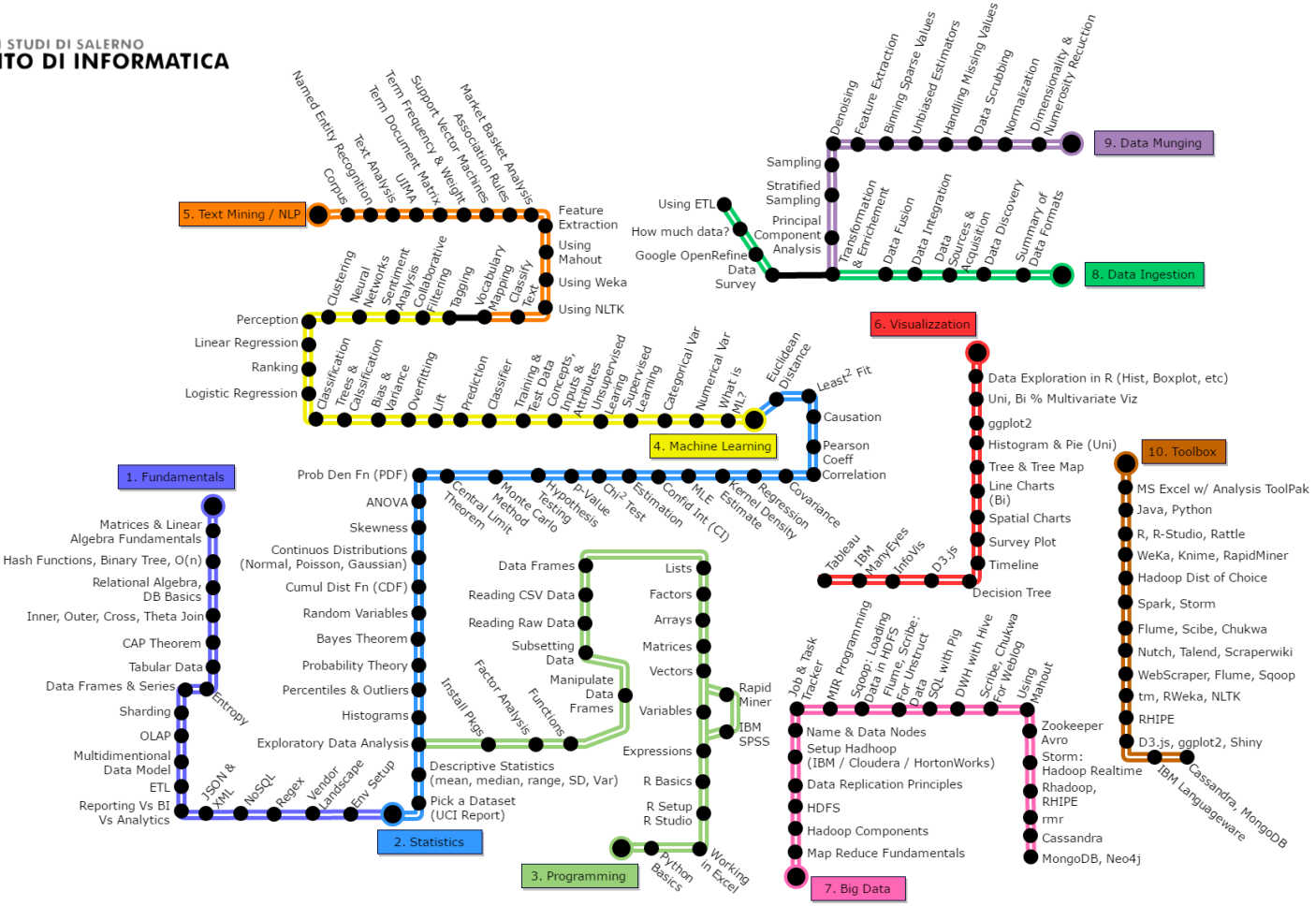
UNIVERSITÀ DEGLI STUDI DI SALERNO
DIPARTIMENTO DI INFORMATICA



Data Science e Machine Learning

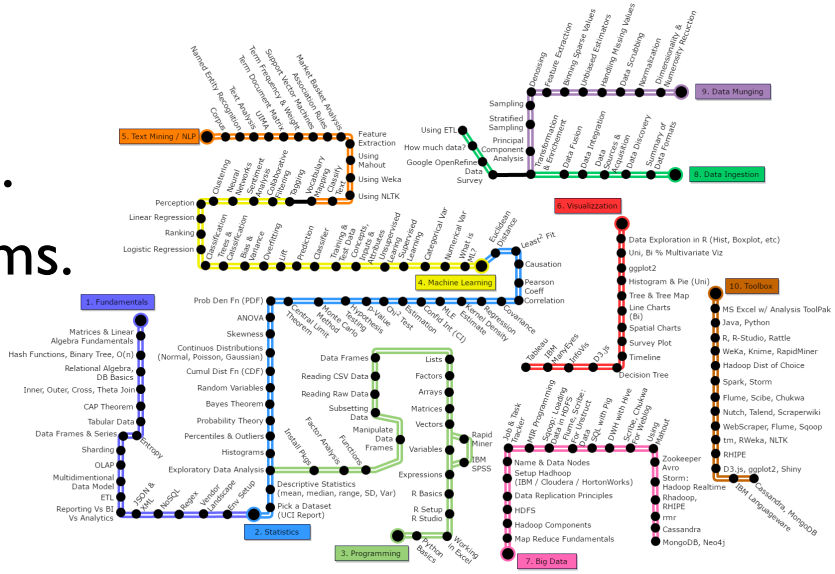
End-to-End-ML-Project

Aurelien Geron: «Hands on Machine Learning with Scikit Learn and TensorFlow, O'Reilly ed.

Prof. Giuseppe Polese, aa 2024-25

Overview

1. **Main steps of a ML Project**
2. Look at the big picture.
3. Get the data.
4. Visualize the data to gain insights.
5. Prepare the data for ML algorithms.
6. Select a model and train it.
7. Fine-tune the model.
8. Present the solution.
9. Launch, monitor, and maintain the system.



End-to-End ML Project

- An end-to-end example project
- The main steps:
  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 the model;
  7. Present the solution;
  8. Launch, monitor, and maintain the system.

Working with Real Data

- When learning about ML it is best to experiment with real-world data, not just artificial datasets.
  - there are thousands of open datasets to choose from
- *Repositories*:
  - UC Irvine Machine Learning Repository
  - Kaggle datasets
  - Amazon's AWS datasets
- *Portals*:
  - *http://dataportals.org/*
  - *http://opendatamonitor.eu/*
  - *http://quandl.com/*

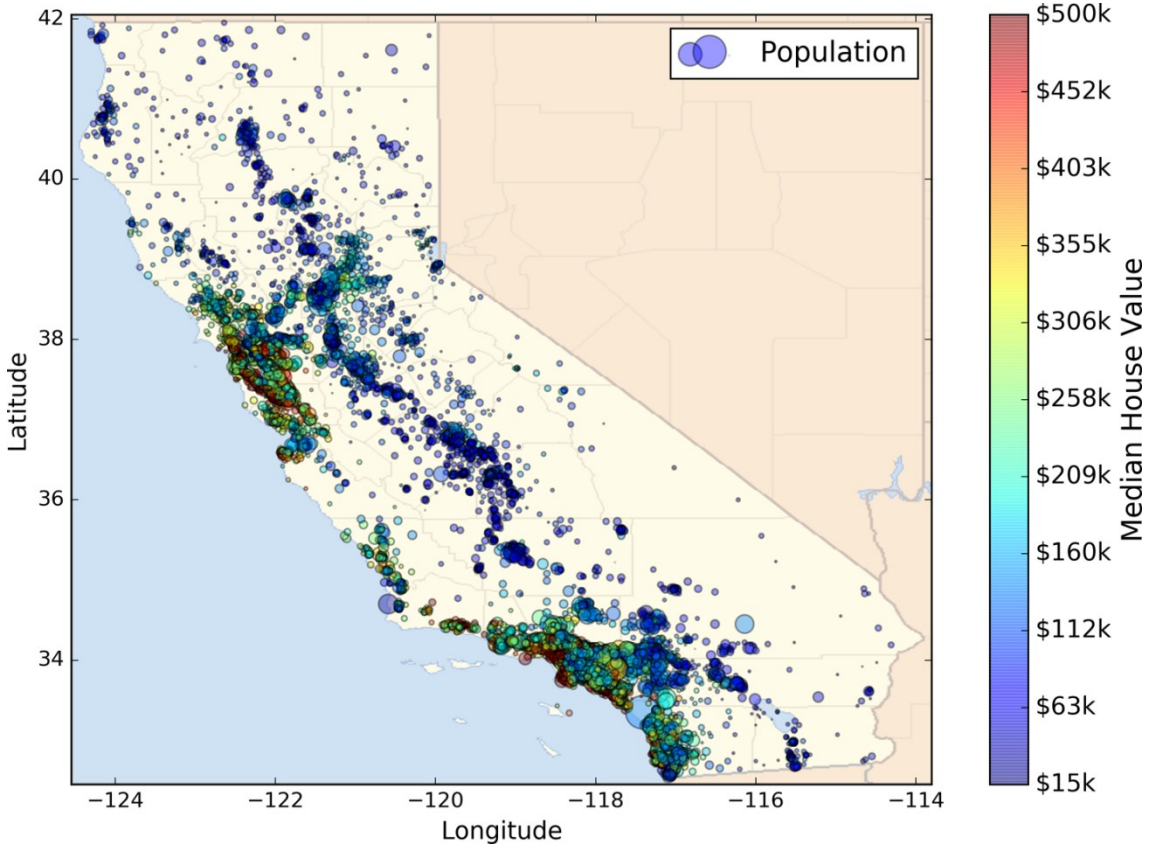
Other open data repositories:

- Wikipedia's list of Machine Learning datasets
- Quora.com question
- Datasets subreddit

The used Dataset

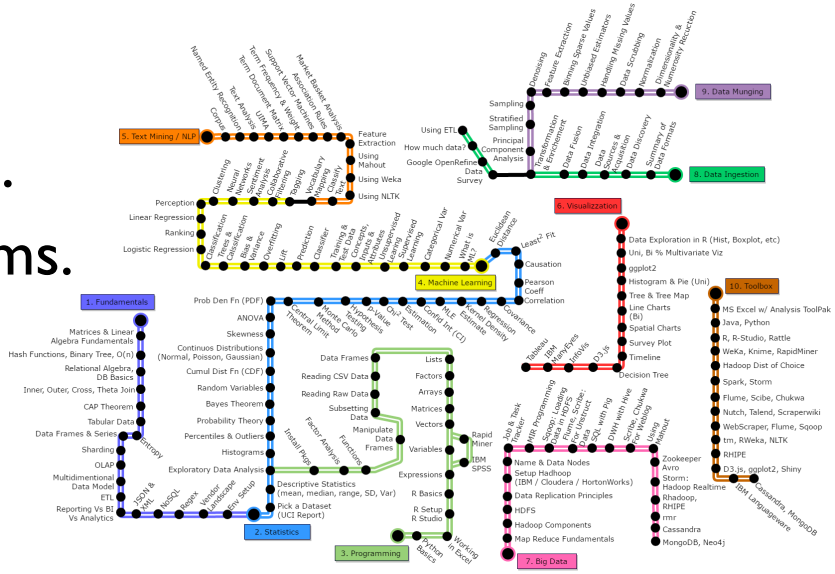
- We will use the California Housing Prices dataset from the StatLib repository

It was based on data from the 1990 California census.



