正则化（regularization）



错误的算法

- 正则化的度可以用一个超参数（hyperparameter）控制。超参数是一个学习算法的参数（而不是模型的）。这样，它是不会被学习算法本身影响的，它在训练中是保持不变的。
- 如果你设定的超参数非常大，就会得到一个几乎是平的模型（斜率接近于 0）；这种学习算法几乎肯定不会过拟合训练数据，但是也很难得到一个好的解。调节超参数是创建机器学习算法非常重要的一部分。

错误的算法

欠拟合训练数据

欠拟合是和过拟合相对的：当你的模型过于简单时就会发生。例如，生活满意度的线性模型倾向于欠拟合；现实要比这个模型复杂的多，所以预测很难准确，即使在训练样本上也很难准确。

解决这个问题的选项包括：

- 选择一个更强大的模型，带有更多参数
- 用更好的特征训练学习算法（特征工程）
- 减小对模型的限制（比如，减小正则化超参数）

测试和验证

- 要知道一个模型推广到新样本的效果，唯一的办法就是真正的进行试验。一种方法是将模型部署到生产环境，观察它的性能。这么做可以，但是如果模型的性能很差，就会引起用户抱怨。
- 更好的选项是将你的数据分成两个集合：训练集和测试集。正如它们的名字，用训练集进行训练，用测试集进行测试。对新样本的错误率称作泛化误差，通过模型对测试集的评估，你可以预估这个误差。

测试和验证

- 如果我们想做一些正则化以避免过拟合。问题是：如何选择正则化超参数的值？
- 通常的解决方案是，再保留一个集合，称作验证集合。用训练集和多个超参数训练多个模型，选择在验证集上有最佳性能的模型和超参数。当你对模型满意时，用测试集再做最后一次测试，以得到泛化误差率的预估。

测试和验证

- 为了避免“浪费”过多训练数据在验证集上，通常的办法是使用交叉验证：训练集分成互补的子集，每个模型用不同的子集训练，再用剩下的子集验证。
- 一旦确定模型类型和超参数，最终的模型使用这些超参数和全部的训练集进行训练，用测试集得到泛化误差率。

一个完整的机器学习项目

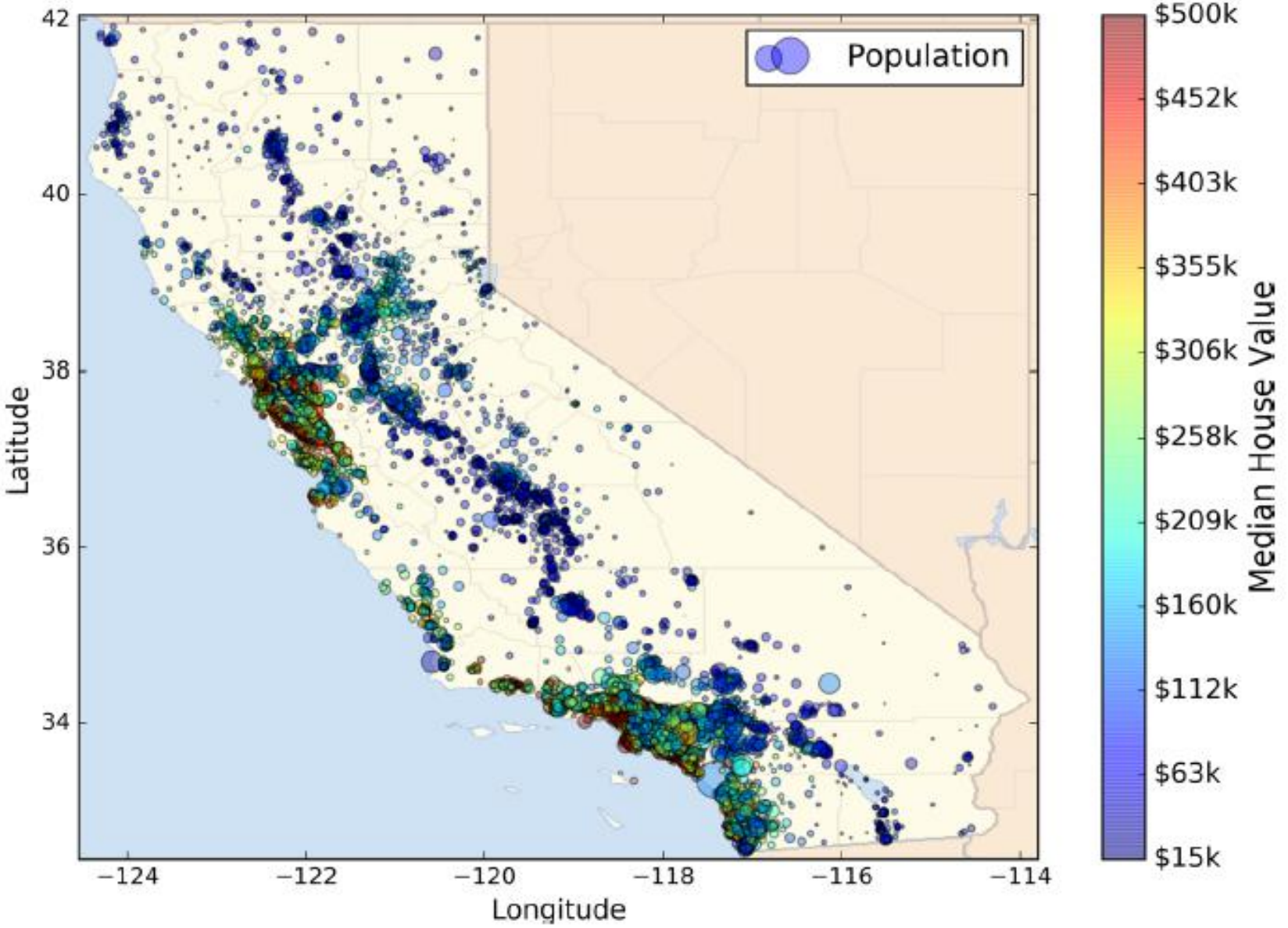
下面完整地学习一个机器学习案例项目。主要步骤包括：

- 项目描述。
- 获取数据。
- 发现并可视化数据，发现规律。
- 为机器学习算法准备数据。
- 选择模型，进行训练。
- 微调模型。
- 给出解决方案。
- 部署、监控、维护系统。

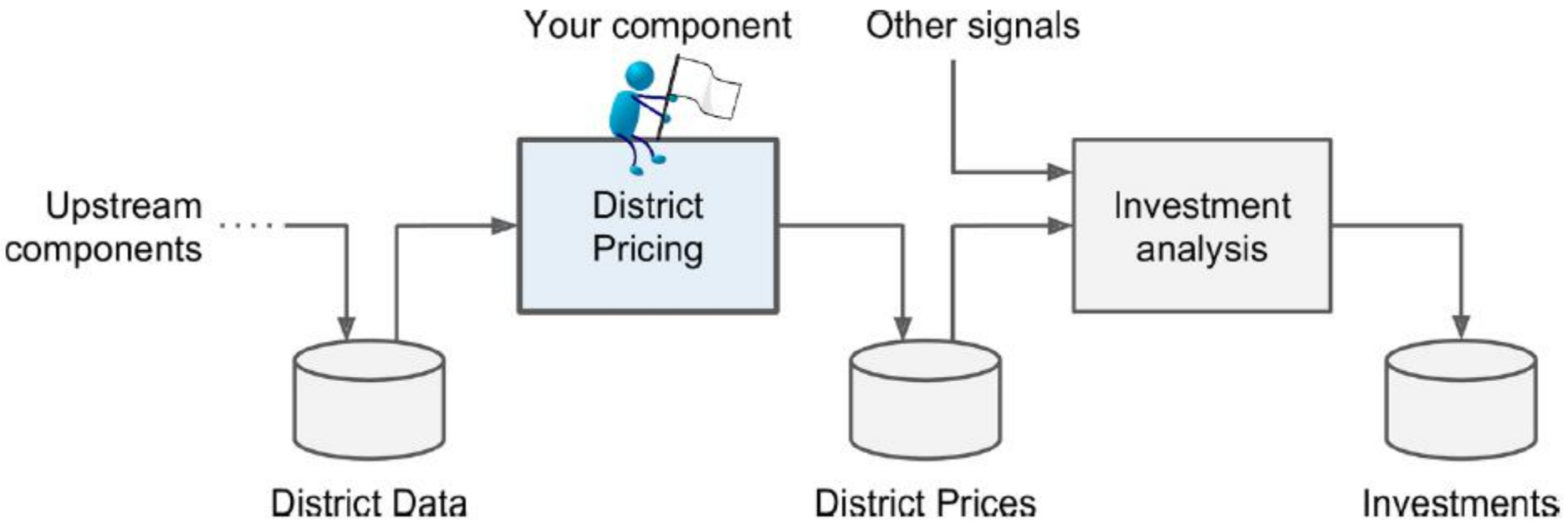
数据集

- 我们选择的是 StatLib 的加州房产价格数据集。这个数据集是基于 1990 年加州普查的数据。数据已经有点老，但是它有许多优点，利于学习，所以假设这个数据为最近的。为了便于教学，我们添加了一个类别属性，并除去了一些旧的属性。

