

به زام خدا
خلاصه فصل
دوم کتاب
HÅND-S-ON
MACHINE
LEARNING

END-TO-END MACHINE LEARNING PROJECT

- Popular open data repositories:

- [OpenML.org](#)
- [Kaggle.com](#)
- [PapersWithCode.com](#)
- [UC Irvine Machine Learning Repository](#)
- [Amazon's AWS datasets](#)
- [TensorFlow datasets](#)

- Meta portals (they list open data repositories):

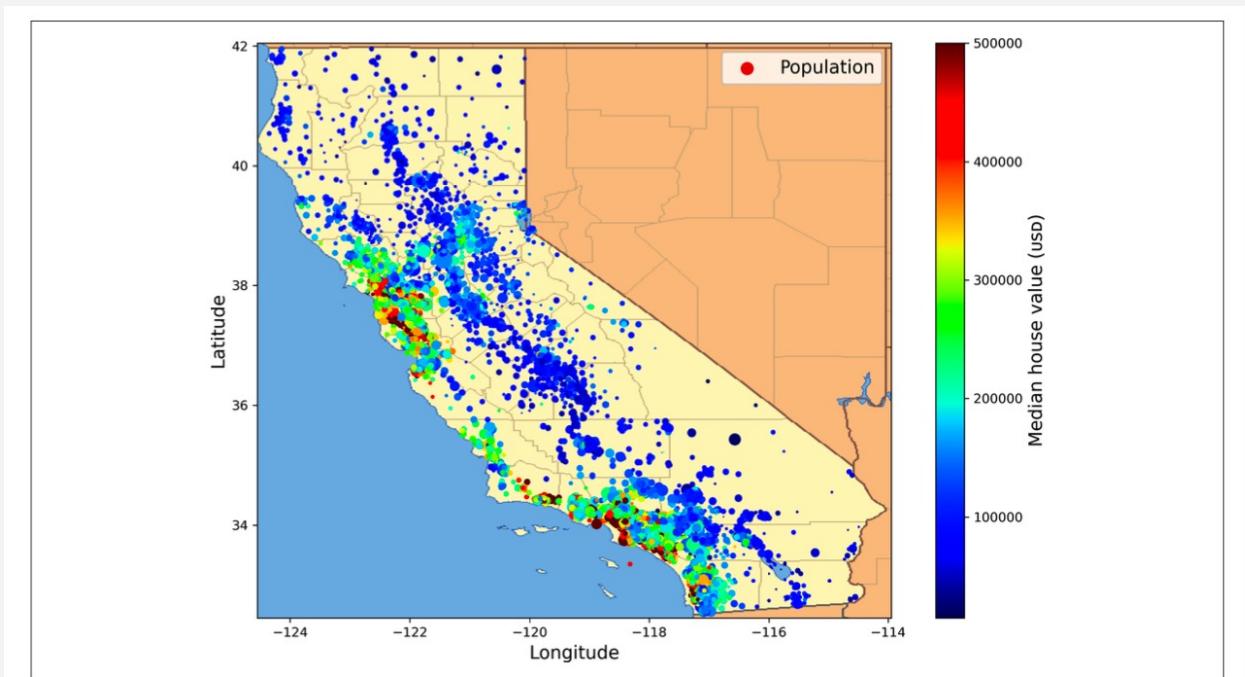
- [DataPortals.org](#)
- [OpenDataMonitor.eu](#)

- Other pages listing many popular open data repositories:

- [Wikipedia's list of machine learning datasets](#)
- [Quora.com](#)
- [The datasets subreddit](#)

- وقتی میخوایم دیتا جمع کنیم میتوانیم از سایت های زیر استفاده کنیم:

داخل این فصل با دیتاست قیمت خونه های کالیفرنیا سر و کار داریم. خوبیش اینه کلی نمونه داره. این دیتاست نمونه هاش شامل یه سری متريک مثل جمعیت، ميانگين درامد، و ... هست.



FRAME THE PROBLEM

- اولین موردی که باید بهش دقت کنیم اینه که بتونیم یه آنالیز خوب از مسئله و دیتاهامون داشته باشیم و طبق اون مدل انتخاب کنیم. برای مثال فرض کنیم میخوایم قیمت یه خونه رو پیش‌بینی کنیم طبق این خروجی باید مدل مناسب رو انتخاب کنیم. کاری که میخوایم انجام بدیم مثلاً توسط چند تا مخصوص این زمینه به صورت دستی داره محاسبه میشه که کار طاقت‌فرساییه پس میایم یه مدل برای این کار train میکنیم.
- اول مشخص میکنیم که از کدام مدل سیستم هایی که قبله دیدیم قراره استفاده کنیم. با توجه به مسئله ای که داریم باید از یه مدل بر پایه supervised learning استفاده کنیم چون هر نمونه داخل دیتاست قیمتیش رو داریم (label) ما تو این مسئله قیمت خونس). یک مسئله regression هست نه classification چون باید یه عدد حقیقی رو خروجی بدیم. در نهایت چون مدل‌مون میخواد رابطه بین فیچر هارو برای پیدا کردن یه فرمول بدست بیاره model based و از batch learning استفاده میکنیم.
- کام دوم اینه یه معیار برای اینکه پرفورمنس مدل چقدره انتخاب کنیم. که داخل مسائل regression معمولاً RMSE (root mean square error) استفاده میکنیم. که خوبیش اینه برای اختلاف های زیاد مقدار ارور رو بیشتر از حد عادی میکنه (توان^۲)

Equation 2-1. Root mean square error (RMSE)

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

NOTATIONS

Notations

This equation introduces several very common machine learning notations that I 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 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 $(\mathbf{x}^{(i)})^T$.³
 - For example, if the first district is as just described, then the matrix \mathbf{X} looks like this:

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

- 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 $\hat{y}^{(1)} - y^{(1)} = 2,000$.
- $\text{RMSE}(\mathbf{X}, h)$ 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}).

CONTINUE

- تنها تابعی که برای اندازه‌گیری پرفورمنس میتوانیم به کار ببریم RMSE نیست و از مواردی مثل (mean MAE) استفاده کنیم.

Equation 2-2. Mean absolute error (MAE)

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

- هرکدامشون برای یه کیسی مورد استفاده قرار میگیره. مثلا جاهایی که outlier خیلی زیاد نداریم میتوانیم از RMSE استفاده کنیم. یه مورد دیگه که هستش اینه که RMSE خیلی روی عدد های بزرگ حساس تره تا عدد های کوچیک مثلا وقتی فاصله رو ۰.۵ در نظر بگیره به توان دو میرسونه و نصف میشه حتی. پس نکته ای که هست اینه از هر کدام با توجه به ماهیت پروژه استفاده کنیم.

DOWNLOAD AND WORK WITH DATA

- اولین کاری که باید انجام بدیم دانلود کردن دیتامون که یک فایل CSV هستش و میتوانیم این کار رو با نوشتن یه تابع انجام بدیم. کارش اینه اول ببیاد دیتاست رو داریم یا نه، اگه نبود از github دانلود کنه.