Overview

1. Main steps of a ML Project
2. **Look at the big picture.**
3. Get the data.
4. Visualize the data to gain insights.
5. Prepare the data for ML algorithms.
6. Select a model and train it.
7. Fine-tune the model.
8. Present the solution.
9. Launch, monitor, and maintain the system.



Look at the Big Picture

build a model of housing prices in California using the California census data.

- This data has metrics such as:
  - The population,
  - Median income,
  - Median house pricing,
  - …
- For each district in California (a district typically has from 600 to 3,000 people).

Look at the Big Picture (2)

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

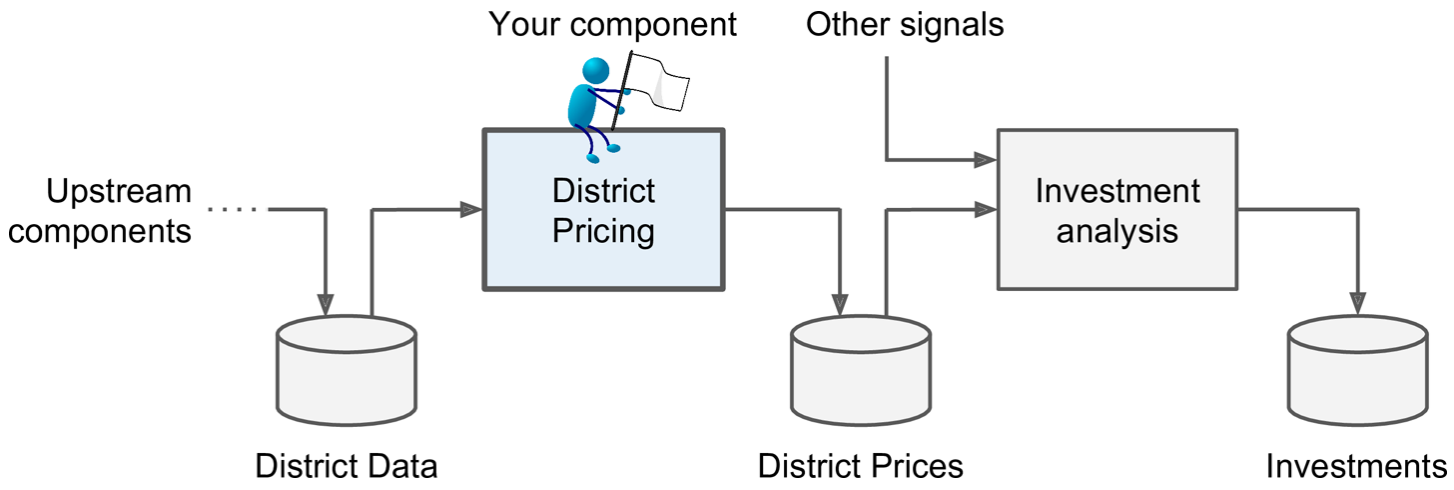
Start Framing the Problem

How does the company expect to use and Benefit from this model?

- This is important because it will determine:
  - how we frame the problem,
  - what algorithms we will select,
  - what performance measure we will use to evaluate our model, and
  - how much effort we should spend tweaking it.

Pipeline systems

- Suppose the answer is that our model's output
  - a prediction of a district's median housing price
- will be fed to another Machine Learning system, along with many other signals



Ask Current Solution

- The next question to ask is what the current solution looks like (if any).
- It will often give us a reference performance, as well as insights on how to solve the problem.
- Suppose the answer is that the district housing prices are currently estimated manually by experts:
  - a team gathers up-to-date information about a district, and if the median housing price is not available, they estimate it using complex rules.
- This is costly and time-consuming.

ML Based Solution

- Suppose manual estimates are wrong by more than 10%.
- This makes us think that it would be useful to train a model to predict a district's median housing price given predictive data about a district.
- The census data looks great for this purpose, since it includes the median housing prices of thousands of districts, as well as other data.

with all this information we are now ready to start designing our system

Frame The Problem

- We are now ready to start designing our system.
- First, we need to frame the problem: is it supervised, unsupervised, or Reinforcement Learning?
- Is it a classification task, a regression task, or something else?
- Should we use batch or online learning techniques?

Frame The Problem (2)

- It is a typical supervised learning task, since we are given labeled training examples.
- It is a regression task, since we are to predict a value.
- More specifically, it is a multivariate regression problem, since we will use multiple features to make a prediction, whereas the prediction of life satisfaction was a univariate regression problem, since it was based on one feature, the GDP per capita.
- There is no continuous flow of data, no need to adjust to changing data rapidly, and the data is small enough to fit in memory, so plain batch learning should do fine.

Huge Data

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

Select a Performance Measure

- Our 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.

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

  - Where *m* is the number of instances.

Alternative Performance Measure

- Even though the RMSE is the preferred performance measure for regression tasks, in some contexts we may prefer to use another function.
- If there are many outlier districts, we may consider the Mean Absolute Error (a.k.a. Average Absolute Deviation), since it is less sensitive to outliers:
  - It gives an idea of how much error the system makes in its predictions, with a higher weight for large errors

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

  - Both the RMSE and the MAE measure the distance between the vectors of predictions and target values.

Other Distance Measures

- RMSE corresponds to the Euclidean Distance, also called $\ell 2$ norm, noted $\| \cdot \|_2$ (or just $\| \cdot \|$).
- MAE corresponds to the ℓ_1 norm (Manhattan norm).
- The ℓ_k norm is the Minkowsky norm, whereas ℓ_0 gives the number of non-zero elements in a vector, and ℓ_∞ 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 RMSE is more sensitive to outliers than MAE.
- If outliers are rare (like in a bell-shaped curve), the RMSE is generally preferred.

Check the Assumptions

It is good practice to list and verify the assumptions that were made so far

- Example: the district prices that our system outputs are going to be fed into a downstream Machine Learning system.
  - we assume that these prices are going to be used as such

what if the downstream system actually converts the prices into categories and then uses those categories instead of the prices themselves?

Classification or Regression?

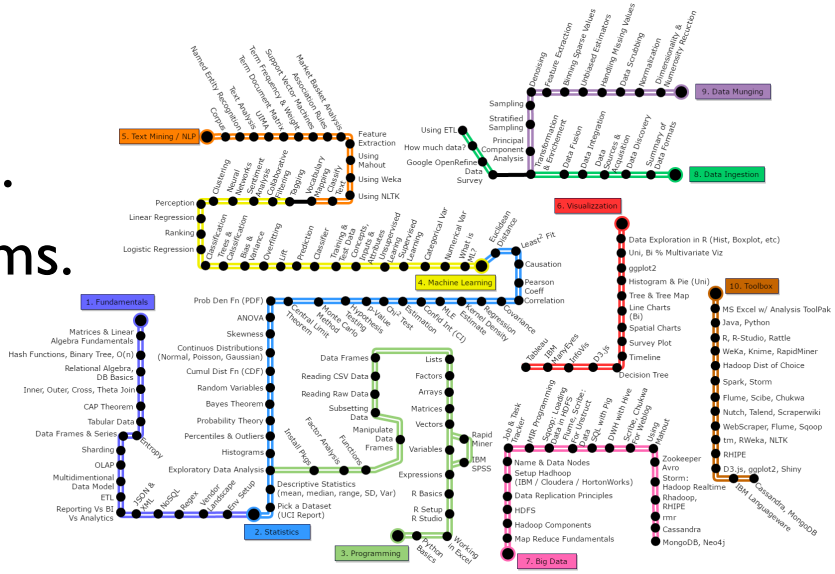
- In this example, rather than the perfect price, the system just needs to get the right category.
- Maybe the problem should have been framed as a classification rather than a regression task.
- We don't want to find this out after working on a regression system for months.
- But suppose that the team in charge of the downstream system indeed wants the actual prices.
- We're set, and we can start coding now!

Workbench Configuration

- To run code examples we can download a Jupyter notebook https://github.com/ageron/handson-ml.
- Install Python (version 3) from https://www.python.org/
- Install the following Python modules: Jupyter, NumPy, Pandas, Matplotlib, and Scikit-Learn.
- They can be installed through a packaging system (e.g., apt-get on Ubuntu, or MacPorts or HomeBrew on macOS), such as the one of the Scientific Python distribution *Anaconda*, or the Python's packaging system, pip, included with the Python binary installers.

Overview

1. Main steps of a ML Project
2. Look at the big picture.
3. **Get the data.**
4. Visualize the data to gain insights.
5. Prepare the data for ML algorithms.
6. Select a model and train it.
7. Fine-tune the model.
8. Present the solution.
9. Launch, monitor, and maintain the system.



Download the Data

- Data for this project can be downloaded from the file *housing.tgz*, which contains a CSV file housing.csv.
- To fetch latest updates, it might be useful to automate data download through the following function:

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

DOWNLOAD_ROOT = "https://raw.githubusercontent.com/ageron/handson-ml/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()
```

Load the Data

- fetch_housing_data() creates a datasets/housing directory in the workspace, downloads housing.tgz, and extracts housing.csv from it.
- Next, we can use the Pandas library to load the data, through the following function:

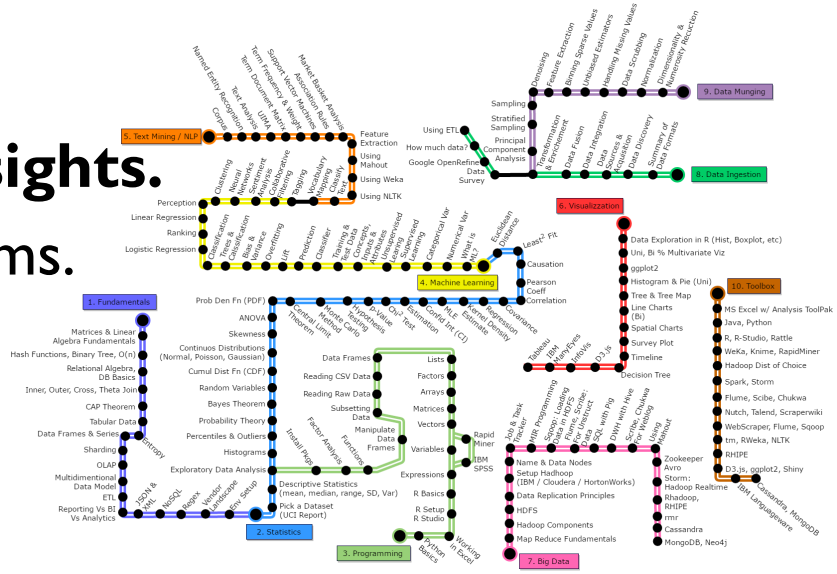
```
import pandas as pd

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

  - It returns a Pandas DataFrame object containing all the data.