数据集


Population
Latitude
Longitude
42
40
38
36
34
−124
−122
−120
−118
−116
−114
Median House Value
$500k
$452k
$403k
$355k
$306k
$258k
$209k
$160k
$112k
$63k
$15k

划定问题


Your component
Other signals
Upstream components
District Pricing
Investment analysis
District Data
District Prices
Investments

划定问题

- 首先，你需要划定问题：监督或非监督，还是强化学习？这是个分类任务、回归任务，还是其它的？要使用批量学习还是线上学习？
- 很明显，这是一个典型的监督学习任务，因为你要使用的是有标签的训练样本（每个实例都有预定的产出，即街区的房价中位数）。并且，这是一个典型的回归任务，因为你要预测一个值。讲的更细些，这是一个多变量回归问题，因为系统要使用多个变量进行预测（要使用街区的人口，收入中位数等等）。最后，没有连续的数据流进入系统，没有特别需求需要对数据变动作出快速适应。数据量不大可以放到内存中，因此批量学习就够了。

选择性能指标

- 下一步是选择性能指标。回归问题的典型指标是均方根误差（RMSE）。均方根误差测量的是系统预测误差的标准差。

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

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

选择性能指标

- 在有些情况下，你可能需要另外的函数。例如，假设存在许多异常的街区。此时，你可能需要使用平均绝对误差（Mean Absolute Error，也称作平均绝对偏差）

Equation 2-2. Mean Absolute Error

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

选择性能指标

- RMSE 和 MAE 都是测量预测值和目标值两个向量距离的方法。有多种测量距离或范数的方法：
- 计算对应欧几里得范数的平方和的根（RMSE）：这个距离介绍过。它也称作ℓ2范数，标记为 $||.||_2$（或只是$||.||$）。
- 计算对应于ℓ1（标记为 $||.||_1$）范数的绝对值和（MAE）。有时，也称其为曼哈顿范数，因为它测量了城市中的两点，沿着矩形的边行走的距离。
- 更一般的，包含n个元素的向量v的ℓk范数（K 阶闵氏范数），定义成

$$\| \mathbf{v} \|_k = \left(|v_0|^k + |v_1|^k + \cdots + |v_n|^k \right)^{\frac{1}{k}}$$

- ℓ0（汉明范数）只显示了这个向量的基数（即，非零元素的个数），ℓ∞（切比雪夫范数）是向量中最大的绝对值。

获取数据

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

DOWNLOAD_ROOT = "https://raw.githubusercontent.com/ageron/handson-ml/master/"
HOUSING_PATH = "datasets/housing"
HOUSING_URL = DOWNLOAD_ROOT + HOUSING_PATH + "/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()
```

加载数据

```
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)
```

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

In [6]:
```
housing.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 10 columns):
longitude             20640 non-null float64
latitude              20640 non-null float64
housing_median_age    20640 non-null float64
total_rooms           20640 non-null float64
total_bedrooms        20433 non-null float64
population            20640 non-null float64
households            20640 non-null float64
median_income         20640 non-null float64
median_house_value    20640 non-null float64
ocean_proximity       20640 non-null object
dtypes: float64(9), object(1)
memory usage: 1.6+ MB
```

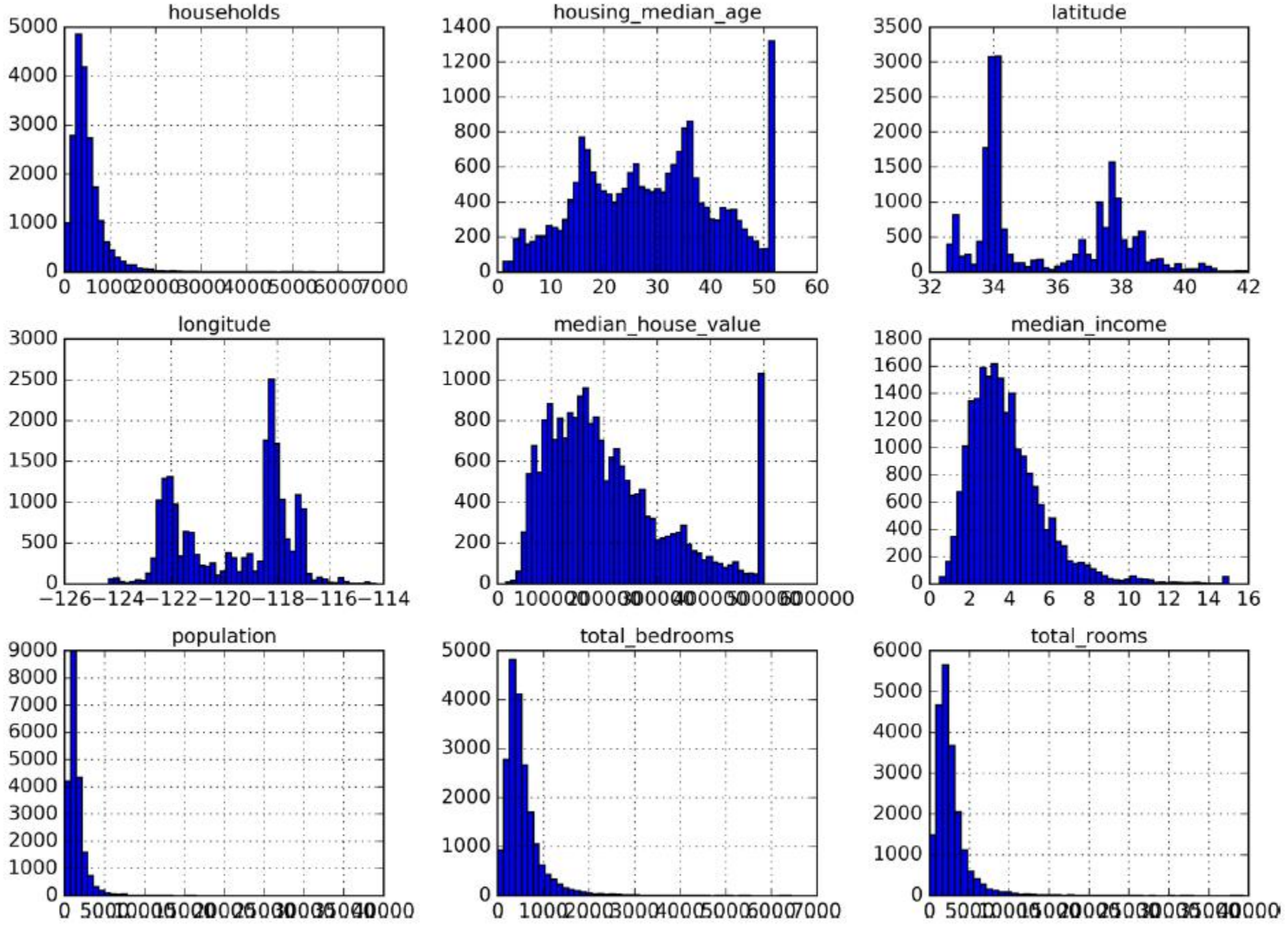
In [8]:
```
housing.describe()
```

Out[8]:

	longitude	latitude	housing_median_age	total_rooms	total_bedr
count	20640.000000	20640.000000	20640.000000	20640.000000	20433.000
mean	-119.569704	35.631861	28.639486	2635.763081	537.87055
std	2.003532	2.135952	12.585558	2181.615252	421.38507
min	-124.350000	32.540000	1.000000	2.000000	1.000000
25%	-121.800000	33.930000	18.000000	1447.750000	296.00000
50%	-118.490000	34.260000	29.000000	2127.000000	435.00000
75%	-118.010000	37.710000	37.000000	3148.000000	647.00000
max	-114.310000	41.950000	52.000000	39320.000000	6445.0000

展示数据

