

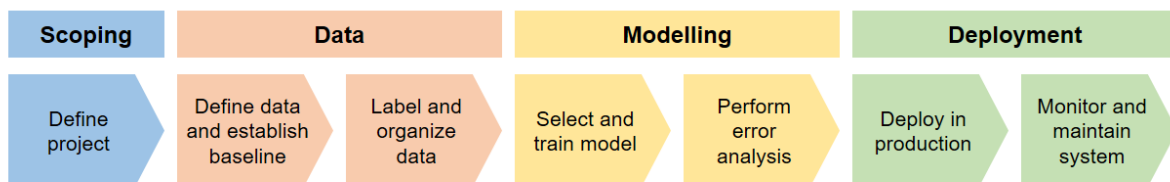
## Ch2: End-to-End Machine Learning Project

Machine Learning project checklist (from Appendix B):

1. Frame the problem and look at the big picture.
2. Get the data.
3. Explore the data to gain insights.
4. Prepare the data to better expose the underlying data patterns to ML algorithms.
5. Explore many different models and shortlist the best ones.
6. Fine-tune your models and combine them into a great solution.
7. Present your solution.
8. Launch, monitor, and maintain your system.

You'll often see slight variations of this checklist. Modify to suit your own situation and needs!

An alternate ML Project checklist (from Andrew Ng's Coursera stuff):



Introduction to our provided data set: California Housing Prices dataset (StatLib repository)

- Based on data from the 1990 California census (hence currently unrealistic).
- Author removed some features and added a categorical feature (more instructive).
- Point of possible confusion: Examples (rows) are not individual houses!!!  
Block groups (a.k.a "districts"): smallest geographical unit for which the US Census Bureau publishes sample data (a block group typically has a population of 600 to 3,000).

# 1. Frame the Problem

## Gather Information:

Talk to people. Hunt them down! This includes not only data stewards, developers responsible for the systems that created/compiled the data, and project leads or C-suite, but also anyone that is responsible for downstream components that could be affected.

The more you nail down at the start, the easier it will be to manage risk and expectations.

What is the business objective?

"Knowing the objective is important because it will determine how you frame the problem, which algorithms you will select, which performance measure you will use to evaluate your model, and how much effort you will spend tweaking it."

Proposed Objective: Prediction of a district's median housing price.

(almost) Universal Objective of Data Science: Turn data into money!

### Ancillary provided details:

- Our predictions will be fed to another ML system along with many other signals.
- Median district house price is currently estimated manually by experts via complex rules.
  - Costly and time-consuming.
  - Estimates were off by more than 20%.
- Owners of the downstream system confirmed that they want a numeric value, not a coarse-grained approximation via category (cheap/medium/expensive).

### Determinations:

- We have labels -> supervised learning task.
- We're trying to predict a continuous target -> regression problem (multiple regression).
- We're only predicting one target from each example -> univariate regression.

Recall (from the footnotes of Chapter1): Why is regression called regression?

"Fun fact: this odd-sounding name is a statistics term introduced by Francis Galton while he was studying the fact that the children of tall people tend to be shorter than their parents. Since the children were shorter, he called this regression to the mean. This name was then applied to the methods he used to analyze correlations between variables."

## Performance Measure:

Basically, how are we going to evaluate and compare models? How can we tell when we've satisfied the objective of the project?

Root Mean Square Error (RMSE) a.k.a "l2 norm".

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

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

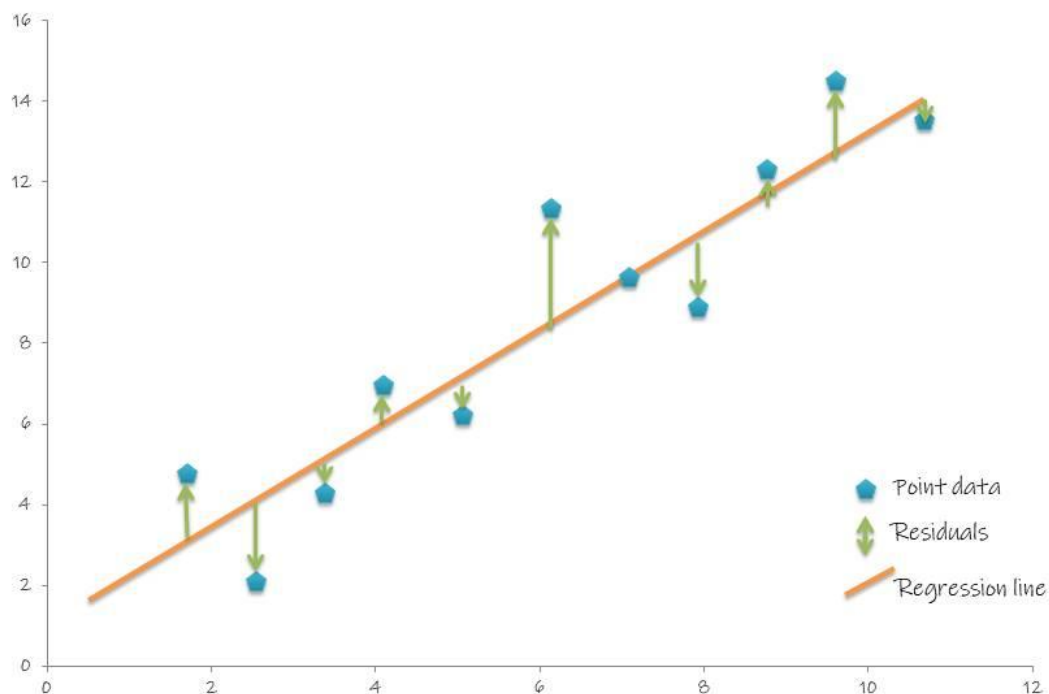
$m$  is the number of instances

$\mathbf{x}^{(i)}$  is a vector of all the feature values

$y^{(i)}$  is its label

$h(\mathbf{x}^{(i)})$  is your system's prediction function, a.k.a "hypothesis function"