Overview

1. Main steps of a ML Project
2. Look at the big picture.
3. Get the data.
4. **Visualize the data to gain insights.**
5. Prepare the data for ML algorithms.
6. Select a model and train it.
7. Fine-tune the model.
8. Present the solution.
9. Launch, monitor, and maintain the system.



Look at the Data

- Look at the top 5 rows using the DataFrame's head() method:

```
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

- Each row represents a district. Beyond those in the screenshot other attributes are *households, median_income, median_house_value, and ocean_proximity.*

Quick Profile Data

- The info() method is useful to quickly profile data, such as number of rows and non null values.

```
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
```

- 20,640 instances are few for ML, but good to start.
- Notice that 207 districts are missing the attribute total_bed_rooms, we need to monitor this later.

Get insights from Data

- Attributes are numerical, except ocean_proximity, but having loaded it from a CSV we know it is a textual.
- If looking at few rows we noticed repetitive values for this column, maybe it is a categorical attribute.
- We can find the categories and how many districts belong to them through the value_counts() method:

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

Summary of Numeric Attributes

- The describe() method shows a summary of numerical attributes.

```
housing.describe()
```

	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(

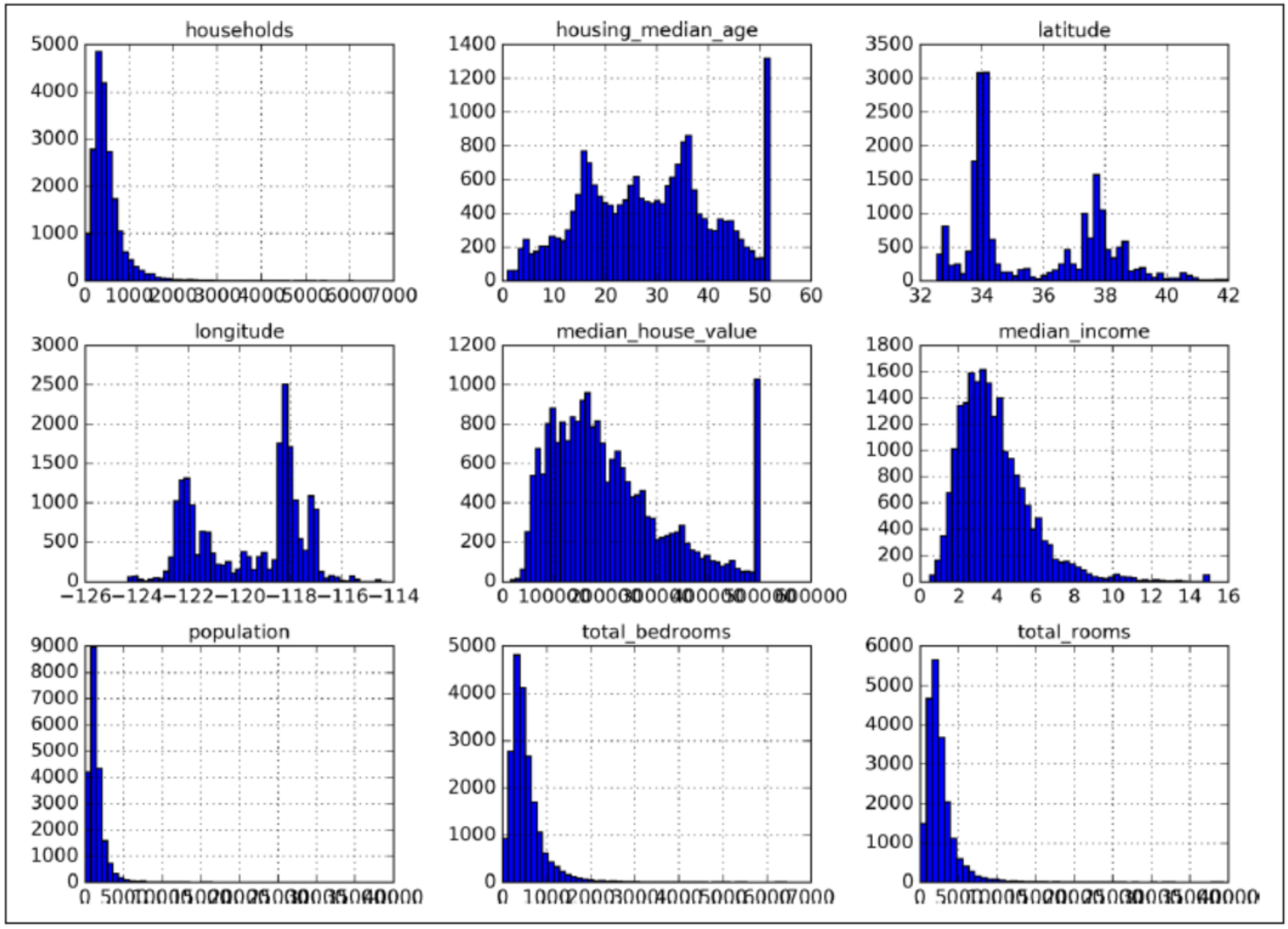
- Null values are ignored (total_bedrooms is 20,433, not 20,640).
- Std is standard deviation, measuring how dispersed the values are.
- 25%, 50%, and 75% rows show percentiles: value below which a percentage of observations falls. For example, 25% of districts have housing_median_age lower than 18, and so on (also called 1st quartile, median, and 3rd quartile).

Visualize data through Histograms

- A quick way to inspect data is to plot a histogram for each numerical attribute.
- We can either plot this one attribute at a time, or for the whole dataset by calling the hist() method:

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

Histograms for this dataset



Insights from Histograms

1. median income does not seem to be expressed in US $. Suppose stakeholders tell us that data has been scaled and capped at 15 (actually 15.0001) for higher median incomes, and at 0.5 (actually 0.4999) for lower median incomes.
2. housing median age and the median house value were also capped. This may be a serious problem since it is our dependent attribute. Our ML algorithms may learn that prices never go beyond that limit. We need to check with users to see if this is a problem. If they need precise predictions even beyond $500,000, then we have two options:
   - Collect proper labels for the districts whose labels were capped.
   - Remove those districts from the training (and from the test) set, since the system should not be evaluated poorly if it predicts values beyond $500,000).

Insights from Histograms (2)

3. Attributes have different scales. (We will discuss this later with feature scaling).
4. Many histograms are tail heavy: they extend much farther to the right of the median than to the left. This may make it a bit harder for some ML algorithms to detect patterns. We will try transforming these attributes later to have more bell-shaped distributions.
5. Hopefully we now have a better understanding of the kind of data we are dealing with.

Test Set

- Before looking at the data further, we need to create a test set, put it aside, and never look at it.
- Because our brain is highly prone to overfitting: if we look at the test set, we may detect some seemingly interesting pattern in the data that leads us to select a particular Machine Learning model.
- When estimating the generalization error on the test set, the estimate might be too optimistic and we will launch a system not performing as expected.
- This is called data snooping bias.

Create a Test Set

- To create a test set we randomly pick some instances, typically 20% of the dataset, and set them aside

```
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]
```

- And invoke it in the following way:

```
>>> train_set, test_set = split_train_test(housing, 0.2)
>>> print(len(train_set), "train +", len(test_set), "test")
16512 train + 4128 test
```

Generating the same Test Set

- This works, but if we run the program again, it will generate a different test set!
- Over time, we (or the ML algorithms) will see the whole dataset, which is what we should avoid.
- One solution is to save the test set on the first run and then load it in subsequent runs.
- Another option is to set the random number generator's seed (e.g., np.random.seed(42)) before calling np.random.permutation(), so that it always generates the same shuffled indices.

Problems with Dataset Updates

- Both previous two solutions will break when fetching an updated dataset.
- We could compute a hash of each instance's identifier, keeping only the last byte, and put the instance in the test set if it is lower or equal to 51 (~20% of 256).
- The test set will be consistent across multiple runs, even if refreshing the dataset.
- The new test set will contain 20% of the new instances, but no instance that was previously in the training set.

Python Implementation

- A possible Python implementation is:

```
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]
```

- Apply adds a column to a dataframe
- Unfortunately, the housing dataset does not have an identifier column. We can use the row index as ID:

```
housing_with_id = housing.reset_index()   # adds an `index` column
train_set, test_set = split_train_test_by_id(housing_with_id, 0.2, "index")
```

Problems with Row Index ID

- When using the row index as ID, we need to make sure that new data gets appended to the end of the dataset, and no row is ever deleted.
- If this is not possible, we need to use more a stable feature as an ID.
- For example, a district's latitude and longitude are guaranteed to be stable for a few million years, so we could combine them into an ID :

```
housing_with_id["id"] = housing["longitude"] * 1000 + housing["latitude"]
train_set, test_set = split_train_test_by_id(housing_with_id, 0.2, "id")
```

Additional Splitting Functions

- Scikit-Learn provides other functions to split datasets into multiple subsets in various ways.
- The simplest function is train_test_split, similar to split_train_test seen earlier, with some more features:
  - A random_state parameter to set the random generator seed
  - Possibility to can pass it multiple datasets with same number of rows, and it will split them on the same indices (useful when having a separate DataFrame for labels):