```
%matplotlib inline # only in a Jupyter notebook
import matplotlib.pyplot as plt
housing.hist(bins=50, figsize=(20,15))
plt.show()
```


households
housing_median_age
latitude
longitude
median_house_value
median_income
population
total_bedrooms
total_rooms

创建测试集

```
import numpy as np
def split_train_test(data, test_ratio):
    shuffled_indices = np.random.permutation(len(data))
    test_set_size = int(len(data) * test_ratio)
    test_indices = shuffled_indices[:test_set_size]
    train_indices = shuffled_indices[test_set_size:]
    return data.iloc[train_indices], data.iloc[test_indices]
```

创建测试集

```
import hashlib
def test_set_check(identifier, test_ratio, hash):
    return hash(np.int64(identifier)).digest()[-1] < 256 * test_ratio
def split_train_test_by_id(data, test_ratio, id_column,
                                                hash=hashlib.md5):
    ids = data[id_column]
    in_test_set = ids.apply(lambda id_: test_set_check(id_, test_ratio, hash))
    return data.loc[~in_test_set], data.loc[in_test_set]
```

创建测试集

- 目前为止，我们采用的都是纯随机的取样方法。当你的数据集很大时（尤其是和属性数相比），这通常可行；但如果数据集不大，就会有采样偏差的风险。当一个调查公司想要对 1000 个人进行调查，它们不是在电话亭里随机选 1000 个人出来。调查公司要保证这 1000 个人对人群整体有代表性。

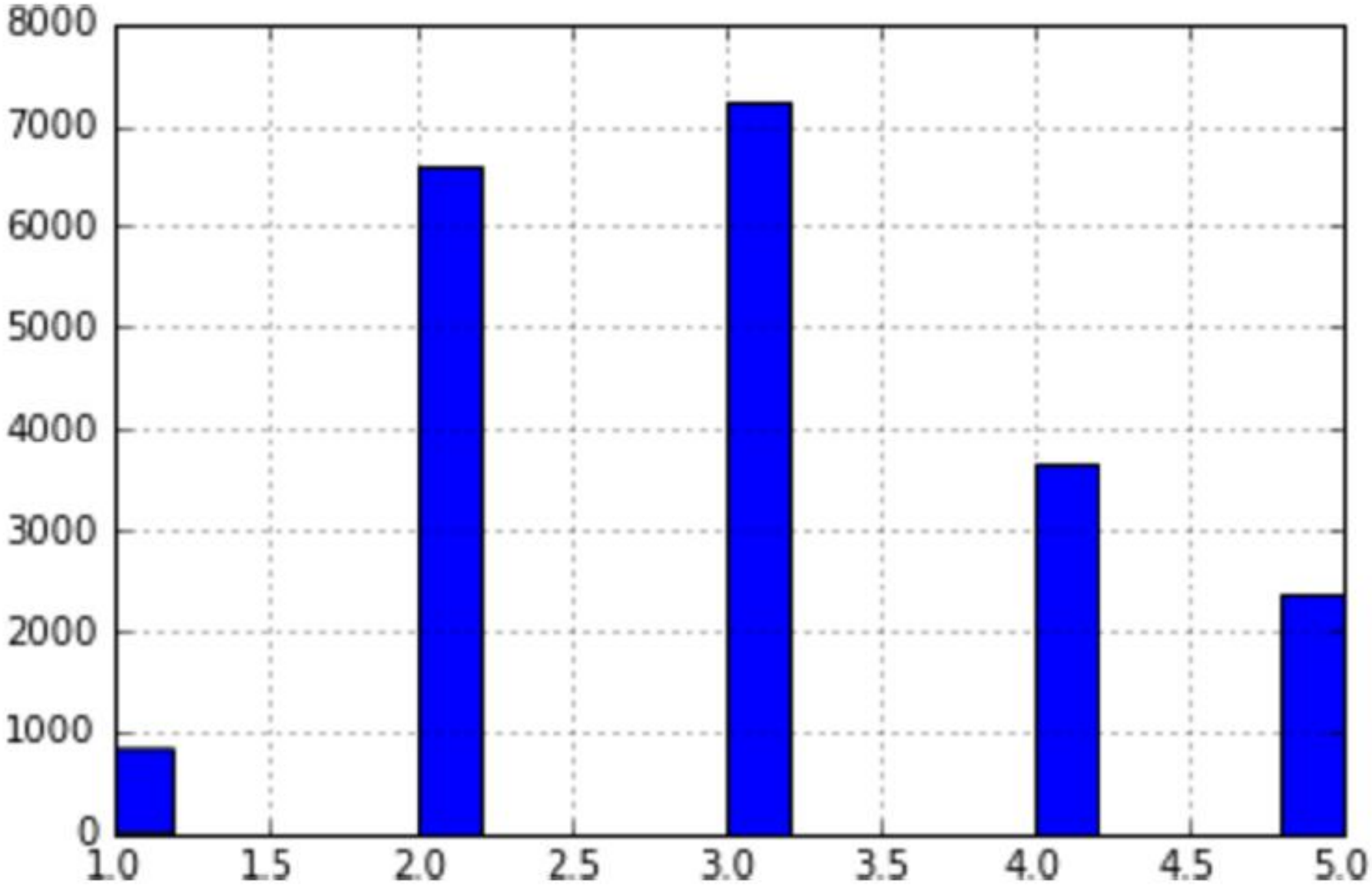
创建测试集

- 例如，美国人口的 51.3% 是女性，48.7% 是男性。所以在美国，严谨的调查需要保证样本也是这个比例：513 名女性，487 名男性。这称作分层采样（stratified sampling）：将人群分成均匀的子分组，称为分层，从每个分层去取合适数量的实例，以保证测试集对总人数有代表性。如果调查公司采用纯随机采样，会有 12% 的概率导致采样偏差：女性人数少于 49%，或多于 54%。不管发生那种情况，调查结果都会严重偏差。

创建测试集

- 假设专家告诉你，收入中位数是预测房价中位数非常重要的属性。你可能想要保证测试集可以代表整体数据集中的多种收入分类。因为收入中位数是一个连续的数值属性，你首先需要创建一个收入类别属性。看一下收入中位数的柱状图：

创建测试集


8000
7000
6000
5000
4000
3000
2000
1000
0
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0

创建测试集

- 以下代码通过将收入中位数除以 1.5（以限制收入分类的数量），创建了一个收入类别属性，用ceil对值舍入（以产生离散的分类），然后将所有大于 5的分类归入到分类 5：

```
housing["income_cat"] = np.ceil(housing["median_income"] / 1.5)
housing["income_cat"].where(housing["income_cat"] > 5, 5.0, inplace=True)
```

创建测试集

- 现在，就可以根据收入分类，进行分层采样。你可以使用 Scikit-Learn 的StratifiedShuffleSplit类：