$\hat{y}^{(i)} = h(\mathbf{x}^{(i)})$  is a predicted value for the target/label for that instance ( $\hat{y}$  is pronounced "y-hat").



Some notes regarding RMSE, if you're curious:

- The squaring--why?
  - To penalize large errors.
  - One of the main reasons is that it is very easy to differentiate (important for derivative-based methods such as gradient descent).
- The square root--why?
  - Brings us back to the natural, interpretable units of the problem. Without the square root, we'd end up talking about  $\$^2$ , whatever the heck that is?
- Why RMSE and not MAE?
  - The higher the norm index, the more it focuses on large values and neglects small ones. The choice of index 2 is slightly arbitrary.
  - MAE is preferable when you know you've got plenty of outliers and you know that residuals are not going to end up having a Normal/Gaussian distribution.

Author's suggestion: Check assumptions!

### Aside: Pipelines

A sequence of data processing components is called a data pipeline.

“Components typically run asynchronously... each component pulls in a large amount of data, processes it, and spits out the result in another data store.”

Is this true?

Synchronous - i.e. sequential, blocking; one task executed at a time; coordinated, or aligned with a clock or timer; executing task must return before proceeding with next task

Asynchronous - non-blocking; "fire and forget" i.e. call functions and continue doing other stuff, knowing that those functions will eventually return results on their own time

Python's asyncio (standard library package) provides typical async/await.

The key difference between synchronous and asynchronous processing is in what the processor does while it waits for an I/O task to complete.

In synchronous execution, the processor remains idle and waits for the I/O task to complete before executing the next set of instructions.

Asynchronous execution is not necessarily parallel execution; think about making breakfast.

Good example: Preparing breakfast (source?).

Pour a cup of coffee.

Heat up a pan, then fry two eggs.

Fry three slices of bacon.

Toast two pieces of bread.

Add butter and jam to the toast.

Pour a glass of orange juice.

## 2. Get the Data





### Setup:

All notebooks, data, extra goodies available at author's github repo:

<https://github.com/ageron/handson-ml2>

If you just want to run the code/notebooks, tinker around, and not deal with installing a bunch of stuff, you can just use Google Colab:

**WARNING:** Please be aware that these services provide temporary environments: anything you do will be deleted after a while, so make sure you download any data you care about.

-  Open in Colab
-  Open in Kaggle
-  launch binder
-  Launch in Deepnote



If you want to run things on your own machine:

Preferred: Anaconda--the easiest way to get up and running.

Less Preferred: What the author does in the book i.e. venv/virtualenv/whatever (unless you have a particular reason to use these).

For Cool Kids: Docker!

Aside: The importance of virtual environments--it's all about dependencies.

Official Python Documentation: "A virtual environment is a Python environment such that the Python interpreter, libraries and scripts installed into it are isolated from those installed in other virtual environments, and (by default) any libraries installed in a "system" Python, i.e., one which is installed as part of your operating system."

Why bother?

- Easier to work on different projects while avoiding package version conflicts--different envs for different types of projects.
- By keeping project dependencies static, or at least isolated, predictable, and explicit, you ensure that if you revisit your project at a later time, it'll actually run. The python DS/ML ecosystem evolves at a rapid pace; functions get deprecated, and package APIs change all the time.
- For ease of sharing and collaboration. If someone wants to run your code, rather than having to guess which versions of the dependencies you used, they can just recreate their own version of your environment (from a file, which you *ought* to have provided).
  - Ex: `$ conda env export > environment.yaml`  
`$ conda env create -f <path_to_yaml_file>`
- Avoiding headaches.
  - If you bork one env, at least your others are fine.
  - `conda 'solving environment'...` wait 487593453 hours, or just bail?

DS/ML folks frequently use 'conda' for package/environment management. It comes with Anaconda/Miniconda distributions, and it works pretty dang well (until it doesn't).

General python developers tend to use 'venv' (part of the python standard library), a 'venv' extension that tries to fix particular limitations/annoyances ('virtualenv', 'virtualenvwrapper'), or 'venv' analog ('pyenv', 'pipenv').

Suggestion: As you build up your various environments over time, try to stick with 'conda' as long as you can. When the time comes that you can't, just switch to using 'pip'.

### Basic example of first time using:

```
$ conda create -n <env_name> jupyter matplotlib numpy pandas scipy scikit-learn
$ conda activate <env_name>
$ jupyter notebook
# If your default browser hasn't popped up, just manually go to http://localhost:8888/
```

### Helpful:

```
$ conda init          # if you didn't already agree to have the Anaconda installer do this for you
$ conda info          # spits out a bunch of version
$ conda env list      # shows you available envs and which one is currently activate (*)
$ conda list --explicit # lists packages installed in your currently activated env
```

## Take a Quick Look at the Data Structure:

The core pandas object is the DataFrame; think of it like an excel sheet on steroids or a table in a SQL database. Each column in the DataFrame is a Series object, which is a one-dimensional ndarray with axis labels; think of it as a snazzy array/list.