```
from sklearn.model_selection import train_test_split

train_set, test_set = train_test_split(housing, test_size=0.2, random_state=42)
```

Sampling Bias

- So far we have considered purely random sampling methods. This is fine if the dataset is large enough.
- But if it is not, there is the risk of introducing a significant sampling bias:
  - When a survey company decides to call 1,000 people to ask them a few questions, they don't just pick 1,000 people randomly in a phone book. They try to ensure that these 1,000 people are representative of the whole population.

Stratified Sampling

- The US population is composed of 51.3% female and 48.7% male, so a survey in US should try to maintain this ratio in the sample: 513 female and 487 male.
- This is called stratified sampling: the population is divided into homogeneous subgroups called strata.
- If using purely random sampling, there would be about 12% chance of sampling a skewed test set with either less than 49% female or more than 54% female.
- Either way, the survey results would be significantly biased.

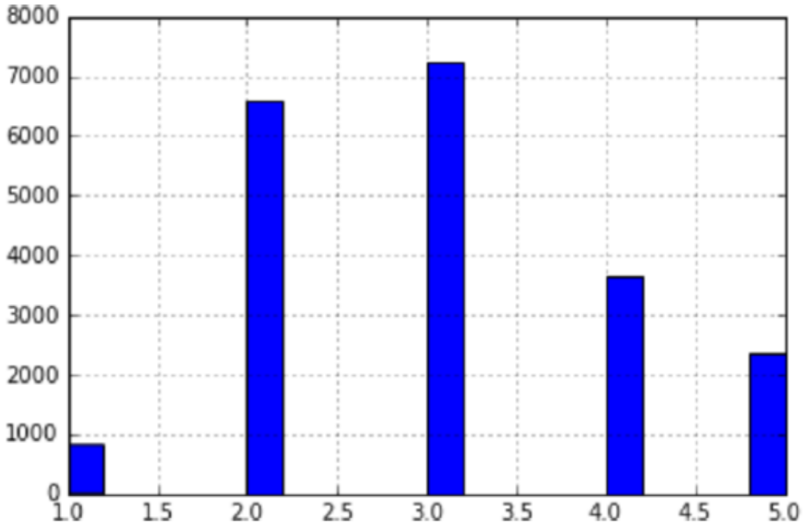
Bias in the Housing Dataset

- Suppose an expert told us that the median income is a very important attribute to predict median housing prices.
- We may want to ensure that the test set is representative of the various categories of incomes in the whole dataset.
- Since it is a continuous numerical attribute, we need to create an income category attribute.
- The median income histogram tells that most values cluster around $20,000–$50,000, but some go far beyond $60,000.
- It is important to have a sufficient number of instances in the dataset for each stratum, or the estimate of the stratum's importance may be biased. This means that we should not have too many strata, and each stratum should be large enough.

Avoid Bias in the Housing Dataset

- The following code creates an income category attribute by dividing the median income by 1.5 (to limit the number of categories), and rounding up using ceil (to have discrete categories), and merging those greater than 5 into category 5:

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



Stratified Sampling for Housing

- We can do stratified sampling based on the income category. We can use Scikit-Learn's StratifiedShuffleSplit class:

```
from sklearn.model_selection import StratifiedShuffleSplit
split = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=42)
for train_index, test_index in split.split(housing, housing["income_cat"]):
    strat_train_set = housing.loc[train_index]
    strat_test_set = housing.loc[test_index]
```

- Let's see if it works as expected. We can start looking at the income category proportions in the test set:

```
>>> strat_test_set["income_cat"].value_counts() / len(strat_test_set)
3.0 0.350533
2.0 0.318798
4.0 0.176357
5.0 0.114583
1.0 0.039729
Name: income_cat, dtype: float64
```

Proportions in the Datasets

- Table below compares the income category proportions in the whole dataset, in the test set generated with purely random sampling, and the one generated with stratified sampling.
- The one generated with stratified sampling has income category proportions almost identical to those in the full dataset, whereas the other one is quite skewed.

	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

Removing Attributes

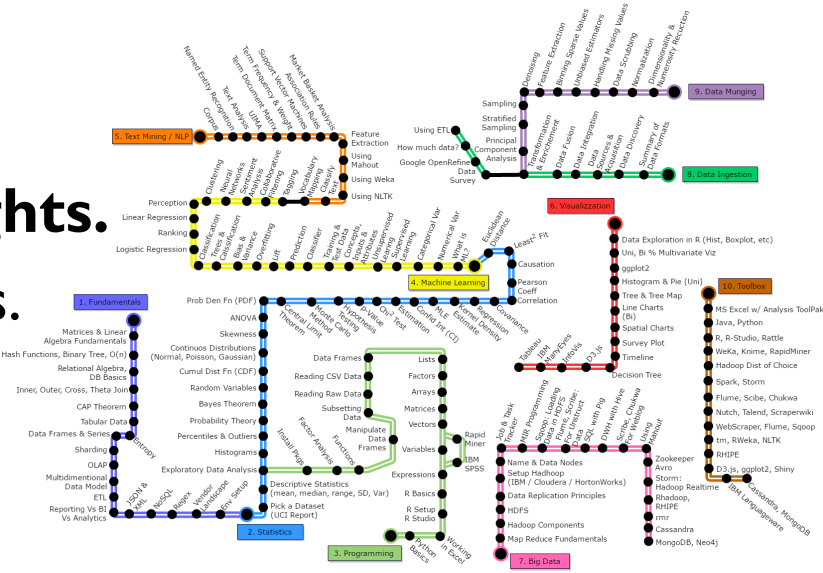
- Now we can remove the income_cat attribute so the data is back to its original state:

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

- Devoting enough time to test set generation is an often neglected but critical part of a ML project.
- Many of these ideas will be useful later when we discuss cross-validation.

Overview

1. Main steps of a ML Project
2. Look at the big picture.
3. Get the data.
4. **Visualize the data to gain insights.**
5. Prepare the data for ML algorithms.
6. Select a model and train it.
7. Fine-tune the model.
8. Present the solution.
9. Launch, monitor, and maintain the system.



Exploring the Training Set

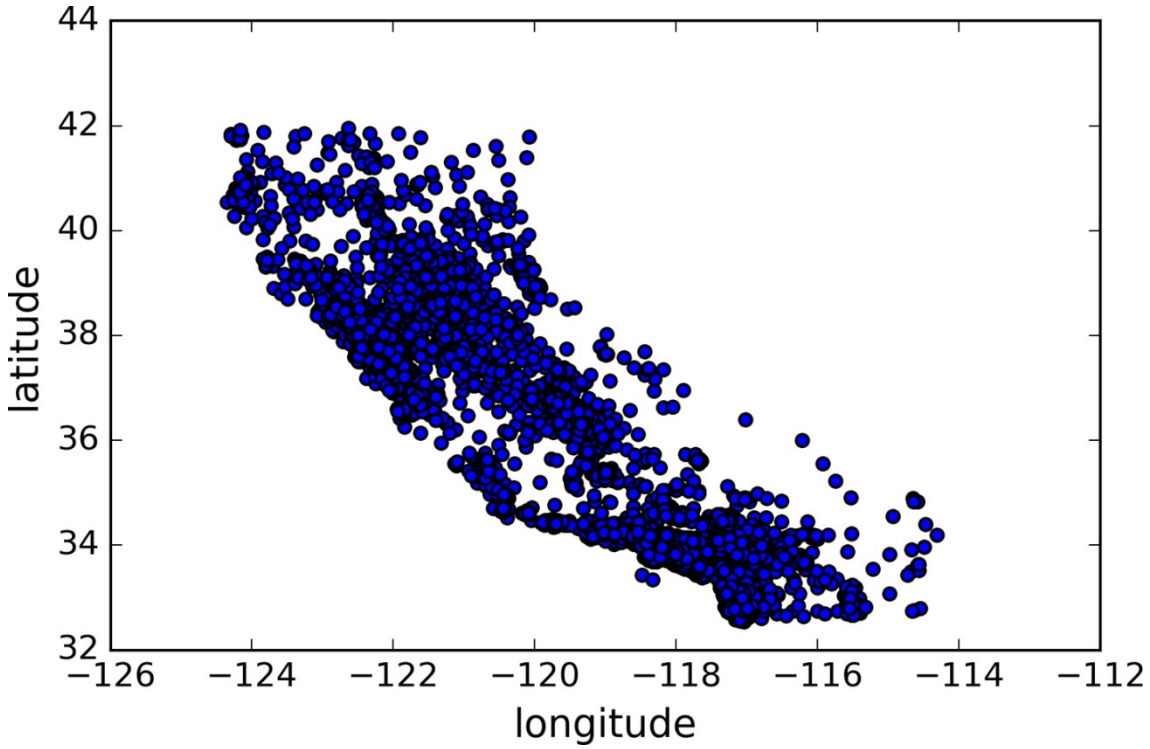
- So far we have taken a quick glance at the data to get a general understanding of them.
- Now the goal is to go a little bit more in depth.
- First, we should put the test set aside and esplore only the training set.
- If the training set is large, we may sample an exploration set, to make manipulations easy and fast.
- Since our training set is quite small we work on its entirety, but it is safer to create a copy of it:

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

Visualizing Geographic Data

- Since there is geographical information, it is a good idea to create a scatterplot to visualize training data:

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

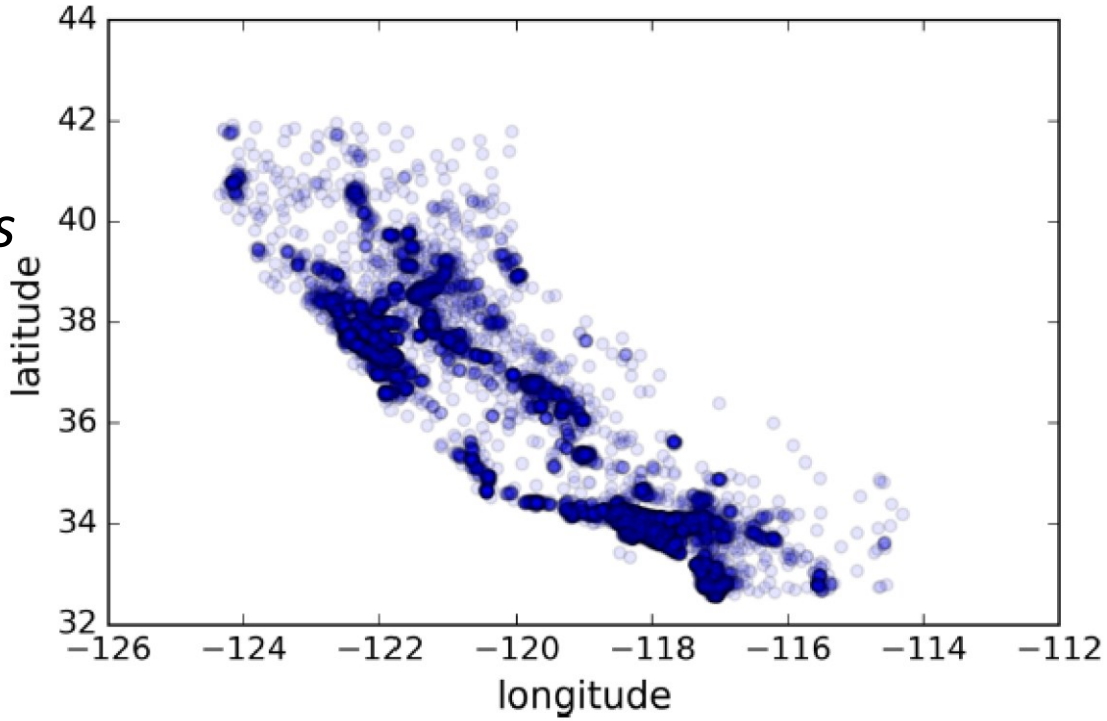


Highlighting high-density areas

- Setting the alpha option to 0.1 highlights places with a high density of data points:

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

Now we can see the high-density areas (the Bay Area and around Los Angeles and San Diego), plus a long line of fairly high density in the Central Valley (around Sacramento and Fresno)



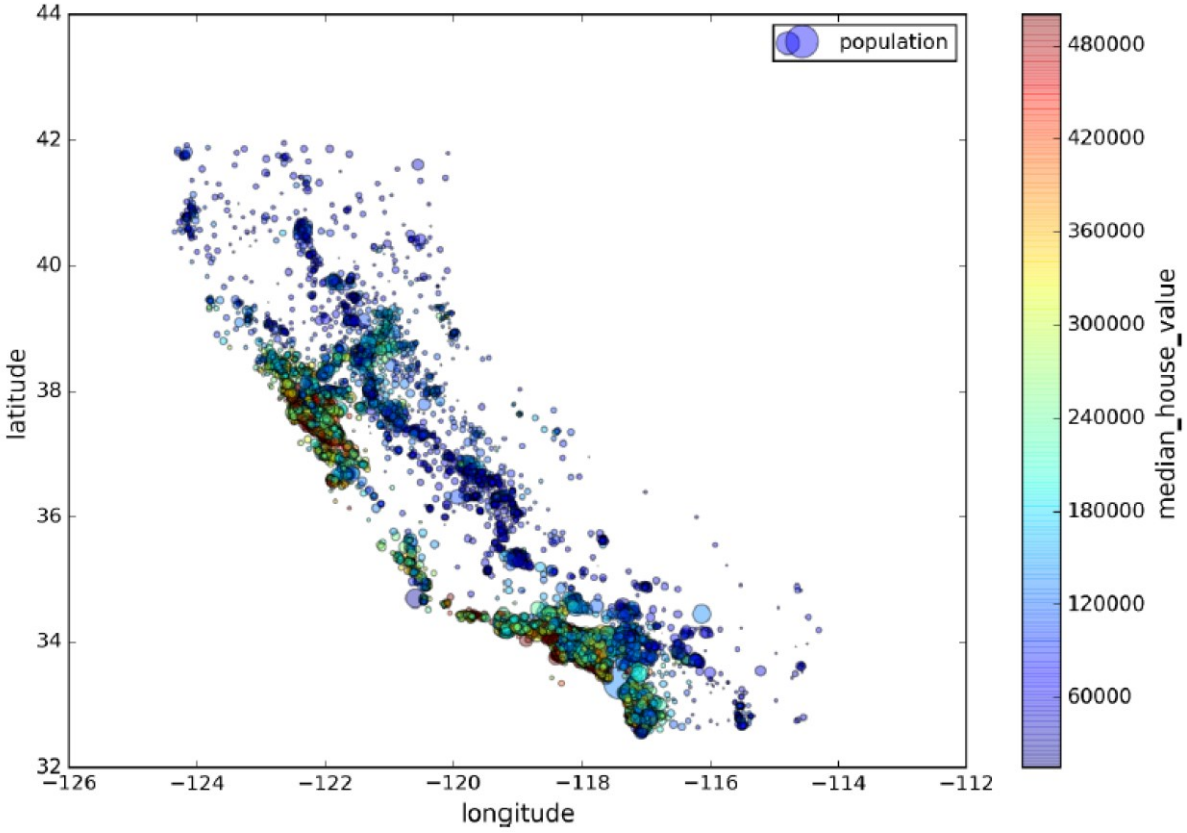
Highlighting Population and Prices

- To highlight housing prices let us use circles' radius to represent districts' population (option s), and colors to represent prices (option c).
- We can use a predefined color map called jet, which ranges from blue (low values) to red (high prices):