```
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]
```

创建测试集

```
>>> housing["income_cat"].value_counts() / len(housing)
3.0 0.350581
2.0 0.318847
4.0 0.176308
5.0 0.114438
1.0 0.039826
Name: income_cat, dtype: float64
```

创建测试集

	Overall	Random	Stratified	Rand. %error	Strat. %error
1.0	0.039826	0.040213	0.039738	0.973236	-0.219137
2.0	0.318847	0.324370	0.318876	1.732260	0.009032
3.0	0.350581	0.358527	0.350618	2.266446	0.010408
4.0	0.176308	0.167393	0.176399	-5.056334	0.051717
5.0	0.114438	0.109496	0.114369	-4.318374	-0.060464

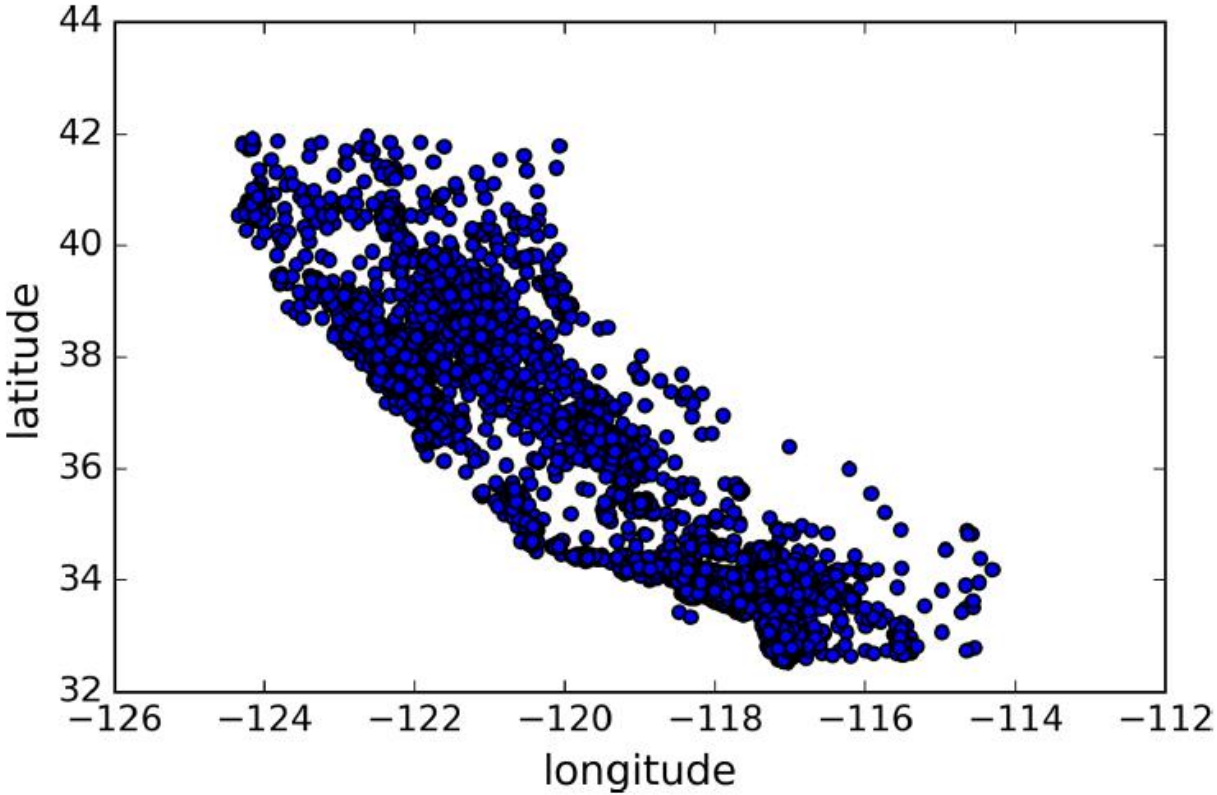
数据探索和可视化

- 目前为止，你只是快速查看了数据，对要处理的数据有了整体了解。现在的目标是更深的探索数据。
- 首先，保证你将测试集放在了一旁，只是研究训练集。另外，如果训练集非常大，你可能需要再采样一个探索集，保证操作方便快速。在我们的案例中，数据集很小，所以可以在全集上直接工作。创建一个副本，以免损伤训练集：

```
housing = strat_train_set.copy()
```

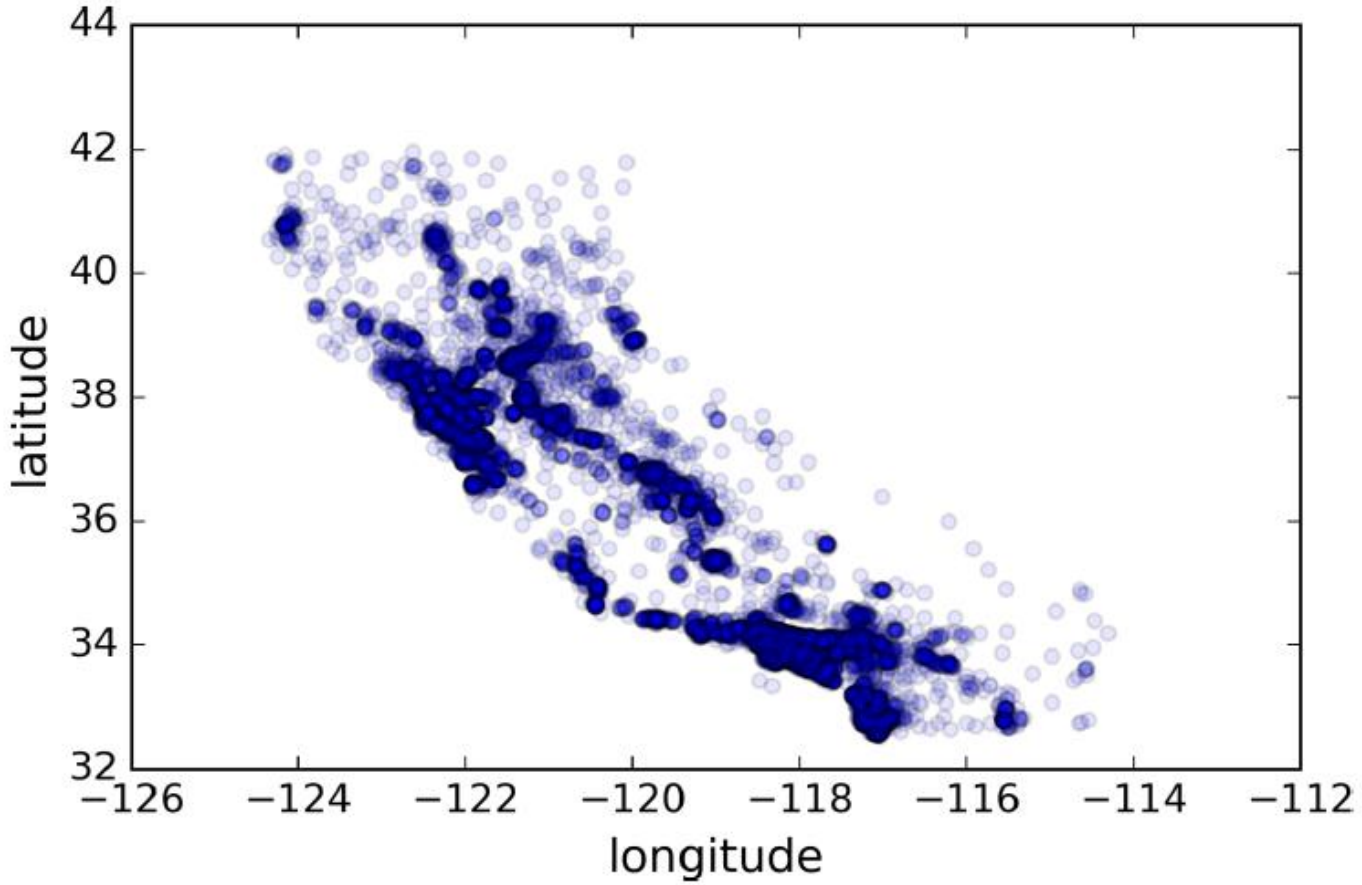
地理数据可视化

- 因为存在地理信息（纬度和经度），创建一个所有街区的散点图来数据可视化是一个不错的主意：
- housing.plot(kind="scatter", x="longitude", y="latitude")



地理数据可视化

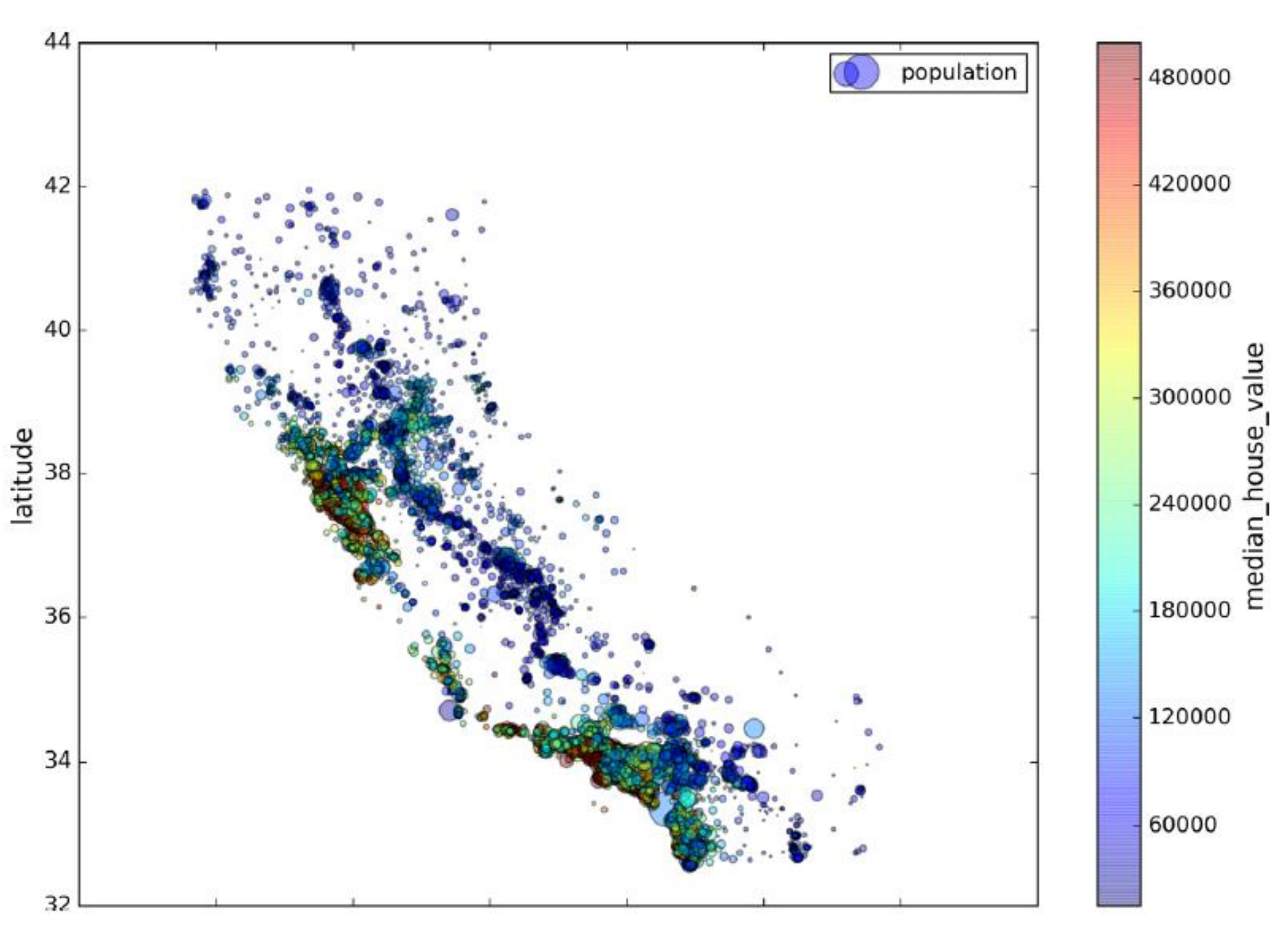
- 将alpha设为 0.1，可以更容易看出数据点的密度：
- housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.1)



地理数据可视化

- 现在来看房价。每个圈的半径表示街区的人口（选项s），颜色代表价格（选项c）。我们用预先定义的名为jet的颜色图（选项cmap），它的范围是从蓝色（低价）到红色（高价）：