### Most useful methods for DataFrames/Series:

head(), or alternatively sample()

housing.head()										
executed in 51ms, finished 15:38:27 2021-10-15										
Out[5]:	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value	ocean_proximity
0	-122.23	37.88	41.0	880.0	129.0	322.0	126.0	8.3252	452600.0	NEAR BAY
1	-122.22	37.86	21.0	7099.0	1106.0	2401.0	1138.0	8.3014	358500.0	NEAR BAY
2	-122.24	37.85	52.0	1467.0	190.0	496.0	177.0	7.2574	352100.0	NEAR BAY
3	-122.25	37.85	52.0	1274.0	235.0	558.0	219.0	5.6431	341300.0	NEAR BAY
4	-122.25	37.85	52.0	1627.0	280.0	565.0	259.0	3.8462	342200.0	NEAR BAY

In [6]: housing.sample(n=10)										
Out[6]:	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value	ocean_proximity
6268	-117.94	34.05	34.0	1519.0	304.0	1262.0	300.0	3.3409	161200.0	<1H OCEAN
7234	-118.14	34.02	42.0	1384.0	458.0	1825.0	455.0	1.4178	145500.0	<1H OCEAN
5163	-118.29	33.96	39.0	1340.0	409.0	1463.0	367.0	1.5294	111400.0	<1H OCEAN
4878	-118.24	34.03	52.0	142.0	47.0	137.0	45.0	1.8333	312500.0	<1H OCEAN
14950	-116.95	32.74	7.0	2722.0	578.0	1429.0	574.0	3.9583	141700.0	<1H OCEAN
3927	-118.54	34.18	25.0	1938.0	457.0	1280.0	425.0	3.9632	240300.0	<1H OCEAN
781	-122.10	37.61	35.0	2361.0	458.0	1727.0	467.0	4.5281	173600.0	NEAR BAY
13939	-117.27	34.23	26.0	6339.0	1244.0	1177.0	466.0	3.7708	110400.0	INLAND
7124	-118.03	33.90	36.0	1143.0	193.0	826.0	188.0	5.3184	171100.0	<1H OCEAN
150	-122.22	37.81	52.0	2024.0	339.0	756.0	340.0	4.0720	270100.0	NEAR BAY

info()

```
In [6]: 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
```

Note: With Pandas, 'object' dtype usually means 'text/string'.

value\_counts()

```
In [10]: housing["ocean_proximity"].value_counts()

Out[10]: <1H OCEAN      9136
         INLAND        6551
         NEAR OCEAN    2658
         NEAR BAY      2290
         ISLAND         5
         Name: ocean_proximity, dtype: int64
```

describe() - Very good for sanity check; look min/max/mean

```
In [7]: housing.describe()

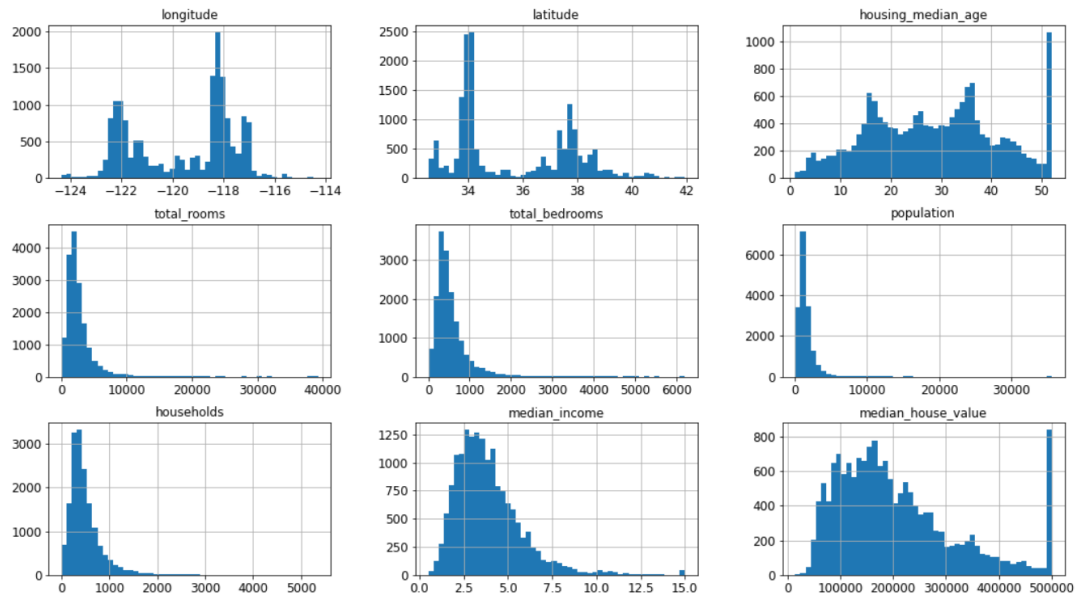
Out[7]:
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value
count	20640.000000	20640.000000	20640.000000	20640.000000	20433.000000	20640.000000	20640.000000	20640.000000	20640.000000
mean	-119.569704	35.631861	28.639486	2635.763081	537.870553	1425.476744	499.539680	3.870671	206855.816909
std	2.003532	2.135952	12.585558	2181.615252	421.385070	1132.462122	382.329753	1.899822	115395.615874
min	-124.350000	32.540000	1.000000	2.000000	1.000000	3.000000	1.000000	0.499900	14999.000000
25%	-121.800000	33.930000	18.000000	1447.750000	296.000000	787.000000	280.000000	2.563400	119600.000000
50%	-118.490000	34.260000	29.000000	2127.000000	435.000000	1166.000000	409.000000	3.534800	179700.000000
75%	-118.010000	37.710000	37.000000	3148.000000	647.000000	1725.000000	605.000000	4.743250	264725.000000
max	-114.310000	41.950000	52.000000	39320.000000	6445.000000	35682.000000	6082.000000	15.000100	500001.000000

Histograms - Quick & easy way to see an approximation of the distribution of numerical data.