```
housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.4,
    s=housing["population"]/100, label="population", figsize=(10,7),
    c="median_house_value", cmap=plt.get_cmap("jet"), colorbar=True,
)
plt.legend()
```

Plotting Population and Prices

Housing prices seems to be related to the location (e.g., close to the ocean) and to the population density, which we maybe already knew.



Looking for Correlations

- In our example the dataset is not too large
  - we can easily compute the standard correlation coefficient (also called Pearson's r) between every pair of attributes
- The correlation coefficient ranges from –1 to 1.
  - When it is close to 1, it means that there is a strong positive correlation
  - When the coefficient is close to –1, it means that there is a strong negative correlation
  - When the coefficient is close to 0, it means no correlation

Correlations in Housing Data

- Let's use the Pearson's coefficient to verify how each attribute correlates with the median house value:

```
>>> corr_matrix["median_house_value"].sort_values(ascending=False)
```

median_house_value	**1.000000**
median_income	**0.687170**
total_rooms	**0.135231**
housing_median_age	**0.114220**
Households	**0.064702**
total_bedrooms	**0.047865**
Population	**-0.026699**
Longitude	**-0.047279**
Latitude	**-0.142826**

Name: median_house_value, dtype: float64

- *the median house value tends to go up with the median income*
- *a small negative correlation between latitude and median house value (prices slightly tend to go down when going to north).*

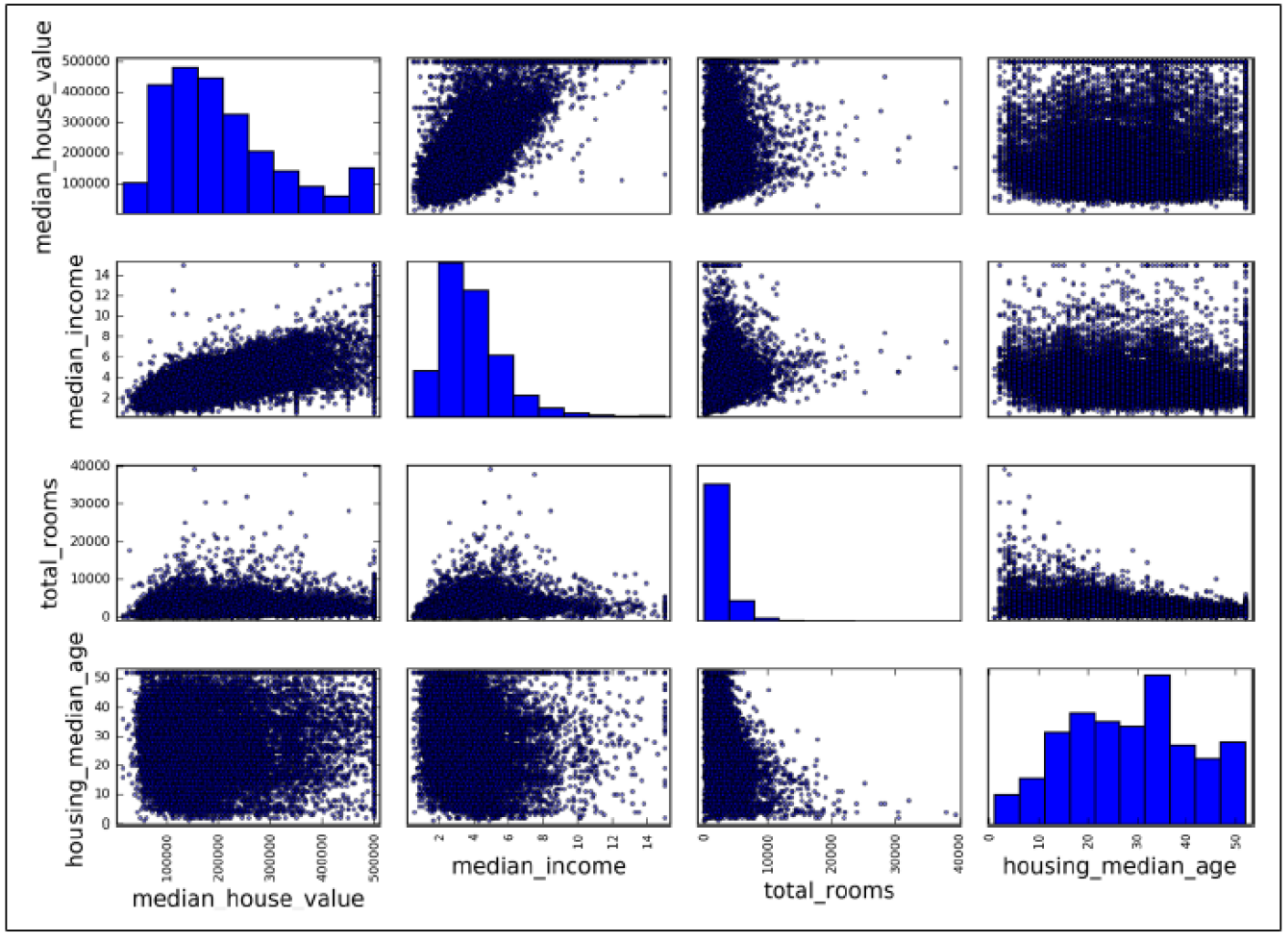
Correlations with Scatter Matrix

- Another way to check for correlations is to use Pandas' scatter_matrix function, which plots a numerical attribute against another numerical attribute.
- Since we have 11 numerical attributes, we would should have $11^2 = 121$ plots, so better focus on few promising attributes mostly correlated with median housing value

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

Housing Dataset's Scatter Matrix

Histograms are used to avoid plotting each variable against itself, since it would only produce straight lines. Pandas provide other options

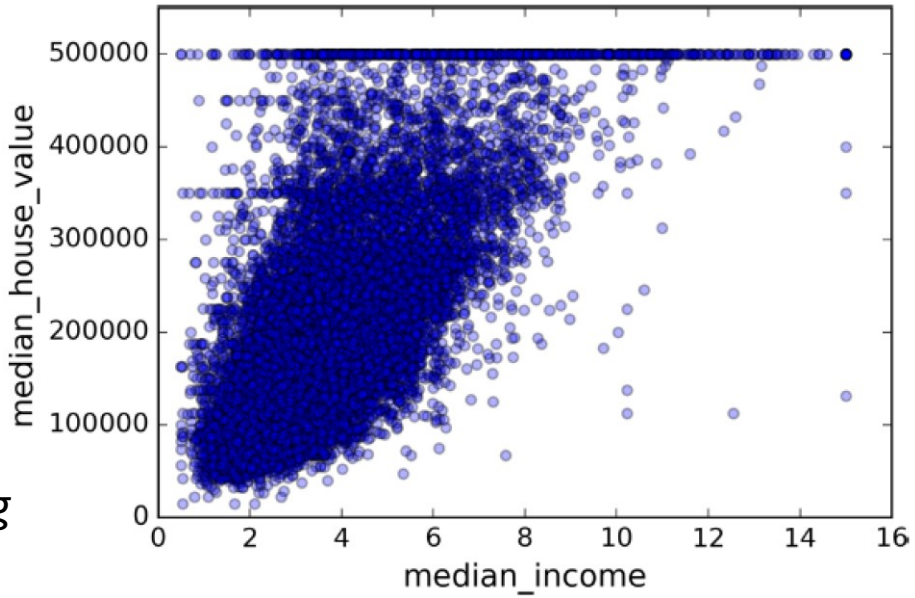


Median Income vs House Value

- The most promising attribute to predict the median house value is the median income, so let's zoom their correlation scatterplot:

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

- *The correlation is evident from the upward trend and the points are not too dispersed.*
- *The price cap at $500,000 is clearly visible as a horizontal line.*
- *Other less obvious straight lines around $450,000, $350,000, and perhaps $280,000, may suggest removing the corresponding districts to prevent our algorithms from learning to reproduce these data quirks.*



Attribute Combinations

- One last thing before preparing the data for Machine Learning algorithms is to try creating new attributes by combining existing ones:
  - Instead of the total number of rooms in a district, we probably need the number of rooms per household.
  - Similarly, the total number of bedrooms maybe needs to be compared to the number of rooms.
  - Also the population per household also seems an interesting attribute combination to look at.