```
housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.4,
    s=housing["population"]/100, label="population",
    c="median_house_value", cmap=plt.get_cmap("jet"), colorbar=True,
)
plt.legend()
```


population
latitude
44
42
40
38
36
34
32
480000
420000
360000
300000
240000
180000
120000
60000
median_house_value

查找关联

- 因为数据集并不是非常大，你可以很容易地使用corr()方法计算出每对属性间的标准相关系数（standard correlation coefficient，也称作皮尔逊相关系数）：

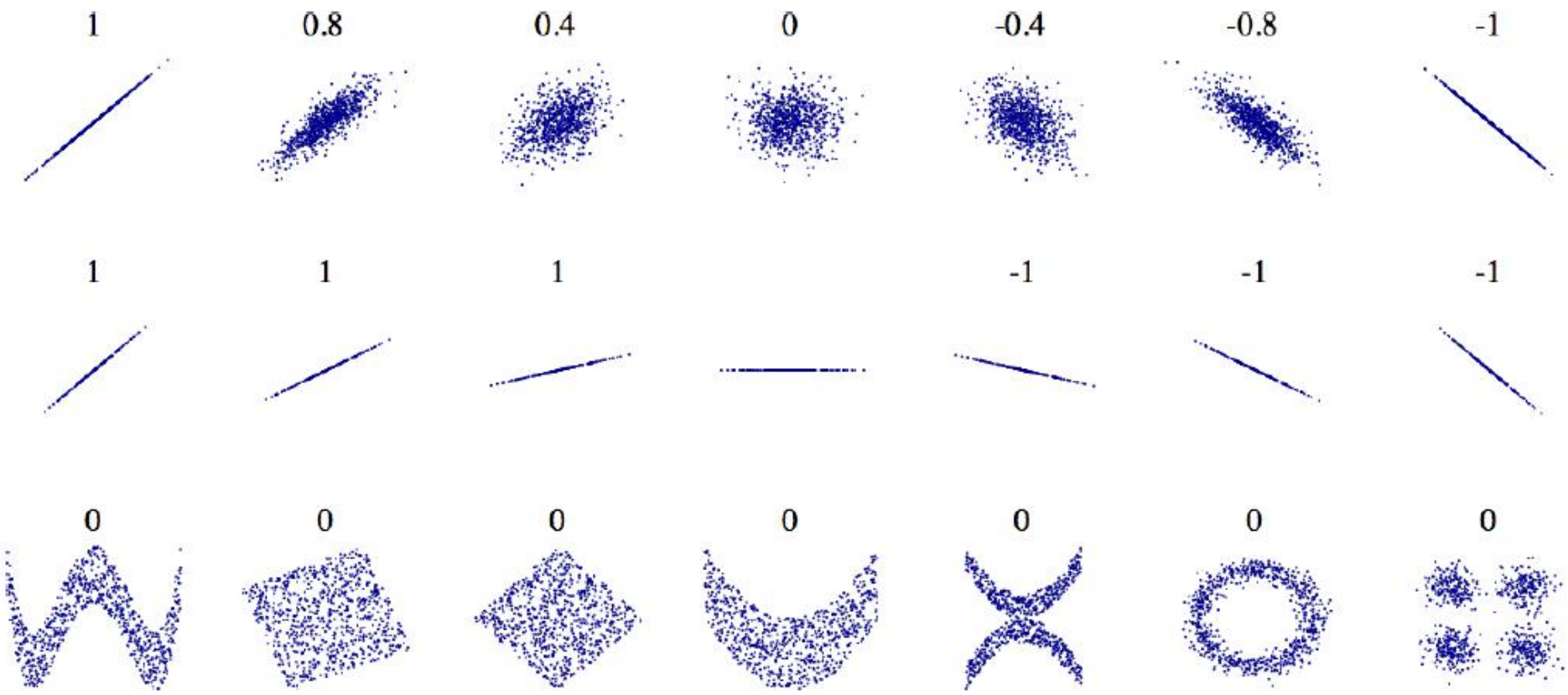
```
corr_matrix = housing.corr()
```

查找关联