Here is the function to fetch and load the data:

```
from pathlib import Path
import pandas as pd
import tarfile
import urllib.request

def load_housing_data():
    tarball_path = Path("datasets/housing.tgz")
    if not tarball_path.is_file():
        Path("datasets").mkdir(parents=True, exist_ok=True)
        url = "https://github.com/ageron/data/raw/main/housing.tgz"
        urllib.request.urlretrieve(url, tarball_path)
        with tarfile.open(tarball_path) as housing_tarball:
            housing_tarball.extractall(path="datasets")
    return pd.read_csv(Path("datasets/housing/housing.csv"))

housing = load_housing_data()
```

هر ردیف نشون دهنده یک sample هستش .
همچنین هر ردیف ستون نشون دهنده یه
ج feature دیتامون هستش.

housing.head()						
	longitude	latitude	housing_median_age	median_income	ocean_proximity	median_house_value
0	-122.23	37.88	41.0	8.3252	NEAR BAY	452600.0
1	-122.22	37.86	21.0	8.3014	NEAR BAY	358500.0
2	-122.24	37.85	52.0	7.2574	NEAR BAY	352100.0
3	-122.25	37.85	52.0	5.6431	NEAR BAY	341300.0
4	-122.25	37.85	52.0	3.8462	NEAR BAY	342200.0

Figure 2-6. Top five rows in the dataset

CONTINUE

- اگه بخوايم همه feature های دیتاست و کلا يه اطلاعات کلی از دیتاست داشته باشيم از متده info استفاده ميکنيم.

```
>>> housing.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 10 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   longitude        20640 non-null   float64
 1   latitude         20640 non-null   float64
 2   housing_median_age 20640 non-null   float64
 3   total_rooms       20640 non-null   float64
 4   total_bedrooms    20433 non-null   float64
 5   population        20640 non-null   float64
 6   households        20640 non-null   float64
 7   median_income     20640 non-null   float64
 8   median_house_value 20640 non-null   float64
 9   ocean_proximity   20640 non-null   object  
dtypes: float64(9), object(1)
memory usage: 1.6+ MB
```

طبق اطلاعاتی که داره نشون میده، اطلاعاتی که داریم شامل 20640 نمونه هستش. از نکات دیگه ای که ميفهمم فيچر total_bedrooms در 26433 نمونه مقدار داره و بقيشون نال هست. باید مراقب اين قضيه باشيم. ocean_proximity از جتس عددن به جز فيچر دیتاست رو ببينيم ميفهمم که اين فيچر از جنس متن هست و احتمالا شامل يه تعداد دسته بندی محدود هست که ميتونی مقاديرش رو به شکل زير ببينيم:

```
>>> housing["ocean_proximity"].value_counts()
<1H OCEAN      9136
INLAND          6551
NEAR OCEAN      2658
NEAR BAY         2290
ISLAND            5
Name: ocean_proximity, dtype: int64
```

CONTINUE

- با متدهای `describe` میتوانیم یه اطلاعات خلاصه از هر فیچر بدست بیاریم.

housing.describe()						
	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	median_house_value
count	20640.000000	20640.000000	20640.000000	20640.000000	20433.000000	20640.000000
mean	-119.569704	35.631861	28.639486	2635.763081	537.870553	206855.816909
std	2.003532	2.135952	12.585558	2181.615252	421.385070	115395.615874
min	-124.350000	32.540000	1.000000	2.000000	1.000000	14999.000000
25%	-121.800000	33.930000	18.000000	1447.750000	296.000000	119600.000000
50%	-118.490000	34.260000	29.000000	2127.000000	435.000000	179700.000000
75%	-118.010000	37.710000	37.000000	3148.000000	647.000000	264725.000000
max	-114.310000	41.950000	52.000000	39320.000000	6445.000000	500001.000000

Figure 2-7. Summary of each numerical attribute

The `count`, `mean`, `min`, and `max` rows are self-explanatory. Note that the null values are ignored (so, for example, the `count` of `total_bedrooms` is 20,433, not 20,640). The `std` row shows the *standard deviation*, which measures how dispersed the values

CONTINUE

- یه راه دیگه برای اینکه بتوانیم یه اطلاعات کلی از فیچر هامون داشته باشیم اینه که هرگدوم رو plot کنیم.

```
import matplotlib.pyplot as plt  
  
housing.hist(bins=50, figsize=(12, 8))  
plt.show()
```

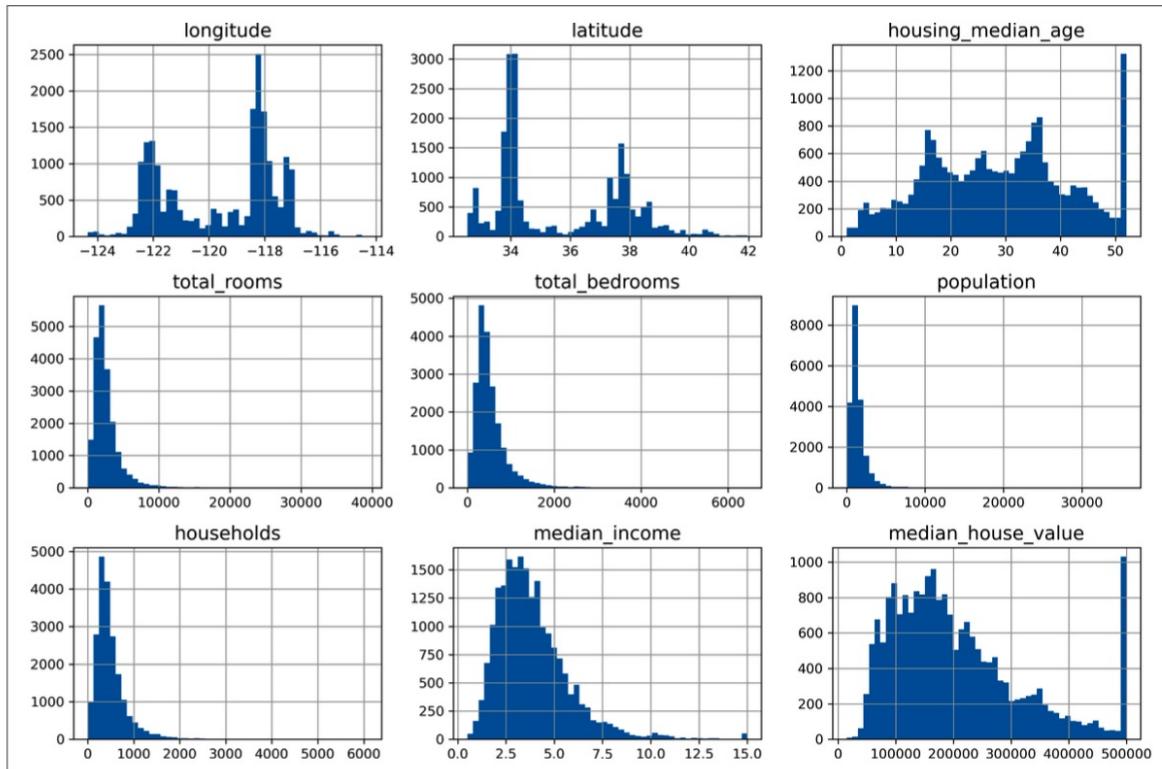


Figure 2-8. A histogram for each numerical attribute

از همین هستوگرام یه سری موضوعات میفهمم مثلاً فیچر median_income یه scale روش اتفاق افتاده و مقادیر حقیقی هر سمپل مثلًا تقسیم به ۱۵۰۰۰ دلار شدن. این موارد اوکیه صرفا باید حواسمن باشه که مقدار خروجی هم ایا اسکیل شده یا نه اگه آره این مورد رو باید داخل فرض هامون قرار بدیم.

CREATE A TESTSET

- باید دقت کنیم که نیاز به یک test set هم داریم که مهمه به چه شکل از training set جداش کنیم. نکته مهمی که داره اینه که test set به یه شکلی باشه که یه سری از خاص نمونه ها رو داشته باشه و بقیه نمونه های جامعه آماری رو نداشته باشه اول از هم regularization میاد پایین و بدیش اینه باعث انتخاب کردن یه مدلی میشه که صرفاً میتونه یه مدلی رو انتخاب کنه که میتوانه اون نمونه های خاص رو خوب جواب بده.
- یه روش اینه به طور زندوم بیایم یه سری از دیتاهای داخل data set رو برداریم و test set رو تشکیل بدیم.

```
import numpy as np

def shuffle_and_split_data(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]
```

You can then use this function like this:

```
>>> train_set, test_set = shuffle_and_split_data(housing, 0.2)
>>> len(train_set)
16512
>>> len(test_set)
4128
```

بدی این روش اینه که اگه دوباره این تابع رو فراخونی کنیم یه test set متفاوت بهمون میده و در طول زمان باعث میشه مدلمون test set های مختلف رو ببینه و از روشون یادبگیره (مفهوم test set بودن رو داره از دست میده بیشتر شبیه training set میشه). راهکار اینه که یه ارگومان seed رو مساوی یه عدد خاص بذاریم تا باعث بشه هر سری test set یکسان بهمون بده. اما این روش هم یه بدی که داره اینه اگه data set اپدیت بشه، اونوقت مشکل بالا دوباره رخ میده. راه حل اینه یه سیستم هش طور پیاده کنیم روی یک یا چند تا فیچر تا اگه مقدار هشی که برمیگردونه به ازای هر نمونه از یه حدی کمتر یا بیشتر بود بر اساس اون تصمیم بگیره که داخل test set بذاره یا نه.