```
In [55]: housing.hist(bins=50, figsize=(18,10));
```



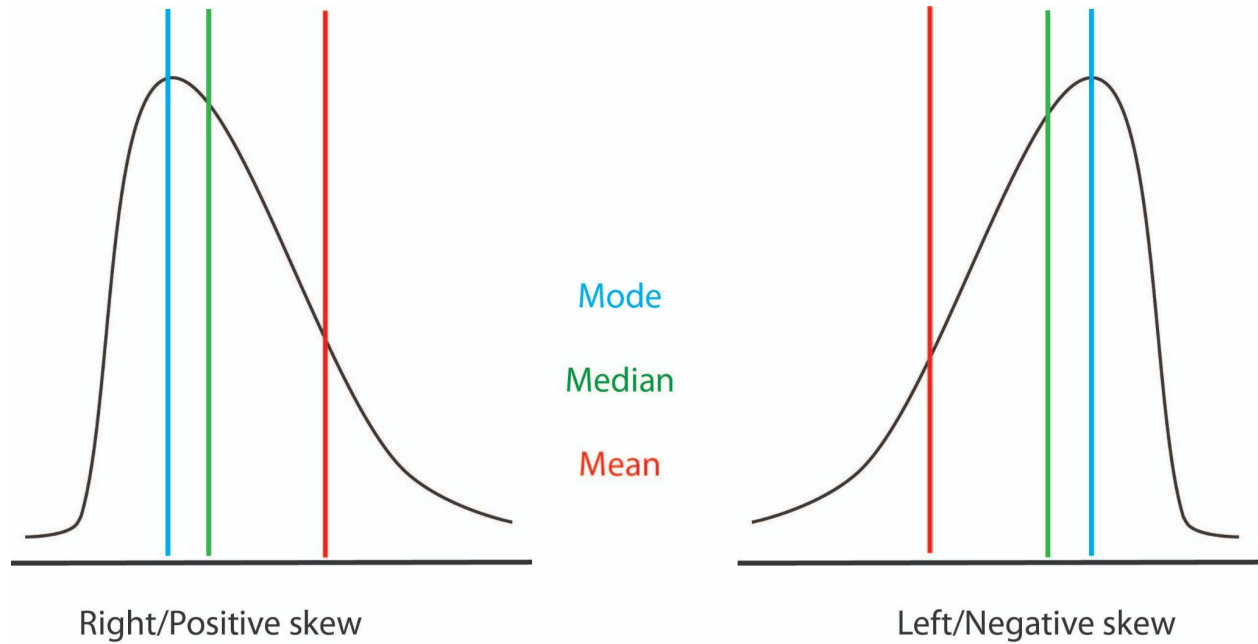
### What to look for:

- Lines/cutoffs.
  - Why? Could indicate problems with data collection, corruption, preprocessing like clipping/winsorization...
- Basic/known distribution types.
  - Why? We have more math and tools available for known distributions. Also, some model types base their theoretical foundations on assumptions about their input distributions (OLS/linear regression). Homoscedasticity?!!?!?!?!?!?!?!?
- Very pronounced skew.
  - Why? Again, considerations for model assumptions. But also, skew will be one of the factors involved with decisions about feature scaling and preprocessing.

### Takeaways:

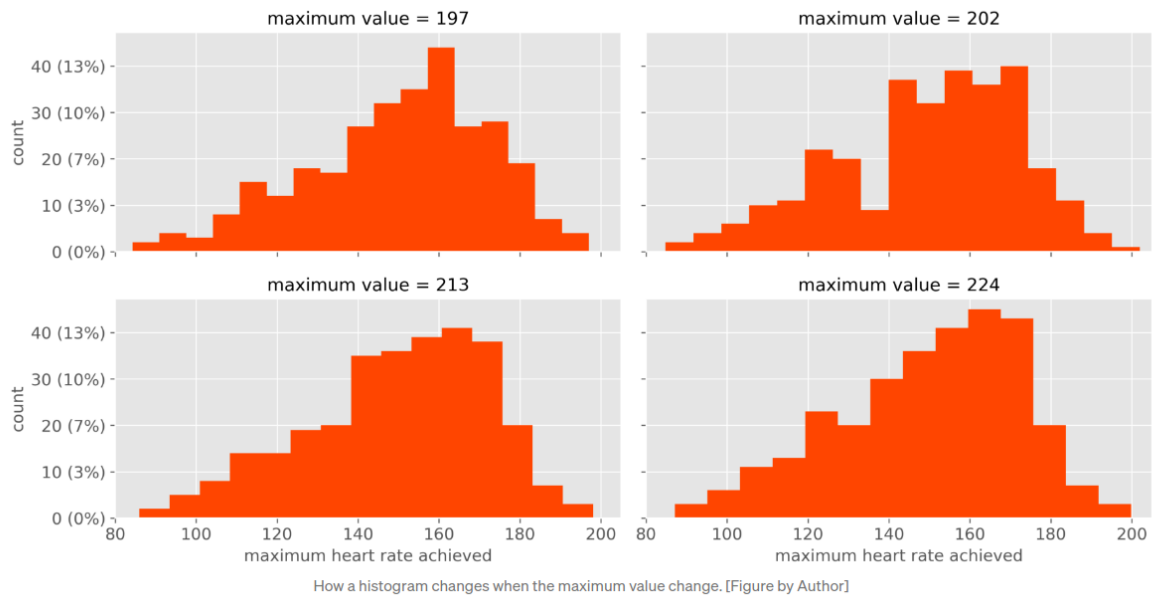
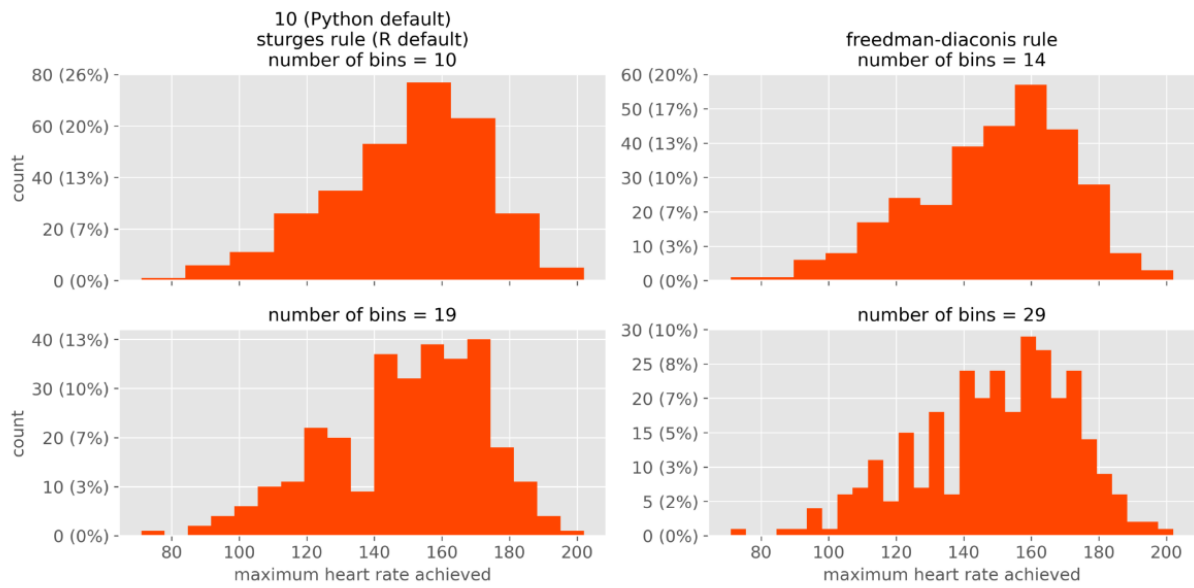
- Median income attribute does not look like it is expressed in US dollars.
  - "...the data has been scaled and capped at 15 (actually, 15.0001) for higher median incomes, and at 0.5 (actually, 0.4999) for lower median incomes. The numbers represent roughly tens of thousands of dollars (e.g., 3 actually means about \$30,000)."
  - It's an example of a preprocessed feature.
- Housing median age, median house value were also capped. Potentially problematic.
- Attributes have very different scales.