```
>>> 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
```

查找关联

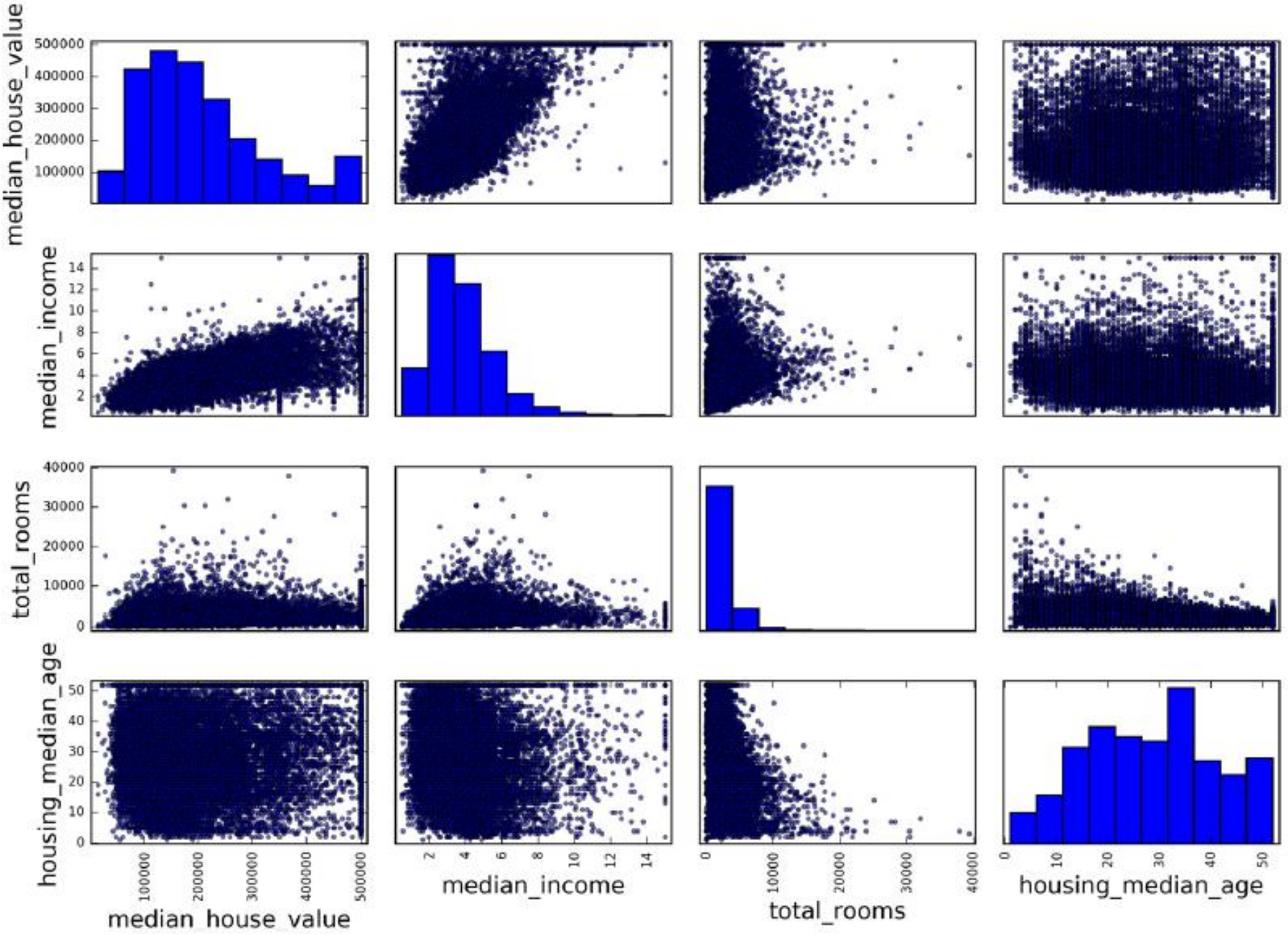
Standard correlation coefficient of various datasets



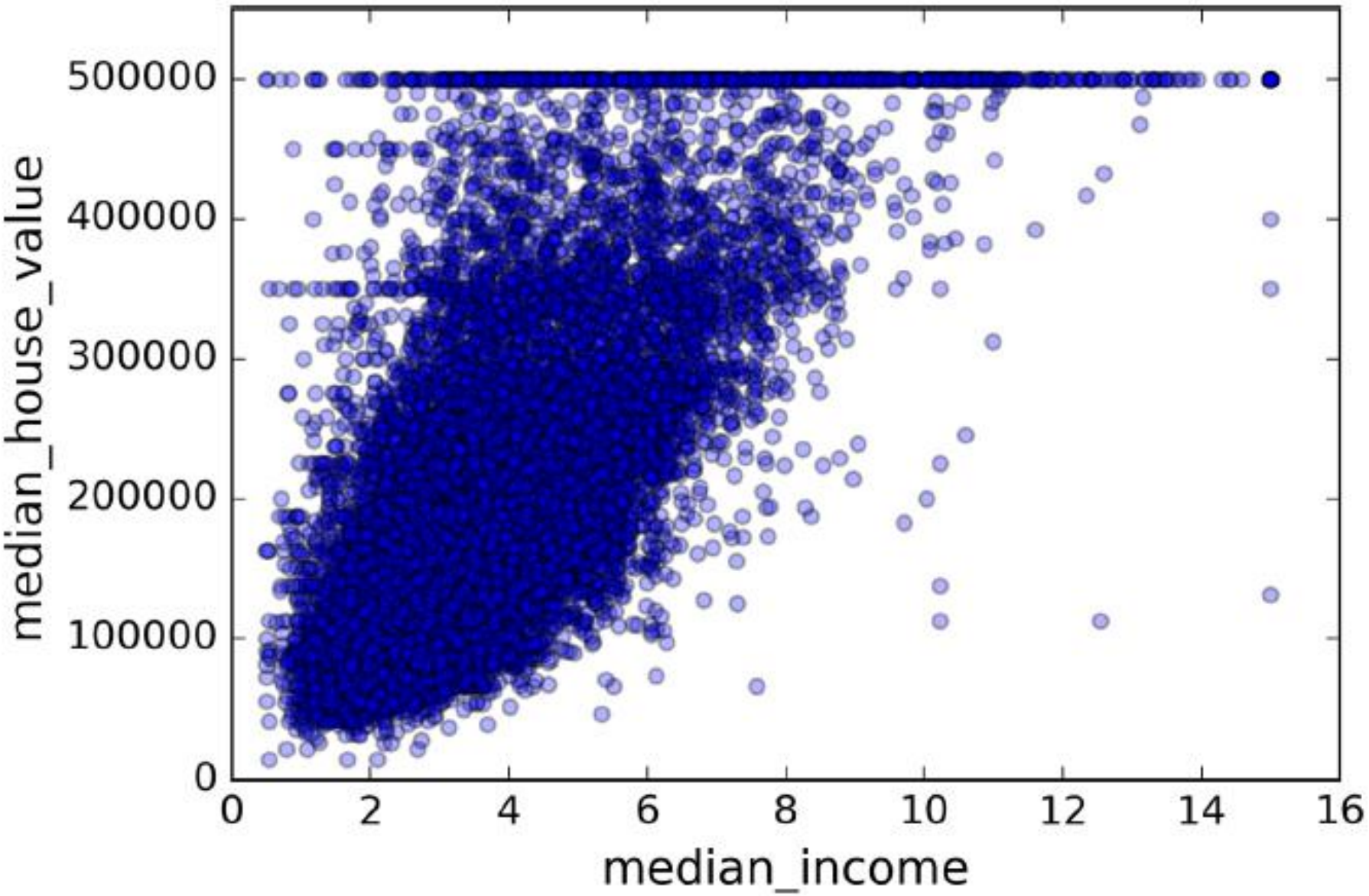
查找关联

- 另一种检测属性间相关系数的方法是使用 Pandas 的scatter_matrix函数，它能画出每个数值属性对每个其它数值属性的图。

```
from pandas.tools.plotting import scatter_matrix
attributes = ["median_house_value", "median_income", "total_rooms",
              "housing_median_age"]
scatter_matrix(housing[attributes], figsize=(12, 8))
```


median_house_value
median_income
total_rooms
housing_median_age
median_house_value
median_income
total_rooms
housing_median_age

```
housing.plot(kind="scatter", x="median_income", y="median_house_value",
             alpha=0.1)
```



属性组合试验

- 给算法准备数据之前，你需要做的最后一件事是尝试多种属性组合。例如，如果你不知道某个街区有多少户，该街区的总房间数就没什么用。你真正需要的是每户有几个房间。相似的，总卧室数也不重要：你可能需要将其与房间数进行比较。每户的人口数也是一个有趣的属性组合。让我们来创建这些新的属性：

```
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"]

>>> corr_matrix = housing.corr()
>>> corr_matrix["median_house_value"].sort_values(ascending=False)
median_house_value 1.000000
median_income 0.687170
rooms_per_household 0.199343
total_rooms 0.135231
housing_median_age 0.114220
households 0.064702
total_bedrooms 0.047865
population_per_household -0.021984
population -0.026699
longitude -0.047279
latitude -0.142826
bedrooms_per_room -0.260070
Name: median_house_value, dtype: float64
```

数据清洗