CONTINUE

- یه خوبی که خودش اومده همچین تابعی رو پیاده سازی کرده و خیلی راحت میتونیم ازش استفاده کنیم:

```
from sklearn.model_selection import train_test_split  
  
train_set, test_set = train_test_split(housing, test_size=0.2, random_state=42)
```

- این متده که به طور رندوم نمونه انتخاب کنیم برای test set و train set اوکیه ولی نکته ای که باید بهش توجه کنیم اینه که وقتی میتونه هر کدام یک نمونه خوب از جامعه آماری ما باشه که به اندازه کافی data set بزرگ باشه اگه نباشه ممکنه bias رخ بده و داخل test set مثلا یه سری سمپل که نزدیک به همن و بقیه جامعه رو نشون نمیده باشه.
- فرض کنید یه فیچر رو حتما بخوایم مطمئن بشیم که مشکل بالا برash پیش نمیاد، برای مثال میانگین درآمد. چون یک فیچر عددی پیوسته هستش اول نیاز داریم به کتگوری های مختلف این اعداد رو تقسیم کنیم.

```
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])
```

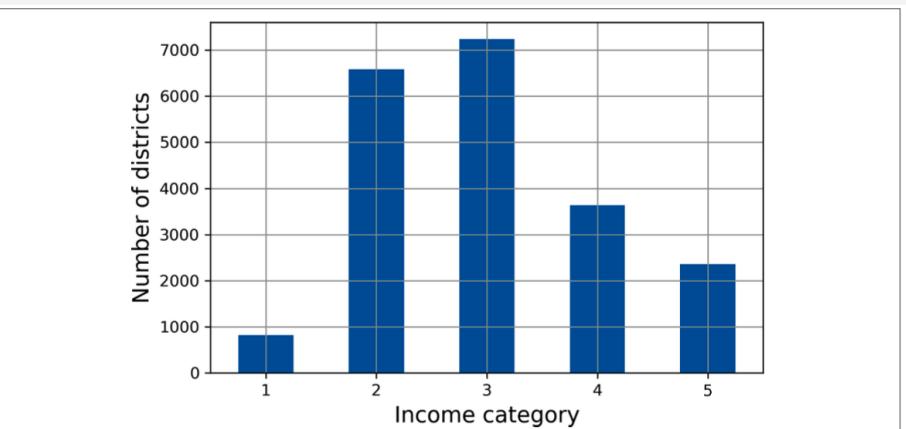


Figure 2-9. Histogram of income categories

CONTINUE

- حالا که دسته بندی رو انجام دادیم و فراوانی اعضا هر دسته مشخصه میتوانیم از قابلیت scikit-learn استفاده کنیم و training set و test set دقیقا با همین درصد بهمون بده.

```
strat_train_set, strat_test_set = train_test_split(  
    housing, test_size=0.2, stratify=housing["income_cat"], random_state=42)
```

- نتیجه این کار رو با روشی که رندوم انتخاب میکردیم:

Income Category	Overall %	Stratified %	Random %	Strat. Error %	Rand. Error %
1	3.98	4.00	4.24	0.36	6.45
2	31.88	31.88	30.74	-0.02	-3.59
3	35.06	35.05	34.52	-0.01	-1.53
4	17.63	17.64	18.41	0.03	4.42
5	11.44	11.43	12.09	-0.08	5.63

حالا میتوانیم فیچر income_cat که خودمون درست کرده بودیم رو پاک کنیم

```
for set_ in (strat_train_set, strat_test_set):  
    set_.drop("income_cat", axis=1, inplace=True)
```

EXPLORE AND VISUALIZE DATA TO GAIN INSIGHT

- خوبه که دیتا رو ویژوالایز کنیم تا ازش یه سری اطلاعات بیشتر کسب کنیم یا به طور کلی یه نگرش خیلی بهتری نسبت به مسئله داشته باشیم. برای مسئله قیمت خونه چون طول و عرض جغرافیایی داریم خوبه با این فیچر ها پلات کنیم دیتابامون رو.

```
housing.plot(kind="scatter", x="longitude", y="latitude", grid=True)  
plt.show()
```

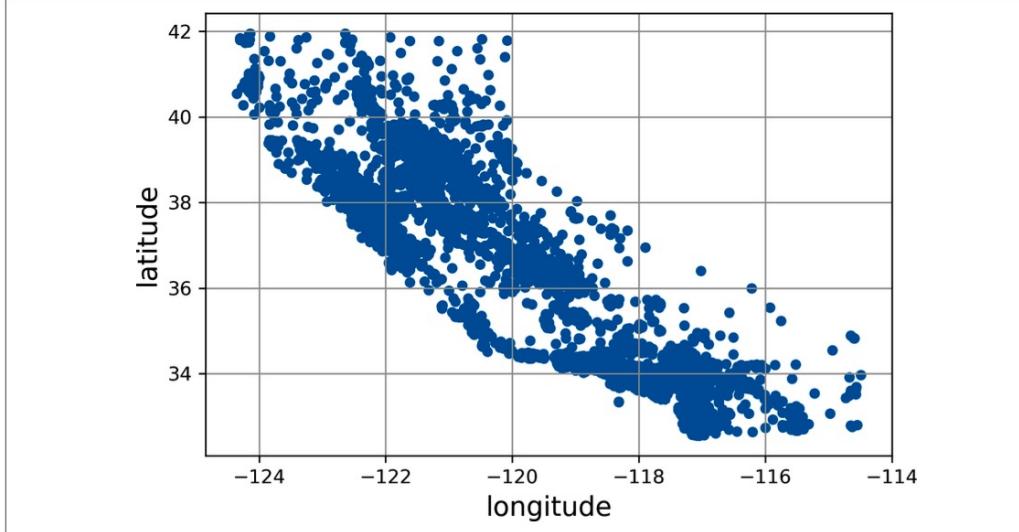


Figure 2-11. A geographical scatterplot of the data

اگه پارامتر آلفا رو مقداردهی کنیم داخل تابع `plot` باعث میشه مناطق پرتراکم و کم تراکم رو از هم تفکیک کنه.

```
housing.plot(kind="scatter", x="longitude", y="latitude", grid=True, alpha=0.2)  
plt.show()
```