- There are some tail-heavy distributions.

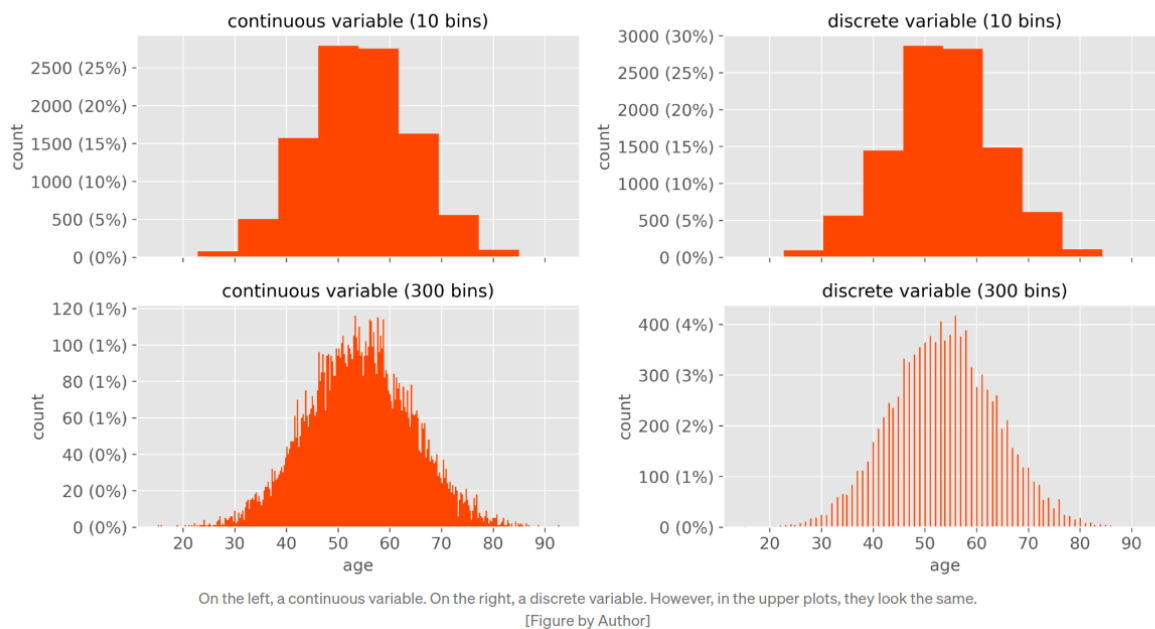
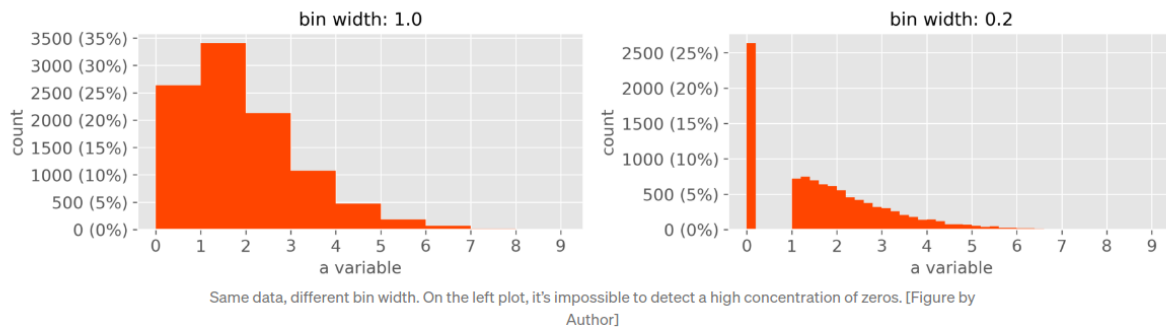
Recall left skew vs right skew: Where is the mean in relation to the median?



Warning: Histograms can be deceiving!

<https://towardsdatascience.com/6-reasons-why-you-should-stop-using-histograms-and-which-plot-you-should-use-instead-31f937a0a81c>





## Create a Test Set:

"your brain is an amazing pattern detection system, which means that it is highly prone to overfitting: if you look at the test set, you may stumble upon some seemingly interesting pattern in the test data that leads you to select a particular kind of Machine Learning model"

It's vitally important to avoid "data snooping bias", "data leakage"--anything that would give us false confidence in our model.

Many different ways to sample and split a dataset into train and test sets...

Some cool visuals:

[https://scikit-learn.org/stable/auto\\_examples/model\\_selection/plot\\_cv\\_indices.html](https://scikit-learn.org/stable/auto_examples/model_selection/plot_cv_indices.html)

Most mature data science packages are going to have some mechanism for controlling the randomness used through their code/algorithms, typically through setting a seed for the pseudo-random number generator that's used under the hood.