```
housing = strat_train_set.drop("median_house_value", axis=1)
housing_labels = strat_train_set["median_house_value"].copy()

housing.dropna(subset=["total_bedrooms"]) # option 1
housing.drop("total_bedrooms", axis=1) # option 2
median = housing["total_bedrooms"].median()
housing["total_bedrooms"].fillna(median) # option 3

from sklearn.preprocessing import Imputer
imputer = Imputer(strategy="median")
housing_num = housing.drop("ocean_proximity", axis=1)
imputer.fit(housing_num)
```

为机器学习算法准备数据

```
>>> imputer.statistics_
array([ -118.51 , 34.26 , 29. , 2119. , 433. , 1164. , 408. , 3.5414])
>>> housing_num.median().values
array([ -118.51 , 34.26 , 29. , 2119. , 433. , 1164. , 408. , 3.5414])

X = imputer.transform(housing_num)

housing_tr = pd.DataFrame(X, columns=housing_num.columns)
```

处理文本和类别属性

- 前面，我们丢弃了类别属性ocean_proximity，因为它是一个文本属性，不能计算出中位数。大多数机器学习算法更喜欢和数字打交道，所以让我们把这些文本标签转换为数字。

```
>>> from sklearn.preprocessing import LabelEncoder
>>> encoder = LabelEncoder()
>>> housing_cat = housing["ocean_proximity"]
>>> housing_cat_encoded = encoder.fit_transform(housing_cat)
>>> housing_cat_encoded
array([1, 1, 4, ..., 1, 0, 3])
```

处理文本和类别属性

```
>>> print(encoder.classes_)
['<1H OCEAN' 'INLAND' 'ISLAND' 'NEAR BAY' 'NEAR OCEAN']
```

- 这种做法的问题是，ML 算法会认为两个临近的值比两个疏远的值要更相似。显然这样不对（比如，分类 0 和分类 4 就比分类 0 和分类 1 更相似）。要解决这个问题，一个常见的方法是给每个分类创建一个二元属性：当分类是<1H OCEAN，该属性为 1（否则为 0），当分类是INLAND，另一个属性等于 1（否则为 0），以此类推。这称作独热编码（One-Hot Encoding），因为只有一个属性会等于 1（热），其余会是 0（冷）。

处理文本和类别属性

- Scikit-Learn 提供了一个编码器OneHotEncoder，用于将整数分类值转变为独热向量。注意fit_transform()用于 2D 数组，而housing_cat_encoded是一个 1D 数组，所以需要将其变形：

```
>>> from sklearn.preprocessing import OneHotEncoder
>>> encoder = OneHotEncoder()
>>> housing_cat_1hot = encoder.fit_transform(housing_cat_encoded.reshape(-1,1))
>>> housing_cat_1hot
<16513x5 sparse matrix of type '<class 'numpy.float64'>'
        with 16513 stored elements in Compressed Sparse Row format>
```

处理文本和类别属性

- 经过独热编码，我们得到了一个有数千列的矩阵，这个矩阵每行只有一个 1，其余都是 0。使用大量内存来存储这些 0 非常浪费，所以稀疏矩阵只存储非零元素的位置。你可以像一个 2D 数据那样进行使用，但是如果你真的想将其转变成一个（密集的）NumPy 数组，只需调用toarray()方法：

- **>>>** housing_cat_1hot.toarray()
- array([[0., 1., 0., 0., 0.],
- [0., 1., 0., 0., 0.],
- [0., 0., 0., 0., 1.],
- ...,
- [0., 1., 0., 0., 0.], …

处理文本和类别属性

- 使用类LabelBinarizer，我们可以用一步执行这两个转换（从文本分类到整数分类，再从整数分类到独热向量）：

```
>>> from sklearn.preprocessing import LabelBinarizer
>>> encoder = LabelBinarizer()
>>> housing_cat_1hot = encoder.fit_transform(housing_cat)
>>> housing_cat_1hot
array([[0, 1, 0, 0, 0],
       [0, 1, 0, 0, 0],
       [0, 0, 0, 0, 1],
       ...,
```

特征缩放

- 数据要做的最重要的转换之一是特征缩放。除了个别情况，当输入的数值属性量度不同时，机器学习算法的性能都不会好。这个规律也适用于房产数据：总房间数分布范围是 6 到 39320，而收入中位数只分布在 0 到 15。注意通常情况下我们不需要对目标值进行缩放。

- 有两种常见的方法可以让所有的属性有相同的量度：线性函数归一化（Min-Max scaling）和标准化（standardization）。

转换流水线

- 你已经看到，存在许多数据转换步骤，需要按一定的顺序执行。幸运的是，Scikit-Learn 提供了类Pipeline，来进行这一系列的转换。下面是一个数值属性的小流水线：

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
num_pipeline = Pipeline([
    ('imputer', Imputer(strategy="median")),
    ('attribs_adder', CombinedAttributesAdder()),
    ('std_scaler', StandardScaler()),])
housing_num_tr = num_pipeline.fit_transform(housing_num)
```

转换流水线

```
from sklearn.pipeline import FeatureUnion
num_attribs = list(housing_num)
cat_attribs = ["ocean_proximity"]
num_pipeline = Pipeline([
('selector', DataFrameSelector(num_attribs)),
('imputer', Imputer(strategy="median")),
('attribs_adder', CombinedAttributesAdder()),
('std_scaler', StandardScaler()), ])
cat_pipeline = Pipeline([
('selector', DataFrameSelector(cat_attribs)),
('label_binarizer', LabelBinarizer()), ])
full_pipeline = FeatureUnion(transformer_list=[
("num_pipeline", num_pipeline),
("cat_pipeline", cat_pipeline), ])
```

转换流水线

```
>>> housing_prepared = full_pipeline.fit_transform(housing)
>>> housing_prepared
array([[ 0.73225807, -0.67331551, 0.58426443, ..., 0. ,
         0. , 0. ],
       [-0.99102923, 1.63234656, -0.92655887, ..., 0. ,
         0. , 0. ],
       [...]
>>> housing_prepared.shape
(16513, 17)
```

选择并训练模型

- Let’s first train a Linear Regression model:

```
from sklearn.linear_model import LinearRegression
lin_reg = LinearRegression()
lin_reg.fit(housing_prepared, housing_labels)
```

- Let’s try it out on a few instances from the training set:

```
>>> some_data = housing.iloc[:5]
>>> some_labels = housing_labels.iloc[:5]
>>> some_data_prepared = full_pipeline.transform(some_data)
>>> print("Predictions:\t", lin_reg.predict(some_data_prepared))
Predictions: [ 303104. 44800. 308928. 294208. 368704.]
>>> print("Labels:\t\t", list(some_labels))
Labels: [359400.0, 69700.0, 302100.0, 301300.0, 351900.0]
```

选择并训练模型

- Let’s measure this regression model’s RMSE on the whole training set using Scikit-Learn’s mean_squared_error function:

```
>>> from sklearn.metrics import mean_squared_error
>>> housing_predictions = lin_reg.predict(housing_prepared)
>>> lin_mse = mean_squared_error(housing_labels, housing_predictions)
>>> lin_rmse = np.sqrt(lin_mse)
>>> lin_rmse
68628.413493824875
```

选择并训练模型

- Let's train a DecisionTreeRegressor. This is a powerful model, capable of finding complex nonlinear relationships in the data. The code should look familiar by now:

```
from sklearn.tree import DecisionTreeRegressor
tree_reg = DecisionTreeRegressor()
tree_reg.fit(housing_prepared, housing_labels)
```

- Now that the model is trained, let's evaluate it on the training set:

```
>>> housing_predictions = tree_reg.predict(housing_prepared)
>>> tree_mse = mean_squared_error(housing_labels,
housing_predictions)
>>> tree_rmse = np.sqrt(tree_mse)
>>> tree_rmse
0.0
```

使用交叉验证做更佳的评估

- 评估决策树模型的一种方法是用函数train_test_split来分割训练集，得到一个更小的训练集和一个验证集，然后用更小的训练集来训练模型，用验证集来评估。
- 另一种更好的方法是使用 Scikit-Learn 的交叉验证功能。下面的代码采用了 K 折交叉验证（K-fold cross-validation）：它随机地将训练集分成十个不同的子集，成为“折”，然后训练评估决策树模型 10 次，每次选一个不用的折来做评估，用其它 9 个来做训练。结果是一个包含 10 个评分的数组：