```
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"]
```

New Correlations

- Let us compute the correlation matrix again:

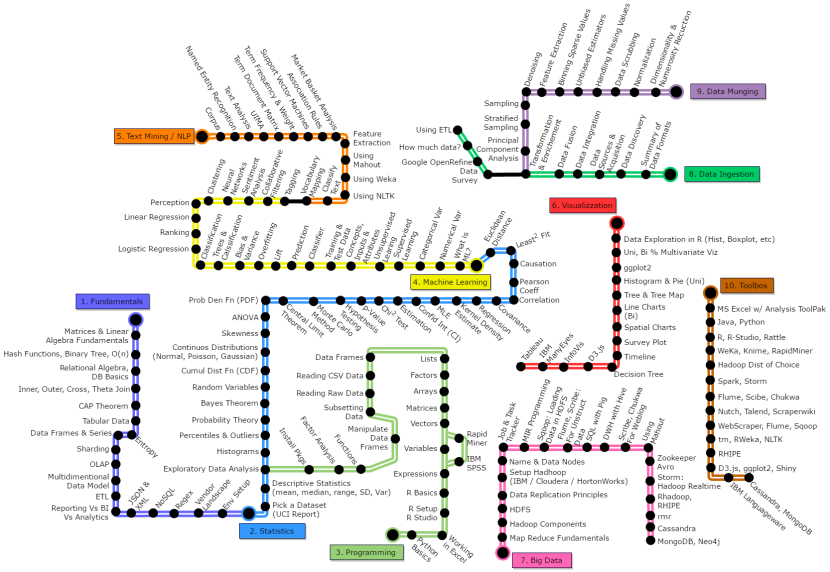
```
>>> corr_matrix = housing.corr()
>>> corr_matrix["median_house_value"].sort_values(ascending=False)
```

median_house_value	1.000000
median_income	0.687160
rooms_per_household	**0.146285**
total_rooms	0.135097
housing_median_age	0.114110
households	0.064506
total_bedrooms	0.047689
population_per_household	**-0.021985**
population	-0.026920
longitude	-0.047432
latitude	-0.142724
bedrooms_per_room	**-0.259984**

Name: median_house_value, dtype: float64

- The new bedrooms_per_room attribute is more (negatively) correlated with the median house value than the total number of rooms or bedrooms (houses with a lower bedroom/room ratio tend to be more expensive).
- The number of rooms per household is also more informative than the total number of rooms in a district

Overview



1. Main steps of a ML Project
2. Look at the big picture.
3. Get the data.
4. Visualize the data to gain insights.
5. **Prepare the data for ML algorithms.**
6. Select a model and train it.
7. Fine-tune the model.
8. Present the solution.
9. Launch, monitor, and maintain the system.

Prepare Data for ML Algorithms

- It's time to prepare the data for ML algorithms.
- Instead of doing it manually, better write functions, for several good reasons:
  - This will allow us to reproduce transformations on any dataset (e.g., the next time we get a fresh dataset).
  - We will gradually build a library of transformation functions that we can reuse in future projects.
  - We can use these functions in our live system to transform the new data before feeding it to algorithms.
  - This will make it possible to easily try various transformations and see which combination of transformations works best.

Separating Predictors and Labels

- Let us first create clean training set, by copying strat_train_set, and separating predictors and labels
- We don't necessarily want to apply the same transformations to predictors and target values:

```
# drop() creates a copy, without affecting strat_train_set
housing = strat_train_set.drop("median_house_value", axis=1)
housing_labels = strat_train_set["median_house_value"].copy()
```

Data Cleaning

- ML algorithms perform poorly with missing features
- For instance, total_bedrooms has some missing values, and we have three options to fix this problem:
  - Get rid of the corresponding districts
  - Get rid of the whole attribute
  - Set Nulls to some value (zero, the mean, the median, etc.)
- We can accomplish these by using DataFrame's dropna(), drop(), and fillna() methods:

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

 # True replacement in place, it returns nothing, with false returns a copy of the object)

Fixing Missing Values

- If we replaced nulls with total_bedrooms' median value computed on the training set, we need to save it to replace nulls also in the test set and in new data
- Scikit-Learn provides the class Imputer to take care of missing values.
- We first need to create an Imputer instance, specifying that we want to replace attribute's missing values with the median of the attribute:

```
from sklearn.preprocessing import Imputer
imputer = Imputer(strategy="median")
```

Fitting Imputer Instance on Data

- Since the median can only be computed on numerical attributes, we need to create a copy of the data without the text attribute ocean_proximity, and then fit the imputer instance to the training data through the fit() method:

```
housing_num = housing.drop("ocean_proximity", axis=1)
imputer.fit(housing_num)
```

- Even if only total_bedrooms had nulls, the imputer computed the median for all attributes, storing the results in its statistics_ instance variable.
- This will be useful in case of missing values in new data

Replacing Nulls through Imputer

- We can now use the "trained" imputer to replace nulls with median values :

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

- X is a Numpy array containing transformed features
- To put it back into a Pandas DataFrame:

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

Text & Categorical Attributes

- Most ML algorithms prefer to work with numbers
- To convert categories of a categorical attribute to numbers we can use Pandas' factorize() method:

```
>>> housing_cat_encoded, housing_categories = housing_cat.factorize()
>>> housing_cat_encoded[:10]
array([0, 0, 1, 2, 0, 2, 0, 2, 0, 0])

>>> housing_categories
Index(['<1H OCEAN', 'NEAR OCEAN', 'INLAND', 'NEAR BAY', 'ISLAND'], dtype='object')
```

- This conversion schema assumes that nearby values are more similar than distant ones, which is not our case (categories 0 and 4 are more similar than 0 and 2)

One-hot Encoding

- To fix the proximity problem when converting categorical attributes, we can create as many binary attributes as the number of categories, setting to 1 the one corresponding to the category of the district and 0 the remaining ones
- This is called one-hot encoding, because only one attribute will be equal to 1 (hot), while the others will be 0 (cold):

```
>>> from sklearn.preprocessing import OneHotEncoder
>>> encoder = OneHotEncoder()
#since  housing_cat_encoded is a 1D array, we need to reshape it as a 2D one
>>> housing_cat_1hot = encoder.fit_transform(housing_cat_encoded.reshape(-1,1))
```

Sparse vs Dense Encoded Matrix

```
>>> housing_cat_1hot
<16512x5 sparse matrix of type '<class 'numpy.float64'>'
with 16512 stored elements in Compressed Sparse Row format>
```

- Notice that the output is a SciPy sparse matrix, instead of a NumPy array.
- This is useful when having attributes with thousands categories, since one-hot encoding yields a matrix with thousands columns full of 0, and a single 1 per row.
- Instead of wasting tons of memory, a sparse matrix only stores the location of nonzero elements.
- toarray() converts it back to a dense NumPy array

Custom Transformers

- Although Scikit-Learn provides many useful transformers, we might need to write our own for custom cleanup operations or for combining attributes.
- We need to create a class and implement 3 methods: fit() (returning self), transform(), and fit_transform()
- We can get the last one by adding TransformerMixin as a base class.
- Adding BaseEstimator, we will also get two extra methods (get_params() and set_params()) useful for automatic hyperparameter tuning (avoiding *args and **kargs in our constructor)

Custom Transformers

- The following transformer class adds the combined attributes discussed earlier:

```
from sklearn.base import BaseEstimator, TransformerMixin
rooms_ix, bedrooms_ix, population_ix, household_ix = 3, 4, 5, 6
class CombinedAttributesAdder(BaseEstimator, TransformerMixin):
def __init__(self, add_bedrooms_per_room = True): # no *args or **kargs
self.add_bedrooms_per_room = add_bedrooms_per_room
def fit(self, X, y=None):
return self # nothing else to do
def transform(self, X, y=None):
rooms_per_household = X[:, rooms_ix] / X[:, household_ix]
population_per_household = X[:, population_ix] / X[:, household_ix]
if self.add_bedrooms_per_room:
bedrooms_per_room = X[:, bedrooms_ix] / X[:, rooms_ix]
return np.c_[X, rooms_per_household, population_per_household,
bedrooms_per_room]
else:
return np.c_[X, rooms_per_household, population_per_household]
attr_adder = CombinedAttributesAdder(add_bedrooms_per_room=False)
housing_extra_attribs = attr_adder.transform(housing.values)
```

The hyperparameter add_bedrooms_per_room allows us to find out whether adding this attribute helps the ML algorithms or not. We can add a hyperparameter For data preparation steps we are not 100% sure. Automating these data preparation steps helps trying several combinations automatically.

Feature Scaling

- ML algorithms don't perform well when numerical attributes have very different scales
- For example, total number of rooms ranges from 6 to 39,320, while median incomes ranges from 0 to 15
- There are two strategies to uniform scales: min-max scaling (a.k.a. normalization) and standardization
- The former shifts and rescales data to make them range from 0 to 1, by subtracting the min value and dividing by the max minus the min.
- Standardization does not bound values to a specific range, since it may be a problem for some algorithms

Normalization vs Standardization

- Standardization is much less affected by outliers.
- For example, suppose a district had (by mistake) 100 as median income, Normalization would crush all other values from 0–15 to 0–0.15
- Scikit-Learn provides a transformer called StandardScaler for standardization.

As for other transformations, scalers must be fit to the training data only, and only afterwards used to transform the training set and the test set (and new data).

Transformation Pipelines

- Scikit-Learn provides the Pipeline class to put sequences of transformations in the right order.
- The Pipeline constructor takes a list of name/estimator pairs defining a sequence of steps.
- All but the last estimator must be transformers (i.e., they must have a fit_transform() method):