```
In [70]: # to make this notebook's output identical at every run
np.random.seed(42)
```

Scikit-Learn's `train_test_split()` function:

```
In [80]: from sklearn.model_selection import train_test_split

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

Discussion: Anybody actually using hashing, low-level techniques for sampling/splitting, anything other than tried-and-tested utility functions?

Stratified Sampling: The population is divided into homogeneous (def) subgroups called strata. You should not have too many strata, and each stratum should be large enough. To stratify, we need groups, hence using Panda's `pd.cut()` method to do "binning". 'median\_income' -> bin into temporary column 'income\_cat' -> use 'income\_cat' for the purpose of stratification -> drop 'income\_cat'

Scikit-Learn's `StratifiedShuffleSplit()` object:

```
In [87]: 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]
```

"We spent quite a bit of time on test set generation for a good reason: this is an often neglected but critical part of a Machine Learning project."

We tend to miss out on this when not working on a real problem; kaggle handles test sets for us, but in an actual business setting it would be our responsibility to ensure the test set is representative of the data we want to make predictions on!

Suggestion: Data Dictionary. Probably a good idea, if it hasn't already been done, to accumulate and document all the basic info you have about the data set.

## 3. Discover and Visualize the Data to Gain Insights

This part of the process is usually called **Exploratory Data Analysis (EDA)**.

Author's tip: "If the training set is very large, you may want to sample an exploration set, to make manipulations easy and fast."