使用交叉验证做更佳的评估

```
from sklearn.model_selection import cross_val_score
scores = cross_val_score(tree_reg, housing_prepared, housing_labels,
                         scoring="neg_mean_squared_error", cv=10)
rmse_scores = np.sqrt(-scores)
```

- Let’s look at the results:

```
>>> def display_scores(scores):
... print("Scores:", scores)
... print("Mean:", scores.mean())
... print("Standard deviation:", scores.std())
>>> display_scores(tree_rmse_scores)
Scores: [ 74678.4916885 64766.2398337 69632.86942005 …
Mean: 71199.4280043
Standard deviation: 3202.70522793
```

使用交叉验证做更佳的评估

- Let's compute the same scores for the Linear Regression model just to be sure:

```
>>> lin_scores = cross_val_score(lin_reg, housing_prepared,
housing_labels,
... scoring="neg_mean_squared_error", cv=10)
...
>>> lin_rmse_scores = np.sqrt(-lin_scores)
>>> display_scores(lin_rmse_scores)
Scores: [ 70423.5893262 65804.84913139 66620.84314068 …
Mean: 68972.377566
Standard deviation: 2493.98819069
```

- That's right: the Decision Tree model is overfitting so badly that it performs worse than the Linear Regression model.

使用交叉验证做更佳的评估

- 现在再尝试最后一个模型：RandomForestRegressor。我们会看到，随机森林是通过用特征的随机子集训练许多决策树。在其它多个模型之上建立模型称为集成学习（Ensemble Learning），它是推进 ML 算法的一种好方法。我们会跳过大部分的代码，因为代码本质上和其它模型一样：

```
>>> from sklearn.ensemble import RandomForestRegressor
>>> forest_reg = RandomForestRegressor()
>>> forest_reg.fit(housing_prepared, housing_labels)
>>> [...]
>>> forest_rmse
22542.396440343684
>>> display_scores(forest_rmse_scores)
Scores: [ 53789.2879722 50256.19806622 52521.55342602 …
Mean: 52634.1919593
Standard deviation: 1576.20472269
```

使用交叉验证做更佳的评估

- 训练集的评分仍然比验证集的评分低很多。解决过拟合可以通过简化模型，给模型加限制（即，规整化），或用更多的训练数据。
- 在深入随机森林之前，你应该尝试下机器学习算法的其它类型模型（不同核心的支持向量机，神经网络，等等），不要在调节超参数上花费太多时间。目标是列出一个可能模型的列表（两到五个）。

模型微调

- 微调的一种方法是手工调整超参数，直到找到一个好的超参数组合。这么做的话会非常冗长，你也可能没有时间探索多种组合。
- 你应该使用 Scikit-Learn 的GridSearchCV来做这项搜索工作。你所需要做的是告诉GridSearchCV要试验有哪些超参数，要试验什么值，GridSearchCV就能用交叉验证试验所有可能超参数值的组合。例如，下面的代码搜索了RandomForestRegressor超参数值的最佳组合：

模型微调

```
from sklearn.model_selection import GridSearchCV
param_grid = [
    {'n_estimators': [3, 10, 30], 'max_features': [2, 4, 6, 8]},
    {'bootstrap': [False], 'n_estimators': [3, 10], 'max_features': [2, 3, 4]},
  ]
forest_reg = RandomForestRegressor()
grid_search = GridSearchCV(forest_reg, param_grid, cv=5,
                           scoring='neg_mean_squared_error')
grid_search.fit(housing_prepared, housing_labels)
```

模型微调

- param_grid告诉 Scikit-Learn 首先评估所有的列在第一个dict中的n_estimators和max_features的3 × 4 = 12种组合。然后尝试第二个dict中超参数的2 × 3 = 6种组合，这次会将超参数bootstrap设为False而不是True（后者是该超参数的默认值）。
- 总之，网格搜索会探索12 + 6 = 18种RandomForestRegressor的超参数组合，会训练每个模型五次（因为用的是五折交叉验证）。换句话说，训练总共有18 × 5 = 90轮！K 折将要花费大量时间，完成后，你就能获得参数的最佳组合，如下所示：

模型微调

```
>>> grid_search.best_params_
{'max_features': 6, 'n_estimators': 30}

>>> grid_search.best_estimator_
RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=None,
   max_features=6, max_leaf_nodes=None, min_samples_leaf=1,
   min_samples_split=2, min_weight_fraction_leaf=0.0,
   n_estimators=30, n_jobs=1, oob_score=False, random_state=None,
   verbose=0, warm_start=False)
```

模型微调

```
>>> cvres = grid_search.cv_results_
... for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
... print(np.sqrt(-mean_score), params)
...
64912.0351358 {'max_features': 2, 'n_estimators': 3}
55535.2786524 {'max_features': 2, 'n_estimators': 10}
52940.2696165 {'max_features': 2, 'n_estimators': 30}
60384.0908354 {'max_features': 4, 'n_estimators': 3}
52709.9199934 {'max_features': 4, 'n_estimators': 10}
50503.5985321 {'max_features': 4, 'n_estimators': 30}
...
49958.9555932 {'max_features': 6, 'n_estimators': 30}
...
59634.0533132 {'bootstrap': False, 'max_features': 3, 'n_estimators': 3}
52456.0883904 {'bootstrap': False, 'max_features': 3, 'n_estimators': 10}
58825.665239 {'bootstrap': False, 'max_features': 4, 'n_estimators': 3}
52012.9945396 {'bootstrap': False, 'max_features': 4, 'n_estimators': 10}
```

模型微调

当探索相对较少的组合时，就像前面的例子，网格搜索还可以。但是当超参数的搜索空间很大时，最好使用RandomizedSearchCV。这个类的使用方法和类GridSearchCV很相似，但它不是尝试所有可能的组合，而是通过选择每个超参数的一个随机值的特定数量的随机组合。这个方法有两个优点：

- 如果你让随机搜索运行，比如 1000 次，它会探索每个超参数的 1000 个不同的值（而不是像网格搜索那样，只搜索每个超参数的几个值）。
- 你可以方便地通过设定搜索次数，控制超参数搜索的计算量。

模型微调

- 另一种微调系统的方法是将表现最好的模型组合起来。组合（集成）之后的性能通常要比单独的模型要好（就像随机森林要比单独的决策树要好），特别是当单独模型的误差类型不同时。

用测试集评估系统

- 调节完系统之后，你终于有了一个性能足够好的系统。现在就可以用测试集评估最后的模型了。这个过程没有什么特殊的：从测试集得到预测值和标签，运行full_pipeline转换数据，再用测试集评估最终模型：

```
final_model = grid_search.best_estimator_
X_test = strat_test_set.drop("median_house_value", axis=1)
y_test = strat_test_set["median_house_value"].copy()
X_test_prepared = full_pipeline.transform(X_test)
final_predictions = final_model.predict(X_test_prepared)
final_mse = mean_squared_error(y_test, final_predictions)
final_rmse = np.sqrt(final_mse) # => evaluates to 48,209.6
```

用测试集评估系统

- 评估结果通常要比交叉验证的效果差一点，如果你之前做过很多超参数微调（因为你的系统在验证集上微调，得到了不错的性能，通常不会在未知的数据集上有同样好的效果）。这个例子不属于这种情况，但是当发生这种情况时，你一定要忍住不要调节超参数，使测试集的效果变好；这样的提升不能泛化到新数据上。

启动、监控、维护系统

- 现在你可以启动系统了！你需要为实际生产做好准备，特别是接入输入数据源，并编写测试。

- 你还需要编写监控代码，以固定间隔检测系统的实时表现，当发生下降时触发报警。这对于捕获突然的系统崩溃和性能下降十分重要。做监控很常见，是因为模型会随着数据的演化而性能下降，除非模型用新数据定期训练。

启动、监控、维护系统

- 最后，你可能想定期用新数据训练模型。你应该尽可能自动化这个过程。如果不这么做，非常有可能你需要每隔至少六个月更新模型，系统的表现就会产生严重波动。如果你的系统是一个线上学习系统，你需要定期保存系统状态快照，好能方便地回滚到之前的工作状态。