```
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)
```

Pipelines' Methods

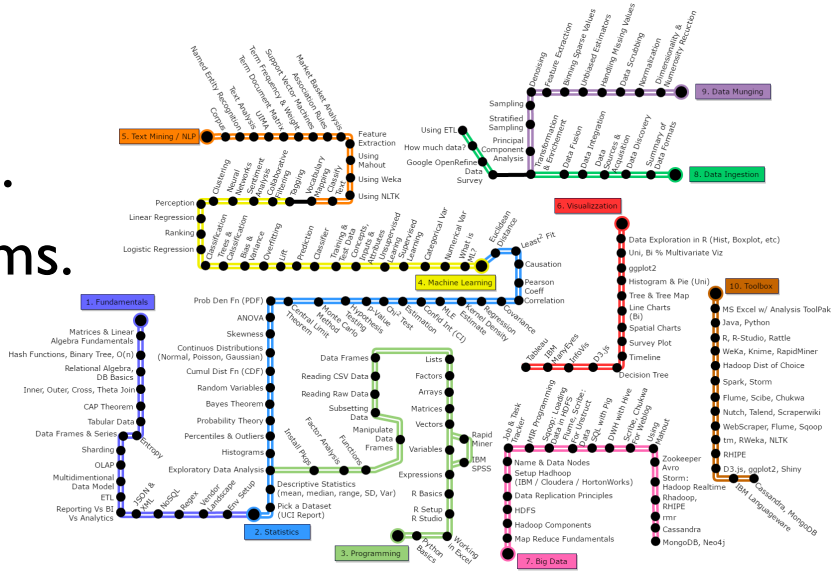
- When invoking the pipeline's fit() method, it calls fit_transform() sequentially on all transformers, passing the output of a call as parameter to next call
- When reaching the final estimator it just calls fit()
- The pipeline exposes same methods as the final estimator.

Joining Pipelines

- To join pipelines we can use FeatureUnion class from Scikit-Learn.
- When its transform() method is called, it runs each transformer's transform() method in parallel, waits for their output, concatenates them, returning the result
- Calling its fit() method calls each transformer's fit() method

Overview

1. Main steps of a ML Project
2. Look at the big picture.
3. Get the data.
4. Visualize the data to gain insights.
5. Prepare the data for ML algorithms.
6. **Select a model and train it.**
7. Fine-tune the model.
8. Present the solution.
9. Launch, monitor, and maintain the system.
10. Summary



Select and Train a Model

- We are now ready to select and train a ML model
- 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 on a few instances from the training data:

```
>>> some_data = housing.iloc[:5]
>>> some_labels = housing_labels.iloc[:5]
>>> some_data_prepared = full_pipeline.transform(some_data)
>>> print("Predictions:", lin_reg.predict(some_data_prepared))
Predictions: [ 210644.6045 317768.8069 210956.4333 59218.9888 189747.5584]
>>> print("Labels:", list(some_labels))
Labels: [286600.0, 340600.0, 196900.0, 46300.0, 254500.0]
```

Measuring Performances

- Let us measure this regression model's RMSE on the whole training set

```
>>> 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
68628.198198489219
```

- This is not a great score:
  - most districts' median_housing_values range between $120,000 and $265,000, so a typical prediction error of $68,628 is not very satisfying
  - Maybe we underfit training data: either features do not provide enough information or the model is not powerful enough

Fixing Underfitting

- To fix underfitting we can select a more powerful model, feed the training algorithm with better features, or reduce the constraints on the model
- We can exclude the last option, since the used model is not regularized
- Let us try train a Decision Tree Regressor:

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

Fixing Underfitting

- Let us try the Decision Tree regressor on a few instances from 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
```

- It seems to be perfect!
  - it is likely that the model has badly overfit the data.
- We don't want to touch the test set until we are confident about the model
  - we need to use part of the training set for training, and part for model validation

Evaluation using Cross-Validation

- The following code performs K-fold cross-validation: it randomly splits the training set into 10 distinct folds, then it trains and evaluates the Decision Tree model 10 times, picking a different fold for evaluation every time and training on the other 9:

```
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)
tree_rmse_scores = np.sqrt(-scores)
```

Results using Cross-Validation

```
>>> def display_scores(scores):
... print("Scores:", scores)
... print("Mean:", scores.mean())
... print("Standard deviation:", scores.std())
...
>>> display_scores(tree_rmse_scores)
Scores: [ 70232.0136482 66828.46839892 72444.08721003 70761.50186201
71125.52697653 75581.29319857 70169.59286164 70055.37863456
75370.49116773 71222.39081244]
Mean: 71379.0744771
Standard deviation: 2458.31882043
```

- Now Decision Tree seems worse than Linear Regression!
- Cross-validation allows us to get both an estimation of model's performances, as well as measure of how precise such estimation is (i.e., its standard deviation).

Cross-Validation for Linear Model

- Let's compute the same scores for the Linear Regression model

```
>>> 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: [ 66782.73843989 66960.118071 70347.95244419 74739.57052552
68031.13388938 71193.84183426 64969.63056405 68281.61137997
71552.91566558 67665.10082067]
Mean: 69052.4613635 #Error lower than Decision Tree
Standard deviation: 2731.6740018
```

- The Decision Tree model is overfitting so badly that it performs worse than the Linear Regression model

Random Forest Regressor

- Let's try the Random Forest Regressor, which trains many Decision Trees on random subsets of the features, averaging their predictions.
- Building a model on top of other models is called Ensemble Learning

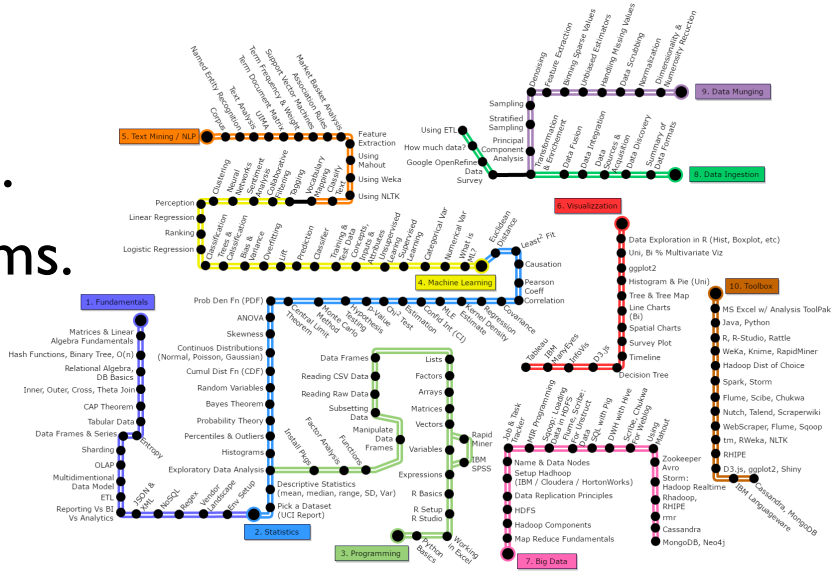
```
>>> from sklearn.ensemble import RandomForestRegressor
>>> forest_reg = RandomForestRegressor()
>>> forest_reg.fit(housing_prepared, housing_labels)
>>> [...]
>>> forest_rmse
21941.911027380233
>>> display_scores(forest_rmse_scores)
Scores: [ 51650.94405471 48920.80645498 52979.16096752 54412.74042021
50861.29381163 56488.55699727 51866.90120786 49752.24599537
55399.50713191 53309.74548294]
Mean: 52564.1902524
Standard deviation: 2301.87380392
```

Shortlist of Models

- We should try several other models (several SVM with different kernels, possibly a neural network, etc.), without spending too much time tweaking hyperparameters
- The goal is to shortlist a few (two to five) promising models, before proceeding

Overview

1. Main steps of a ML Project
2. Look at the big picture.
3. Get the data.
4. Visualize the data to gain insights.
5. Prepare the data for ML algorithms.
6. Select a model and train it.
7. **Fine-tune the model.**
8. Present the solution.
9. Launch, monitor, and maintain the system.



Fine-Tune The Model

- Having a shortlist of promising models
  - We now need to fine-tune them
- Grid Search: fiddle with hyperparameters manually, until we find a good combination of hyperparameter values (tedious work!)
- Randomized Search: evaluate a given number of random combinations by selecting a random value for each hyperparameter at every iteration
- Ensemble Methods: combine models that perform best
- Analyze the best models and their errors

Grid Search

- To avoid manually finding the right combination of hyperparameter's values, Scikit-Learn's GridSearchCV can be used.
- We need to tell it which hyperparameters we want it to experiment with, and what values to try out
- It will evaluate all the possible combinations of hyperparameter values, using cross-validation 5

Grid Search for RandomForest Regressor

- As an example the following code searches the best combination of hyperparameter values for the RandomForestRegressor:

```
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)
```

Comments to Grid Search Code

- This param_grid tells Scikit-Learn to first evaluate $3 \times 4 = 12$ combinations of n_estimators and max_features hyperparameter values
- Then, $2 \times 3 = 6$ combinations, with the bootstrap hyperparameter set to False (True is the default value).
- Thus, grid search explores 18 hyperparameter values combinations, training each model five times (five-fold cross validation), for a total of 90 rounds of training!
- To print the achieved best combination of parameters:

```
>>> grid_search.best_params_
{'max_features': 8, 'n_estimators': 30}
```

Best Estimators

- Since 8 and 30 are the max evaluated values, we should probably try searching again with higher values, since the score might improve.
- We can also get the best estimator directly:

```
>>> grid_search.best_estimator_
RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=None,
max_features=8, max_leaf_nodes=None, min_impurity_decrease=0.0,
min_impurity_split=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=42,
verbose=0, warm_start=False)
```