Ex: `df_for_plotting = df.sample(n=df.shape[0]//10)`

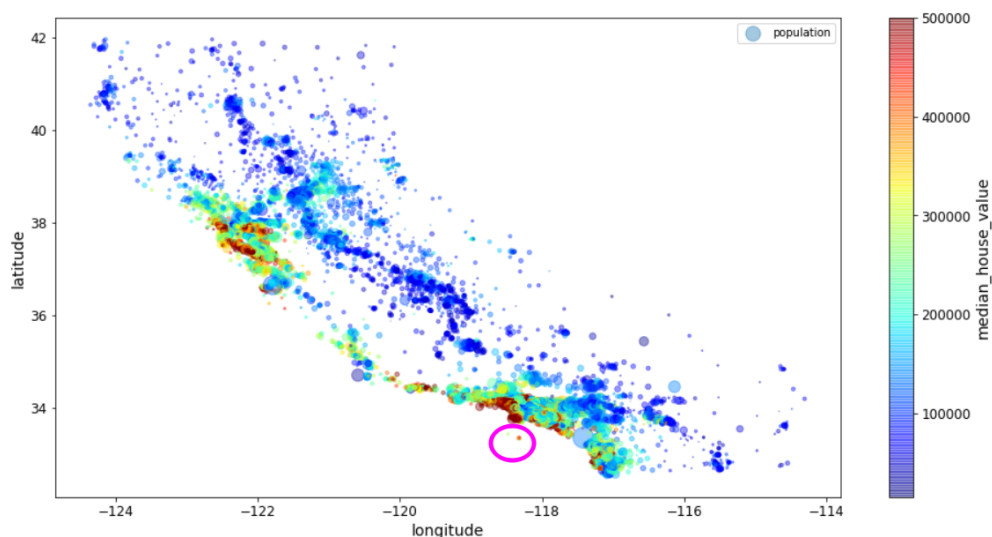
Also to mitigate visual clutter!

This is particularly true when playing with visualizations using unsupervised methods like t-SNE, UMAP, etc.

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

-> adjust 'alpha', use various sizes/colors/shapes/'hue'

```
In [37]: housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.4,
s=housing["population"]/100, label="population", figsize=(16,8),
c="median_house_value", cmap=plt.get_cmap("jet"), colorbar=True,
sharex=False)
plt.legend();
```



Location matters! Proximity to the ocean is important, as well as proximity to urban/city centers.

"A clustering algorithm should be useful for detecting the main cluster and for adding new features that measure the proximity to the cluster centers."

Extra plotting tip: Get them plots bigger!

1) Set a better default plot size, up in the 'imports' section of your notebook:

```
# Settings
import warnings
warnings.filterwarnings('ignore')
%matplotlib inline
pd.set_option('display.max_columns', None)
pd.set_option('display.max_rows', 100)
pd.set_option('display.float_format', lambda x: '%.4f' % x)
pd.set_option('display.max_colwidth', None)
plt.rcParams['figure.figsize'] = [16, 8]
plt.rcParams.update({'font.size': 25})
plt.rcParams['agg.path.chunksize'] = 10000
sns.set_style('whitegrid')
sns.set_context('notebook', font_scale=1.2, rc={'lines.linewidth': 2.0})
```

2) Explicitly create a figure object, *then* plot:

3) Use 'figsize' arg that's available in many plotting functions (see above CA scatterplot).

#### Other miscellaneous plotting tips:

- Assign the plot function to a variable, or use ';' to mute plot function object/handle output.
- Try not to be lazy... label axis, enable legends, customize tick labels/format/spacing, etc.
- Anyone else have some tips???

## Correlations:

Standard correlation coefficient (also called Pearson's r) with Pandas corr() method.

Correlation coefficient  $[-1, 1]$

~0 means no linear correlation

```
In [38]: corr_matrix = housing.corr()

In [39]: corr_matrix["median_house_value"].sort_values(ascending=False)

Out[39]: median_house_value    1.000000
         median_income      0.688075
         total_rooms       0.134153
         housing_median_age  0.105623
         households        0.065843
         total_bedrooms     0.049686
         population        -0.024650
         longitude          -0.045967
         latitude           -0.144160
         Name: median_house_value, dtype: float64
```

Warning: Common correlation coefficients (like Pearson's r) have limitations:

- Correlation coefficient only measures linear correlations.
- May completely miss out on nonlinear relationships.
- Strength of correlation is not related to slope.