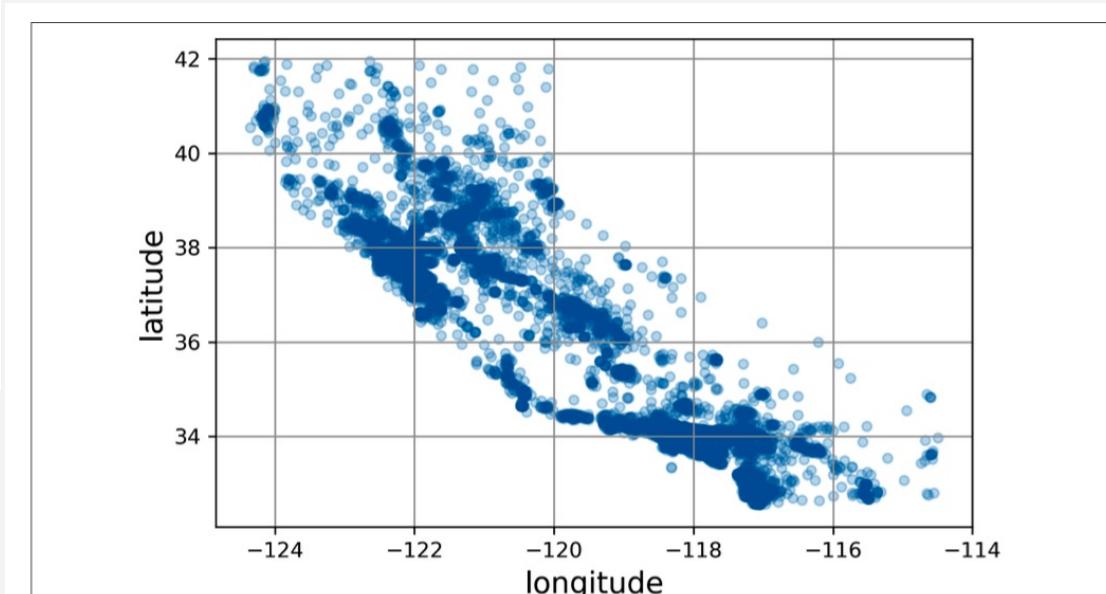


Figure 2-12. A better visualization that highlights high-density areas

CONTINUE

```
housing.plot(kind="scatter", x="longitude", y="latitude", grid=True,
             s=housing["population"] / 100, label="population",
             c="median_house_value", cmap="jet", colorbar=True,
             legend=True, sharex=False, figsize=(10, 7))

plt.show()
```

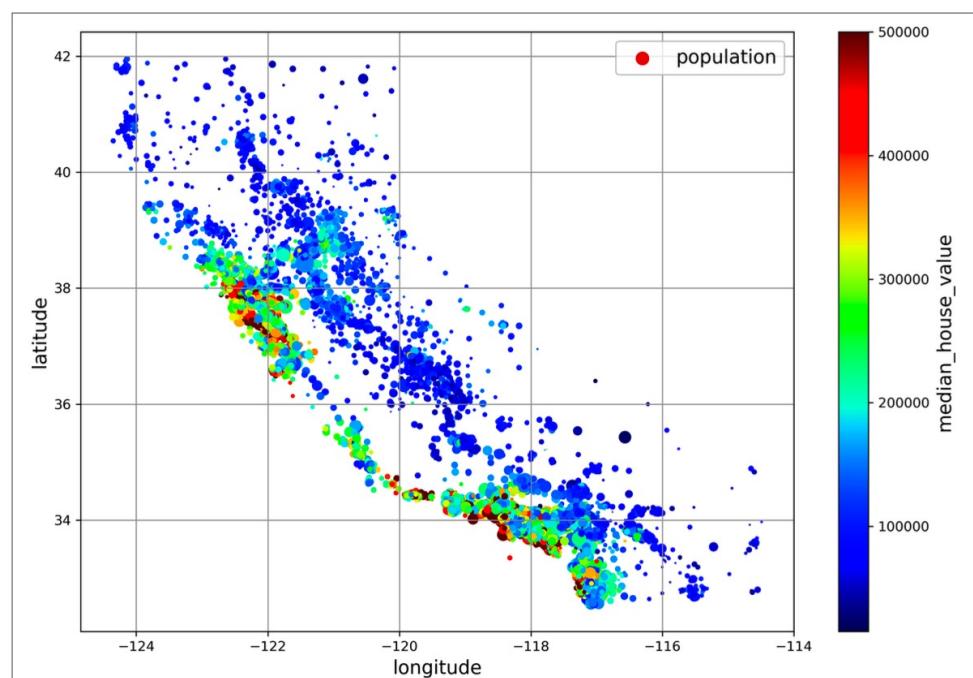


Figure 2-13. California housing prices: red is expensive, blue is cheap, larger circles indicate areas with a larger population

- حالا میتوانیم جمعیت و قیمت رو هم ویژوالایز کنیم: هرچی پرمجمعیت تر شعاع دایره بزرگ تر و هرچی گرون تر رنگ نزدیک به قرمز.

از این پلات کردن به یه سری نتایج میرسیم: قیمت خونه خیلی به لوکیشن خونه ها و جمعیت اون منطقه مرتبطه. برای مثال هرچقدر به اقیانوس نزدیک تر باشه خونه ها قیمت بیشتری دارن تا نسبت به خونه هایی که به نواحی مرکزی نزدیک ترن. پس یه سری ارتباط رو خودمون پیدا کردیم.

LOOK FOR CORRELATIONS

- فرض کنیم که دیتامون پیچیده تر از این حرف باشه که بتوانیم خودمون ارتباط بین فیچر ها با خروجیمون رو بفهمیم، پس ازتابع corr استفاده میکنیم تا بتونه این ارتباطات رو برآمون پیدا کنه. (ماتریس کوواریانس رو میسازه)

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

Now you can 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      0.688380
total_rooms        0.137455
housing_median_age  0.102175
households         0.071426
total_bedrooms     0.054635
population        -0.020153
longitude          -0.050859
latitude           -0.139584
Name: median_house_value, dtype: float64
```

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

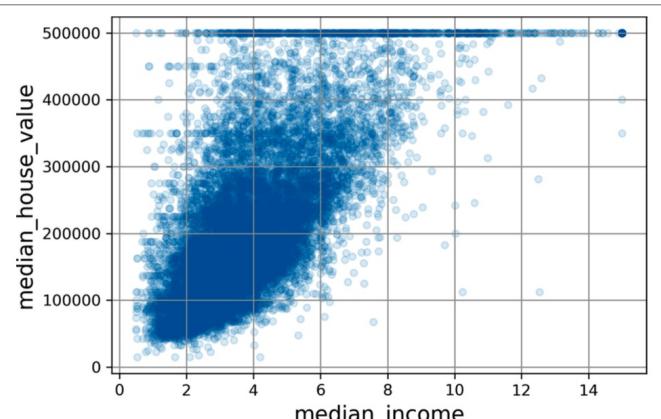


Figure 2-15. Median income versus median house value

هرچی ارتباط ما عددش به یک نزدیک باشه نشون میده که دو تا فیچر ارتباط مستقیم بیشتری با هم دارن و هرچقدر این عدد به منفی یک نزدیک باشه نشون میده این ارتباط برعکسه. و عدد ۰ به معنی خنثی بودن (نبود یک ارتباط خطی بین دو فیچر). بدی این روش اینه که فقط ارتباط های خطی رو پیدا میکنه و ارتباط های غیرخطی رو نمیتونه بفهمه.

همچنین میتونیم ارتباط بین فیچر با خروجی رو به صورت یک پلات دو بعدی که روی محور X فیچر و روی محور Y خروجیمون باشه تا بتوانیم یه شهودی ازش بگیریم

از این نمودار چند تا چیز رو میتونیم بفهمیم. یکی اینکه یه ارتباط مستقیم وجود داره و با بالاتر رفتن نمودار از سمت X ها y هم بالا رفته. دومی اینکه دو تا خط صاف تو نواحی 450k , 500k و ... هست که میتونیم این نمونه هارو حذف کنیم (یه جورایی میتونیم تشخیص بدیم داده پرت هستن)

EXPERIMENT WITH ATTRIBUTE COMBINATIONS

- شاید بخوایم ترکیبی از فیچر هارو امتحان کنیم. چرا؟ برای مقال به تنها بی تعداد اتاق خواب معیار خوبی نباشه و معیار تعداد اتاق خواب به ازای افراد خونه یا به ازای تعداد کل اتاق های خونه معیار بهتری باشه. پس میتونیم همچین فیچر هایی رو تشکیل بدیم:

```
housing["rooms_per_house"] = housing["total_rooms"] / housing["households"]
housing["bedrooms_ratio"] = housing["total_bedrooms"] / housing["total_rooms"]
housing["people_per_house"] = housing["population"] / housing["households"]
```

- حالا اگه correlation بگیریم میبینیم که فیچر های جدید دارن ارتباط بیشتری رو مشخص میکنن برامون:

```
>>> corr_matrix = housing.corr()
>>> corr_matrix["median_house_value"].sort_values(ascending=False)
median_house_value    1.000000
median_income        0.688380
rooms_per_house     0.143663
total_rooms          0.137455
housing_median_age   0.102175
households           0.071426
total_bedrooms       0.054635
population          -0.020153
people_per_house     -0.038224
longitude            -0.050859
latitude             -0.139584
bedrooms_ratio       -0.256397
Name: median_house_value, dtype: float64
```

PREPARE DATA AND CLEAN DATA

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

- اول از همه لیبل دیتاهامون رو جدا میکنیم چون قراره به عنوان y ها شناخته بشه

- دوم از همه باید دیتاهایی که باهاشون کار میکنیم به قول گفتی تمیز باشن. یعنی چی؟ یعنی اگه یه sample یک سری از فیچرهاش مقدار ندارن باید یه فکری به حالشون کنیم. مثلًا مقدار اون فیچر رو صفر یا میانگین یا ... بذاریم. یا اینکه اون sample که فیچرش دیتا نداره رو پاک کنیم.

```
median = housing["total_bedrooms"].median() # option 3
housing["total_bedrooms"].fillna(median, inplace=True)
```

```
housing.dropna(subset=["total_bedrooms"], inplace=True)
```

- برای آپشن اینکه بخوایم مقادیر رو با میانگین یا مد یا ... پر کنیم یه راه بهتری هم هست اونم استفاده از imputer که خوبیش اینه میاد این پارامتر آماری رو برای هر فیچر حساب میکنه و اگه بعدا یه دیتا داشتیم از هرجایی میتوانیم سریع با استفاده از imputer مقادیرش رو پر کنیم.

```
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(strategy="median")
```

CONTINUE

- از اونجایی که میانگین یک ویژگی برای اعداده پس باید روی فیچر های عددی این حرکت رو بزنیم، اول یه دیتاست که شامل فقط فیچر های عددی باشه درست میکنیم بعد به imputer میدیم تا مقادیر میانگین رو بدست بیاره بعد کل دیتاست رو با استفاده از این میایم مقادیری که خالیه رو جای گذاری میکنیم

```
housing_num = housing.select_dtypes(include=[np.number])
```

Now you can fit the `imputer` instance to the training data using the

```
imputer.fit(housing_num)
```

```
X = imputer.transform(housing_num)
```

- صرف نکته ای که میمونه اینه که X الان فقط یه آرایه از مقادیر عددی و متنی هستش و ستون های ما که اسم فیچر ها بودن پریدن اما خیلی کار سختی نیست بخوایم این موارد رو اضافه کنیم:

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

HANDLING TEXT AND CATEGORICAL ATTRIBUTES

- تا اینجا فقط با فیچر های عددی کار کردیم اگه یه فیچر متنی داشتیم چی کار کنیم؟ برای مثال تو دیتاستی که روش کار میکنیم "نزدیکی به اقیانوس" یه فیچر متنی بود که شامل یه سری مقدار های محدود بود که میتونیم به کتگوری های مختلف تقسیم بندی کنیم، اکثر الگوریتم های ماشین لرنینگ ترجیح میدن با عدد کار کنن جای متن پس بیایم این موارد رو تبدیل کنیم به عدد.
- یه راه اینه که بیایم از encoder استفاده کنیم و به هر کتگوری یه عدد نسبت بده.

```
>>> housing_cat = housing[["ocean_proximity"]]
>>> housing_cat.head(8)
   ocean_proximity
13096      NEAR BAY
14973    <1H OCEAN
3785       INLAND
14689       INLAND
20507    NEAR OCEAN
1286       INLAND
18078    <1H OCEAN
4396      NEAR BAY
```

```
from sklearn.preprocessing import OrdinalEncoder
ordinal_encoder = OrdinalEncoder()
housing_cat_encoded = ordinal_encoder.fit_transform(housing_cat)

Here's what the first few encoded values in housing_cat_encoded look like:

>>> housing_cat_encoded[:8]
array([[3.],
       [0.],
       [1.],
       [1.],
       [4.],
       [1.],
       [0.],
       [3.]])
```

CONTINUE

- روش قبلی روش بدی نیست ولی یه بدی داره اونم اینکه کتگوری هایی که عددهشون نزدیک به هم هستن رو فرض میکنه که به هم شباخت دارن که فرض نسبتاً اشتباھیه مگه اینکه کتگوری هارو به صورت مرتب میچیديم. تو مثال ما همچین فرضی نیست پس دنبال راه حل بعدی هستیم که استفاده از onehot encoder که استفاده از onehot encoder هستش.

```
from sklearn.preprocessing import OneHotEncoder  
  
cat_encoder = OneHotEncoder()  
housing_cat_1hot = cat_encoder.fit_transform(housing_cat)
```

- خروجی این کار یه ماتریس اسپارسه که باید تبدیلش کنیم به یه آرایه دو بعدی تا بتونیم بینیم ماتریس رو.

```
>>> housing_cat_1hot.toarray()  
array([[0., 0., 0., 1., 0.],  
       [1., 0., 0., 0., 0.],  
       [0., 1., 0., 0., 0.],  
       ...,  
       [0., 0., 0., 0., 1.],  
       [1., 0., 0., 0., 0.],  
       [0., 0., 0., 0., 1.]])
```

But OneHotEncoder is smarter: it will detect the unknown category and raise an exception. If you prefer, you can set the handle_unknown hyperparameter to "ignore", in which case it will just represent the unknown category with zeros:

```
>>> cat_encoder.handle_unknown = "ignore"  
>>> cat_encoder.transform(df_test_unknown)  
array([[0., 0., 0., 0., 0.],  
       [0., 0., 1., 0., 0.]])
```



If a categorical attribute has a large number of possible categories (e.g., country code, profession, species), then one-hot encoding will result in a large number of input features. This may slow down training and degrade performance. If this happens, you may want to replace the categorical input with useful numerical features related to the categories: for example, you could replace the ocean_proximity feature with the distance to the ocean (similarly, a country code could be replaced with the country's population and GDP per capita). Alternatively, you can use one of the encoders provided by the category_encoders package on [GitHub](#). Or, when dealing with neural networks, you can replace each category with a learnable, low-dimensional vector called an *embedding*. This is an example of *representation learning* (see Chapters 13 and 17 for more details).

```
>>> cat_encoder.feature_names_in_  
array(['ocean_proximity'], dtype=object)  
>>> cat_encoder.get_feature_names_out()  
array(['ocean_proximity_<1H OCEAN', 'ocean_proximity_INLAND',  
      'ocean_proximity_ISLAND', 'ocean_proximity_NEAR BAY',  
      'ocean_proximity_NEAR OCEAN'], dtype=object)  
>>> df_output = pd.DataFrame(cat_encoder.transform(df_test_unknown),  
...                                         columns=cat_encoder.get_feature_names_out(),  
...                                         index=df_test_unknown.index)  
...  
...
```

FEATURE SCALING AND TRANSFORMATION

- یه نکته مهمی که وجود داره اینه که الگوریتم های ماشین لرینیگ خیلی عملکرد خوبی ندارن وقتی که فیچر های عددیمون scale های خیلی متفاوتی دارن. مثلا مجموع تعداد اتاق ها میتونه از ۶ باشه تا ۱۰۰۰ تا ولی میانگین درامد دیدیم یه عدد بین ۰ تا ۱۵ بود. اینشکلی مدل بایاس پیدا میکنه به سمت فیچری که عدد های بزرگ تری و بیشتر روی اون فیچر تمرکز میکنه. پس نیاز داریم scale همه تو یه رنج باشه، برای این کار ۲ روش داریم:
min-max scaling – standardization
- از استاندارد سازی استفاده میکنه و عددامون رو بین یه رنج خاص قرار میده. این کارو با کم کردن مقدار هر سمپل از min سمپل ها و تقسیم اون به اختلاف بین max و min انجام میده. میشه این شکلی پیاده کرد:

ference between the min and the max. Scikit-Learn provides a transformer called `MinMaxScaler` for this. It has a `feature_range` hyperparameter that lets you change the range if, for some reason, you don't want 0–1 (e.g., neural networks work best with zero-mean inputs, so a range of -1 to 1 is preferable). It's quite easy to use:

```
from sklearn.preprocessing import MinMaxScaler  
  
min_max_scaler = MinMaxScaler(feature_range=(-1, 1))  
housing_num_min_max_scaled = min_max_scaler.fit_transform(housing_num)
```

CONTINUE

- ۲. در این متده دقيقا از مفهوم استاندارد سازی در آمار استفاده ميکنیم به اين شكل هر سمپل رو منها ميانگين کرده و به انحراف معیار تقسیم ميکنیم که نتیجتا يك جامعه اماری با ميانگین صفر و واريанс يك داریم. در اين روش داده های ما لزوما بین دو خاص نیستش. و خوبی که اين متده نسبت به قبلی داره کمتر حساس بودن اون به داده های پرت هستش چون در متده قبلی بخاطر اون داده پرت مجبوریم اونم داخل scale صفر تا يك بیاریم. ولی تو این متده نه. پیاده سازی:

```
from sklearn.preprocessing import StandardScaler
```

```
std_scaler = StandardScaler()  
housing_num_std_scaled = std_scaler.fit_transform(housing_num)
```

- اگه يه فيچر خيلي داده هاي رو داشته باشن که از ميانگين دوره (واريانس زيادي داره) اول باید از مقاديرش يا راديکال يا لگاريتم بگيريم بعد دو تا متده دو تا متده قبل رو انجام بدیم.

exponentially less frequent. Figure 2-17 shows how much better this feature looks when you compute its log: it's very close to a Gaussian distribution (i.e., bell-shaped).

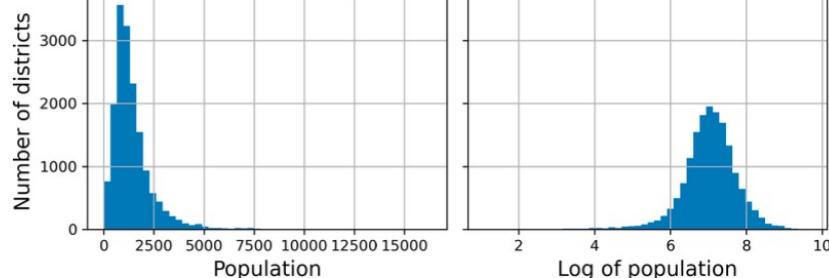


Figure 2-17. Transforming a feature to make it closer to a Gaussian distribution

CONTINUE

- همینجوری که فیچر های ورودی رو عوض کرد همین کارو برای خروجیمون (اگه دیتاهای دور از میانیگین زیاد داشته باش) انجام بدیم. میتوانیم از متدهای قبلی مثل لگاریتم گرفتن استفاده کنیم ولی باید دقت کنیم که الان مدلمون دیتا مقیاس شده رو میتونه پیشبینی کنه پس باید عددی که مدل بهمون میده رو به مقدار درستش برسونیم. که این کارو scikit-learn با استفاده از متدهای inverse_transform انجام میده.

```
from sklearn.linear_model import LinearRegression

target_scaler = StandardScaler()
scaled_labels = target_scaler.fit_transform(housing_labels.to_frame())

model = LinearRegression()
model.fit(housing[["median_income"]], scaled_labels)
some_new_data = housing[["median_income"]].iloc[:5] # pretend this is new data

scaled_predictions = model.predict(some_new_data)
predictions = target_scaler.inverse_transform(scaled_predictions)
```

اپشن بعدیمون اینه از یه کلاس دیگه استفاده کنیم که مستقیما همین کارو برامون انجام میده .

```
from sklearn.compose import TransformedTargetRegressor

model = TransformedTargetRegressor(LinearRegression(),
                                    transformer=StandardScaler())
model.fit(housing[["median_income"]], housing_labels)
predictions = model.predict(some_new_data)
```

CUSTOM TRANSFORMATION

- یه بخش کتاب هم او مده توی چند صفحه این مفهوم رو توضیح داده که اگه transformation که میخوایم انجام بدیم از شکل اینکه از تابع های پیشفرض استفاده نکنیم و یه سری عمل خاص انجام بدیم مثلا scale رو به جای ۰ تا ۱ بین ۰ تا ۱۰۰ بذاریم یا یه سری فیچر رو با هم میکس کنیم. تمامی این کارها نیاز داره که یه سری ابجکت transformer درست کنیم که این کارهارو برامون انجام بده.
- اینجا صرفا یه مثال میدارم برای اطلاعات بیشتر از خود کتاب میشه کامل خوند. برای این کار باید یه کلاس درست کنیم که از ۲ کلاس ارث بری میکنه و حتما باید دو متده fit و transform رو پیاده کنیم. داخل fit یه سری پارامتر یادمیگیره از دیتامون (برای مثال میانگین) و داخل transform اون کاری که دوست داریم رو انجام میدیم.

```
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.utils.validation import check_array, check_is_fitted

class StandardScalerClone(BaseEstimator, TransformerMixin):
    def __init__(self, with_mean=True): # no *args or **kwargs!
        self.with_mean = with_mean

    def fit(self, X, y=None): # y is required even though we don't use it
        X = check_array(X) # checks that X is an array with finite float values
        self.mean_ = X.mean(axis=0)
        self.scale_ = X.std(axis=0)
        self.n_features_in_ = X.shape[1] # every estimator stores this in fit()
        return self # always return self!

    def transform(self, X):
        check_is_fitted(self) # looks for learned attributes (with trailing _)
        X = check_array(X)
        assert self.n_features_in_ == X.shape[1]
        if self.with_mean:
            X = X - self.mean_
        return X / self.scale_
```

TRANSFORMATION PIPELINE

- همینطور که میشه حدس زد نیاز هست يه سری دیتا transformation های مختلف رو دیتامون به ترتیب انجام بدیم. برای اینکه این مشکل حل بشه scikit-learn او مده يه کلاس معرفی کرده که میتونیم يه نمونه ساده از دو تا transformation هم رو این شکلی پیاده کنیم:

```
from sklearn.pipeline import Pipeline

num_pipeline = Pipeline([
    ("impute", SimpleImputer(strategy="median")),
    ("standardize", StandardScaler()),
])

```

- پایپ لاین بالا این شکلیه اول simpleimputer روی دیتامون يه سری کارا انجام میده بعد دیتایی که درست کرده رو پاس میده تا فعالیت خودش رو انجام بده. حالا میتونیم ببینیم چی کار کرده:

```
>>> housing_num_prepared = num_pipeline.fit_transform(housing_num)
>>> housing_num_prepared[:2].round(2)
array([[-1.42,  1.01,  1.86,  0.31,  1.37,  0.14,  1.39, -0.94],
       [ 0.6 , -0.7 ,  0.91, -0.31, -0.44, -0.69, -0.37,  1.17]])
```

CONTINUE

قبل‌ا دیدیم که به چه شکل می‌توانیم با فیچر‌های غیر عددی کار کنیم حالا اگه بخوایم یه transformation برای فیچر‌های عددی و یه مورد دیگه برای فیچر‌های غیر عددی تشکیل بدیم چی کار کنیم؟ راهش اینه فیچر‌های عددی و غیر عددی رو مشخص کنیم و نسبت به هر کدام انتخاب کنیم چه transformation چه انتخاب کنیم:

```
from sklearn.compose import ColumnTransformer

num_attribs = ["longitude", "latitude", "housing_median_age", "total_rooms",
               "total_bedrooms", "population", "households", "median_income"]
cat_attribs = ["ocean_proximity"]

cat_pipeline = make_pipeline(
    SimpleImputer(strategy="most_frequent"),
    OneHotEncoder(handle_unknown="ignore"))

preprocessing = ColumnTransformer([
    ("num", num_pipeline, num_attribs),
    ("cat", cat_pipeline, cat_attribs),
])
```

```
from sklearn.compose import make_column_selector, make_column_transformer

preprocessing = make_column_transformer(
    (num_pipeline, make_column_selector(dtype_include=np.number)),
    (cat_pipeline, make_column_selector(dtype_include=object)),
)
```

Now we're ready to apply this ColumnTransformer to the housing data:

```
housing_prepared = preprocessing.fit_transform(housing)
```

از اونجایی که شاید نوشتن اسم همه فیچر‌ها کار منطقی نباشه می‌توانیم بر اساس دیتا تایپ هر فیچر انتخاب کنیم چه transformation انجام بشه.

ALL WE DONE IN A SINGLE CODE

```
def column_ratio(X):
    return X[:, [0]] / X[:, [1]]

def ratio_name(function_transformer, feature_names_in):
    return ["ratio"] # feature names out

def ratio_pipeline():
    return make_pipeline(
        SimpleImputer(strategy="median"),
        FunctionTransformer(column_ratio, feature_names_out=ratio_name),
        StandardScaler())

log_pipeline = make_pipeline(
    SimpleImputer(strategy="median"),
    FunctionTransformer(np.log, feature_names_out="one-to-one"),
    StandardScaler())
cluster_simil = ClusterSimilarity(n_clusters=10, gamma=1., random_state=42)
default_num_pipeline = make_pipeline(SimpleImputer(strategy="median"),
                                      StandardScaler())
preprocessing = ColumnTransformer([
    ("bedrooms", ratio_pipeline(), ["total_bedrooms", "total_rooms"]),
    ("rooms_per_house", ratio_pipeline(), ["total_rooms", "households"]),
    ("people_per_house", ratio_pipeline(), ["population", "households"]),
    ("log", log_pipeline, ["total_bedrooms", "total_rooms", "population",
                           "households", "median_income"]),
    ("geo", cluster_simil, ["latitude", "longitude"]),
    ("cat", cat_pipeline, make_column_selector(dtype_include=object)),
],
    remainder=default_num_pipeline) # one column remaining: housing_median_age
```

SELECT MODEL, TRAIN AND EVALUTE IT

- با تموم کارهایی که کردیم حالا دیتامون آمادس تا به مدل داده بشه و train بشه.

```
from sklearn.linear_model import LinearRegression  
  
lin_reg = make_pipeline(preprocessing, LinearRegression())  
lin_reg.fit(housing, housing_labels)
```

- تمام، الان یه مدل بر پایه linear regression داریم. اول یه پایپلاین درست کردیم و بخش آخرش یه مدل رگرسیون خطی گذاشتیم تا در انتها پیش‌بینی انجام بده. بیایم نتایج پیش‌بینی با مقادیر واقعی رو یه بررسی کنیم:

```
>>> housing_predictions = lin_reg.predict(housing)  
>>> housing_predictions[:5].round(-2) # -2 = rounded to the nearest hundred  
array([243700., 372400., 128800., 94400., 328300.])  
>>> housing_labels.iloc[:5].values  
array([458300., 483800., 101700., 96100., 361800.])
```

- مدلی هستش که پرفورمنس خیلی بدی نداره ولی خیلی خوبم نیست چون برای نمونه اول یه اختلاف ۲۰۰ هزارتایی بین مقدار واقعی و چیزی که پیش‌بینی کرده وجود داره. میتوانیم به طور میانیگن حساب کنیم چه مقدار اختلاف وجود داره تو کل دیتاهامون.

- حدود ۶۸ هزار دلار اختلاف پیش‌بینی و عدد واقعی هست! خیلی خوب نیست.

```
>>> from sklearn.metrics import mean_squared_error  
>>> lin_rmse = mean_squared_error(housing_labels, housing_predictions,  
...                                squared=False)  
...  
>>> lin_rmse  
68687.89176589991
```

CONTINUE

- این یه مثال واضح از اینکه underfit رخ بده. چون مدل حتی روی training set هم پرفورمنس خوبی نداشته! از قبل میدونم یه راه برای حل این مشکل اینه یه مدل قدرتمند تر انتخاب کنیم. پس با decision tree این کارو پیش میبریم.

```
from sklearn.tree import DecisionTreeRegressor

tree_reg = make_pipeline(preprocessing, DecisionTreeRegressor(random_state=42))
tree_reg.fit(housing, housing_labels)

Now that the model is trained, you evaluate it on the training set:

>>> housing_predictions = tree_reg.predict(housing)
>>> tree_rmse = mean_squared_error(housing_labels, housing_predictions,
...                                     squared=False)
...
>>> tree_rmse
0.0
```

همینطور که میبینیم به ارور \circ رسیدیم. خب این نشون نمیده که مدلمون خیلی خوبه، بلکه نشون میده که overfitting رخ میده. پس راه کار چیه؟ راه کار اینه که یه بخش دیتا رو برای train و یه بخش رو برای validation کنار بذاریم.

- مفهومی که گفتیم بهش میگن cross-validation به این شکل که ما میاییم از کل دیتامون مثلا به ۱۰ بخش مساوی تقسیمش میکنیم و یه مدل رو یه بار با ۹ بخش train و با یه بخش که باقی مونده و تا حالا ندیدتش اون رو ارزیابی میکنیم:

```
from sklearn.model_selection import cross_val_score

tree_rmses = -cross_val_score(tree_reg, housing, housing_labels,
                             scoring="neg_root_mean_squared_error", cv=10)
```



Scikit-Learn's cross-validation features expect a utility function (greater is better) rather than a cost function (lower is better), so the scoring function is actually the opposite of the RMSE. It's a negative value, so you need to switch the sign of the output to get the RMSE scores.

CONTINUE

- حالا که میتوانیم ۱۰ تا خروجی متفاوت از درخت داشته باشیم، هر سری درخت یه پیش بینی انجام میده

```
>>> pd.Series(tree_rmses).describe()
count    10.000000
mean    66868.027288
std     2060.966425
min     63649.536493
25%    65338.078316
50%    66801.953094
75%    68229.934454
max     70094.778246
dtype: float64
```

حالا میتوانیم ببینیم هرکدام چطوری عمل کردن، نکته ای که وجود داره به طور متوسط ۶۶هزار دلار تفاوت داره پس خیلی تفاوت خاصی با رگرسیون خطی نداشته این مدل.

در واقع چیزی که داخل cross-validation اتفاق میفته اینه که بیایم یه مدل رو k بار بر اساس بخش های مختلف دیتاست evaluate کنیم و ببینیم به طور متوسط چی کار کرده و کدام حالت بهترین مدل رو داشته تا از روی اون یه سری پارامتر برداریم حالا میتوانیم یه درخت جدید رو بر اساس پارامتر هایی که مدل قبلی داشته train کنیم.

How Cross-Validation Works

- Data Splitting:** The dataset is divided into multiple subsets or "folds." A common method is k-fold cross-validation, where the data is split into k equally sized folds.
- Training and Validation:** The model is trained on $k - 1$ folds and validated on the remaining fold. This process is repeated k times, with each fold being used as the validation set exactly once.
- Performance Averaging:** The performance metric (such as accuracy, precision, recall, etc.) is calculated for each iteration, and the results are averaged to provide an overall performance estimate.
- Final Model Selection:** The model that will be used as your final decision tree is typically trained on the entire dataset, using the same hyperparameters and configurations that performed best during cross-validation.

CONTINUE

- بیایم یه مدل دیگه به اسم random forest انتخاب کنیم. کاری که این مدل انجام میده اینه که کلی درخت بر اساس یه سری فیچر زندوم train میکنه و در نهایت جواب خروجی من میشه میانگین جواب خروجی همه مدل ها. به این کار که بیایم کلی مدل train کنیم و جواب رو ازشون میانگین بگیریم ensemble میگن.