- *If GridSearchCV is initialized with refit=True (which is default), then once it finds the best estimator using crossvalidation, it retrains it on the whole training set*

Printing Evaluation Scores

```
>>> cvres = grid_search.cv_results_
>>> for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
print(np.sqrt(-mean_score), params)
```

```
...
63647.854446 {'n_estimators': 3, 'max_features': 2}
55611.5015988 {'n_estimators': 10, 'max_features': 2}
53370.0640736 {'n_estimators': 30, 'max_features': 2}
60959.1388585 {'n_estimators': 3, 'max_features': 4}
52740.5841667 {'n_estimators': 10, 'max_features': 4}
50374.1421461 {'n_estimators': 30, 'max_features': 4}
58661.2866462 {'n_estimators': 3, 'max_features': 6}
52009.9739798 {'n_estimators': 10, 'max_features': 6}
50154.1177737 {'n_estimators': 30, 'max_features': 6}
57865.3616801 {'n_estimators': 3, 'max_features': 8}
51730.0755087 {'n_estimators': 10, 'max_features': 8}
49694.8514333 {'n_estimators': 30, 'max_features': 8}   #Minimum MSE
62874.4073931 {'n_estimators': 3, 'bootstrap': False, 'max_features': 2}
54643.4998083 {'n_estimators': 10, 'bootstrap': False, 'max_features': 2}
59437.8922859 {'n_estimators': 3, 'bootstrap': False, 'max_features': 3}
52735.3582936 {'n_estimators': 10, 'bootstrap': False, 'max_features': 3}
57490.0168279 {'n_estimators': 3, 'bootstrap': False, 'max_features': 4}
51008.2615672 {'n_estimators': 10, 'bootstrap': False, 'max_features': 4}
```

RMSE score for best combination is 49,694, slightly better than 52,564 score we got with default hyperparameter values. We have fine-tuned our best model!

Data Preparation with Hyperparameters

- We can treat some data preparation steps as hyperparameters
- For example, grid search can automatically verify whether to add a feature we were not sure about (e.g., using add_bedrooms_per_room hyperparameter of our CombinedAttributesAdder transformer)
- It may may also find the best way to handle outliers, missing features, feature selection, and more

Randomized Search

- When the hyperparameter search space is large, it is often preferable to use RandomizedSearchCV
- W.r.t. GridSearchCV, it evaluates only a given number of random combinations by selecting a random value for each hyperparameter at each iteration
- Main advantages:
  - If we let randomized search run for *n* iterations, it will explore *n* different values for each hyperparameter (instead of just a few values with GridSearch)
  - We have more control on the computing budget to allocate for hyperparameter search, setting the number of iterations

Ensemble Methods

- Another way to fine-tune our system is to combine models that perform best
- The "ensemble" often performs better than the best individual models, especially if these make very different types of errors

Analyze Best Models

- We often gain good insights on the problem by inspecting the best models
- For example, the RandomForestRegressor can indicate the relative importance of each attribute for making accurate predictions:

```
>>> feature_importances =
grid_search.best_estimator_.feature_importances_
>>> feature_importances
array([ 7.33442355e-02, 6.29090705e-02, 4.11437985e-02,
        1.46726854e-02, 1.41064835e-02, 1.48742809e-02,
        1.42575993e-02, 3.66158981e-01, 5.64191792e-02,
        1.08792957e-01, 5.33510773e-02, 1.03114883e-02,
        1.64780994e-01, 6.02803867e-05, 1.96041560e-03,
        2.85647464e-03])
```

Let's display these importance scores next to their corresponding attribute names

Feature Importance

```
>>> extra_attribs = ["rooms_per_hhold", "pop_per_hhold",
"bedrooms_per_room"]
>>> cat_encoder = cat_pipeline.named_steps["cat_encoder"]
>>> cat_one_hot_attribs = list(cat_encoder.categories_[0])
>>> attributes = num_attribs + extra_attribs + cat_one_hot_attribs
>>> sorted(zip(feature_importances, attributes), reverse=True)
[(0.36615898061813418, 'median_income'),
(0.16478099356159051, 'INLAND'),
(0.10879295677551573, 'pop_per_hhold'),
(0.073344235516012421, 'longitude'),
(0.062909070482620302, 'latitude'),
(0.056419179181954007, 'rooms_per_hhold'),
(0.053351077347675809, 'bedrooms_per_room'),
(0.041143798478729635, 'housing_median_age'),
(0.014874280890402767, 'population'),
(0.014672685420543237, 'total_rooms'),
(0.014257599323407807, 'households'),
(0.014106483453584102, 'total_bedrooms'),
(0.010311488326303787, '<1H OCEAN'),
(0.0028564746373201579, 'NEAR OCEAN'),
(0.0019604155994780701, 'NEAR BAY'),
(6.0280386727365991e-05, 'ISLAND')]
```

- *We may want to try dropping less useful features (apparently only one ocean_proximity category is useful, so we could try dropping the others).*

Evaluation on the Test Set

- Once the tuned model performs sufficiently well, it is the time to evaluate it on the test set.
- All we have to do is to get predictors and labels from our test set, run the full_pipeline to transform the data (call transform(), not fit_transform()!), and evaluate the final model on the test set:

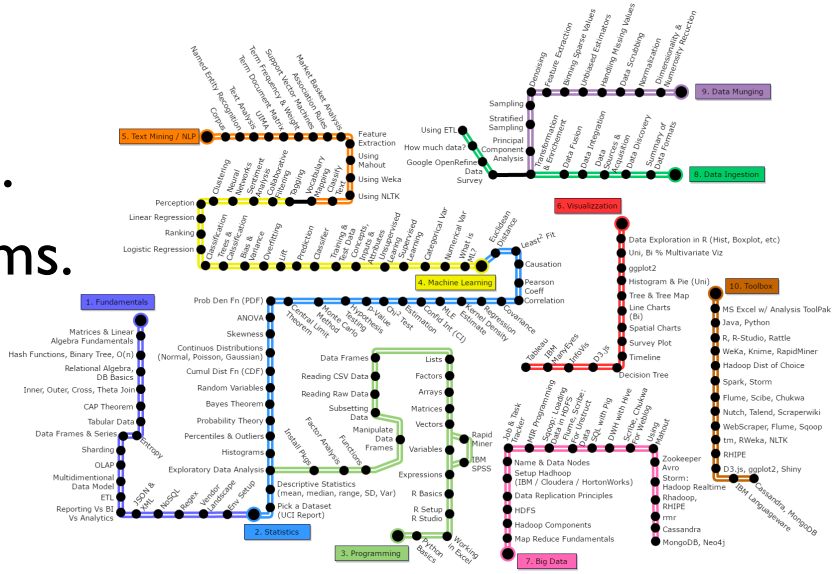
```
final_model = grid_search.best_estimator_
X_test = strat_test_set.drop("median_house_value", axis=1)
y_test = strat_test_set["median_house_value"].copy()
X_test_prepared = full_pipeline.transform(X_test)
final_predictions = final_model.predict(X_test_prepared)
final_mse = mean_squared_error(y_test, final_predictions)
final_rmse = np.sqrt(final_mse) # => evaluates to 47,766.0
```

Performances on the Test Set

- Performances on the test set will usually be worse than before if we did a lot of hyperparameter tuning
- We must avoid tweaking hyperparameters, since the improvements would unlikely generalize to new data
- In case of extremely poor performances, better review some previous choice
- Once we get acceptable results on the test set, it is time to present our solution

Overview

1. Main steps of a ML Project
2. Look at the big picture.
3. Get the data.
4. Visualize the data to gain insights.
5. Prepare the data for ML algorithms.
6. Select a model and train it.
7. Fine-tune the model.
8. **Present the solution.**
9. Launch, monitor, and maintain the system.

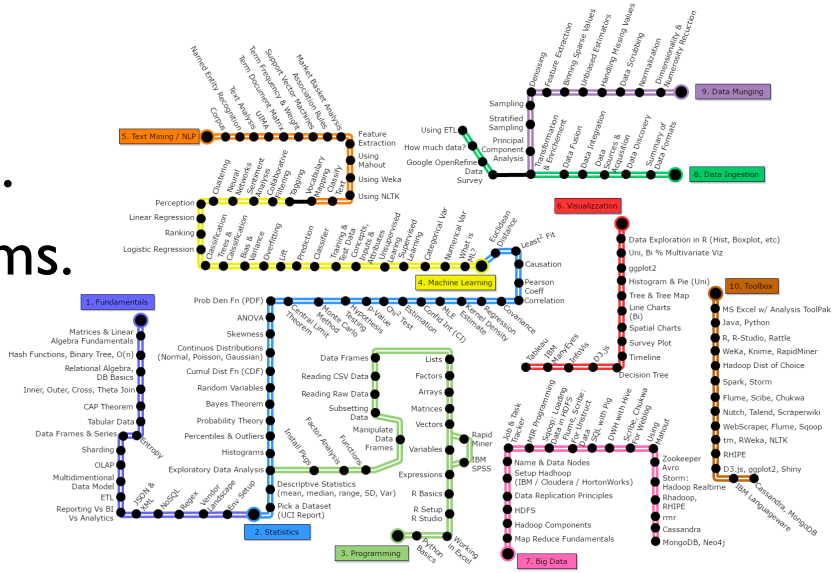


Present the Solution

- To present our solution we should:
  - highlight what we have learned
  - what assumptions were made
  - what are system's limitations
  - document everything
  - create presentations with clear visualizations and easy-to-remember statements (e.g., "the median income is the number one predictor of housing prices")

Overview

1. Main steps of a ML Project
2. Look at the big picture.
3. Get the data.
4. Visualize the data to gain insights.
5. Prepare the data for ML algorithms.
6. Select a model and train it.
7. Fine-tune the model.
8. Present the solution.
9. **Launch, monitor, and maintain the system.**



Launch, Monitor, and Maintain the System

- Once got approval to launch the system, we need to:
  - get the solution ready for production;
  - write monitoring code to check system's live performance (it will require sampling system's predictions and evaluating them with an expert);
  - evaluate the system's input data quality;
  - train our models on a regular basis using fresh data (better automating this process);
  - In case of an online learning system we should save snapshots of its state at regular intervals, so we can easily roll back to a previously working state.

Conclusion

- Much of the work is in data preparation steps, building monitoring tools, setting up human evaluation pipelines, and automating model training
- ML algorithms are important, but we should spend enough time on the overall process and know 3 or 4 algorithms well rather than spend all time exploring advanced algorithms
- A good place to exercise is a competition website like http://kaggle.com/, providing a dataset to play with, a clear goal, and people to share the experience with