```
from sklearn.ensemble import RandomForestRegressor

forest_reg = make_pipeline(preprocessing,
                           RandomForestRegressor(random_state=42))
forest_rmses = -cross_val_score(forest_reg, housing, housing_labels,
                                scoring="neg_root_mean_squared_error", cv=10)
```

Let's look at the scores:

```
>>> pd.Series(forest_rmses).describe()
count    10.000000
mean     47019.561281
std      1033.957120
min      45458.112527
25%     46464.031184
50%     46967.596354
75%     47325.694987
max     49243.765795
dtype: float64
```

و بلاخره مدلمنون یه نتیجه بهتر از مدل های قبلی ارائه داد. اگه بازم نیاز داشتیم پرفورمنس بهتری بگیریم احتمالا باید مدل های دیگه رو هم امتحان کنیم.

FINE-TUNE YOUR MODEL

- یه جاهایی ممکنه ندونی که یه مدل رو بر اساس چه hypuerparameter گاما رو چند بذاری؟ یک دهم؟ یک صدم. به پارامتر هایی که خودمون مشخص میکنیم تو فرایند training چند باشن hyperparameter میگیم.
- ۱. grid search: یه روش اینه عدد های مختلف رو امتحان کنیم و ببینیم کدام مدل بهتر بود ولی روش خوبی نیست واقعا چون کلی وقت میبره ازمون. راه دیگه اینه از gridsearchcv استفاده کنیم.

```
from sklearn.model_selection import GridSearchCV

full_pipeline = Pipeline([
    ("preprocessing", preprocessing),
    ("random_forest", RandomForestRegressor(random_state=42)),
])
param_grid = [
    {'preprocessing__geo_n_clusters': [5, 8, 10],
     'random_forest__max_features': [4, 6, 8]},
    {'preprocessing__geo_n_clusters': [10, 15],
     'random_forest__max_features': [6, 8, 10]},
]
grid_search = GridSearchCV(full_pipeline, param_grid, cv=3,
                           scoring='neg_root_mean_squared_error')
grid_search.fit(housing, housing_labels)
```

Notice that you can refer to any hyperparameter of any estimator in a pipeline, even if this estimator is nested deep inside several pipelines and column transformers. For example, when Scikit-Learn sees "preprocessing__geo_n_clusters", it splits this string at the double underscores, then it looks for an estimator named "preprocessing" in the pipeline and finds the preprocessing ColumnTransformer. Next, it looks for a transformer named "geo" inside this ColumnTransformer and finds the ClusterSimilarity transformer we used on the latitude and longitude attributes. Then it finds this transformer's n_clusters hyperparameter. Similarly, random_forest__max_features refers to the max_features hyperparameter of the estimator named "random_forest", which is of course the RandomForest model (the

کاری که انجام میده اینه که میاد حالت های مختلف ترکیب hyperparameter رو بررسی میکنه و بهترین ترکیب رو بهمون میگه کدام حالت بوده.

There are two dictionaries in this param_grid, so GridSearchCV will first evaluate all $3 \times 3 = 9$ combinations of n_clusters and max_features hyperparameter values specified in the first dict, then it will try all $2 \times 3 = 6$ combinations of hyperparameter values in the second dict. So in total the grid search will explore $9 + 6 = 15$

combinations of hyperparameter values, and it will train the pipeline 3 times per combination, since we are using 3-fold cross validation. This means there will be a grand total of $15 \times 3 = 45$ rounds of training! It may take a while, but when it is done you can get the best combination of parameters like this:

```
>>> grid_search.best_params_
{'preprocessing__geo_n_clusters': 15, 'random_forest__max_features': 6}
```

In this example, the best model is obtained by setting n_clusters to 15 and setting max_features to 8.

CONTINUE

- ۲. randomized search: اگه تعداد فیچر و تعداد hyperparameter ها زیاد بشه بدی روش قبلی اینه که کلی وقت میبره تا بیاد همه جایگشت های مختلف رو امتحان کنه پس راهکار چیه؟ به طور رندوم یه سری از ترکیب هارو برداریم. خوبیای خودشم داره مثلًا اگه هیچ ایده ای بابت یه hyperparameter نداشته باشیم یا اگه مقادیرش بتونه یه رنج وسیعی باشه سخت میشه با حالت قبلی همه حالات رو پوشش بدیم. به این شکل هم پیاده سازی میشه کرد:

```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint

param_distributions = {'preprocessing__geo_n_clusters': randint(low=3, high=50),
                      'random_forest__max_features': randint(low=2, high=20)}

rnd_search = RandomizedSearchCV(
    full_pipeline, param_distributions=param_distributions, n_iter=10, cv=3,
    scoring='neg_root_mean_squared_error', random_state=42)

rnd_search.fit(housing, housing_labels)
```

ANALYZING THE BEST MODEL

- حالا که یه مدل خوب داریم میتونیم بفهمیم کدام فیچر ها تاثیر گذار بودن تو ساخت مدلمون:

```
>>> final_model = rnd_search.best_estimator_ # includes preprocessing
>>> feature_importances = final_model["random_forest"].feature_importances_
>>> feature_importances.round(2)
array([0.07, 0.05, 0.05, 0.01, 0.01, 0.01, 0.01, 0.19, [...], 0.01])
```

Let's sort these importance scores in descending order and display them next to their corresponding attribute names:

```
>>> sorted(zip(feature_importances,
...             final_model["preprocessing"].get_feature_names_out(),
...             reverse=True)
...
[(0.18694559869103852, 'log__median_income'),
 (0.0748194905715524, 'cat__ocean_proximity_INLAND'),
 (0.06926417748515576, 'bedrooms__ratio'),
 (0.05446998753775219, 'rooms_per_house__ratio'),
 (0.05262301809680712, 'people_per_house__ratio'),
 (0.03819415873915732, 'geo__Cluster 0 similarity'),
 [...]
 (0.00015061247730531558, 'cat__ocean_proximity_NEAR BAY'),
 (7.301686597099842e-05, 'cat__ocean_proximity_ISLAND')]
```

حالا میتونیم یه سری فیچر که بدردمون نمیخورن رو پاک کنیم. البته به جز روش دستی خود scikit-learn با استفاده از selectfrommodel میتونه این کارو برامون انجام بده.

بعد از این کار نوبت این هستش که مطمئن بشیم مدلمون توانایی اینو داره روی کنگوری های مختلفی از دیتا پیش‌بینی های خوبی انجام بده. بعد از اینکه مطمئن شدیم از این بابت بعدش میتوانیم از صحت مدلمون مطمئن بشیم.

EVALUATE YOUR MODEL ON TEST SET

- در نهایت باید از میزان دقیقیت مدلمان را test set اطلاع پیدا کنیم. این کار را به شکل زیر میتوانیم انجام بدیم:

```
X_test = strat_test_set.drop("median_house_value", axis=1)
y_test = strat_test_set["median_house_value"].copy()

final_predictions = final_model.predict(X_test)

final_rmse = mean_squared_error(y_test, final_predictions, squared=False)
print(final_rmse) # prints 41424.40026462184
```

- در نهایت فقط باید این ریز نکته را بررسی کنیم که عدد ارور را بدست اوردیم. اگه این عدد خیلی بزرگ تر از ارور های validation بود باید متوجه بشیم چون داخل فرایند انتخاب کردن hyperparameter کلی cross-validation زدیم مدلمان به این سمت رفته که validation set را کم کنه و خیلی به هدف اصلیمون که مدل خوبی باشه برای اینکه همه دیتاهایی که ندیده رو به طور خوبی پیش‌بینی کنه پیش نرفتیم. اگه همچین مشکلی پیش بیاد باید با دوباره دنبال پیداکردن hyperparameter های بهتری برای مدل باشیم.