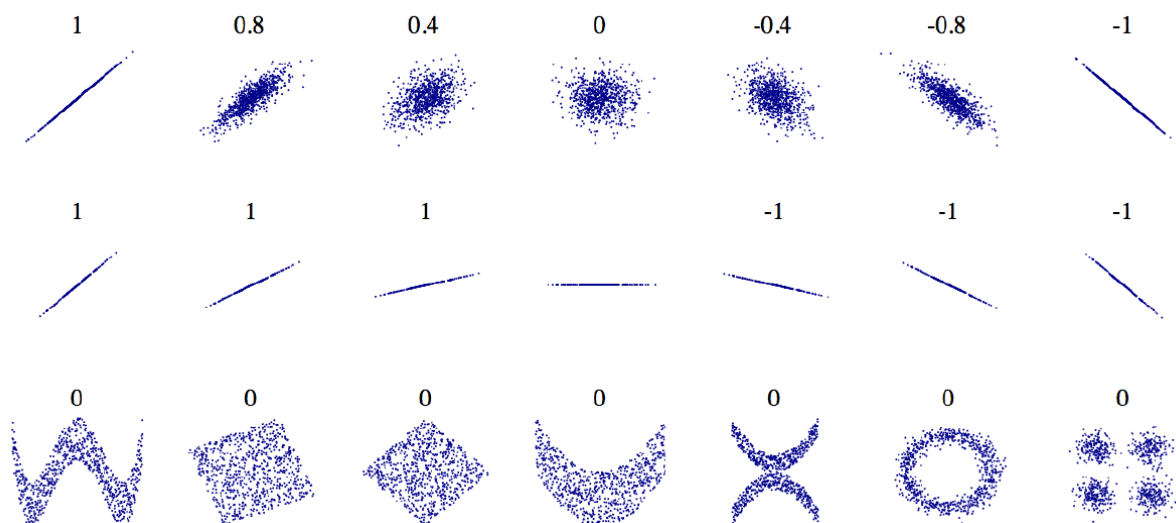
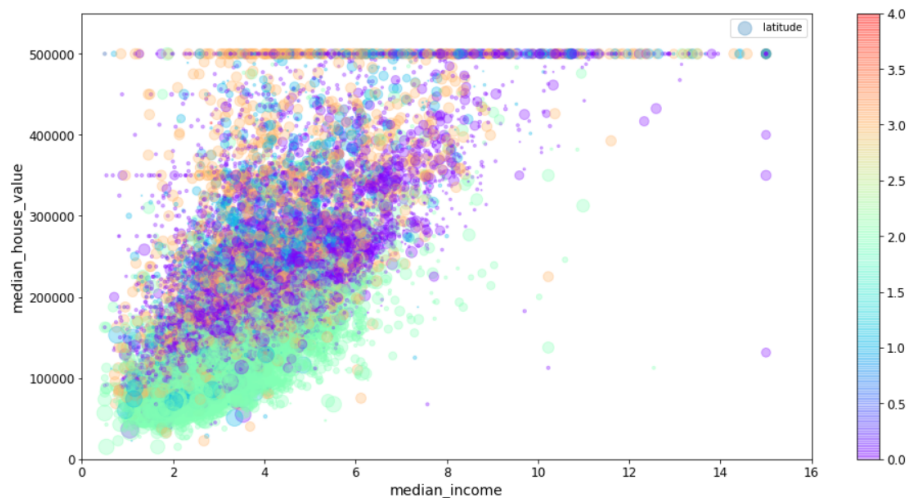


Figure 2-14. Standard correlation coefficient of various datasets (source: Wikipedia; public domain image)

Hence the need for scatter plots.

```
In [52]: housing.plot(kind="scatter", x="median_income", y="median_house_value",
s=np.power((1+housing['latitude'] - housing['latitude'].min()), 2.5), label='latitude',
c=housing['ocean_proximity'].map(dict(zip(housing['ocean_proximity'].unique().tolist(),
np.arange(len(housing['ocean_proximity'].unique()).tolist())))),
cmap=plt.get_cmap('rainbow'), alpha=0.3, figsize=[16,8], colorbar=True, sharex=False)
plt.axis([0, 16, 0, 550000]);
```



#### Takeaways:

- Correlation is noticeable/strong; imagine drawing an upward line along the density.
- Price cap at \$500, evident from the horizontal line at.
- Other artifacts are evident from lines around 460k, 350k, 280k, 220k, and so on.

Author's suggestion: "You may want to try removing the corresponding districts to prevent your algorithms from learning to reproduce these data quirks."

Remember, we have control over the train set. This may sound sketchy, but whatever we can do (with appropriate justification) to make the data more representative and lead to better generalization in the end is fair game.

## Intuitive Feature Engineering:

"try out various attribute combinations. For example, the total number of rooms in a district is not very useful if you don't know how many households there are. What you really want is the number of rooms per household. Similarly, the total number of bedrooms by itself is not very useful: you probably want to compare it to the number of rooms. And the population per household also seems like an interesting attribute combination to look at."

With ML problems, this sort of feature engineering can matter much more than model choice.



```
In [42]: 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"]
```

```
In [43]: corr_matrix = housing.corr()
corr_matrix["median_house_value"].sort_values(ascending=False)
```

```
Out[43]: 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
```

### Takeaways:

- Houses with a lower bedroom/room ratio tend to be more expensive. Think about this for a moment... larger houses will tend to have allotted a reasonable number of bedrooms, then the remaining 'room budget' will go to leisure rooms.
- Number of rooms per household is more informative than the total number of rooms (larger houses tend to be more expensive).

Remember: Data science is an iterative process. Once you get a prototype up and running, you can analyze the output to gain more insights. Some packages/models may provide you with statistics like p-values (linear regression implementations like OLS in the statsmodels package), or some type of feature importance.

## 4. Prepare the Data for Machine Learning Algorithms

Author's suggestion: Write functions.

- Allows you to reproduce these transformations easily on any dataset.
- Accumulate your own snippets and boilerplate code.
- You can use these functions when deploying your model.
- Easy experimentation.

Remember: Separate the predictors and the labels--don't necessarily want to apply the same transformations to the predictors and the target values.

## Data Cleaning

Most Machine Learning algorithms cannot work with missing features.

Problem: 'total\_bedrooms' attribute has some missing values. There are a few options...

- Get rid of the corresponding districts (drop rows).
- Get rid of the whole attribute (drop column).
- Set the values to some value (fill or impute).

```
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)
```

Note: Usually a good idea to avoid `inplace=True`.

Scikit-Learn's SimpleImputer object:

```
In [113]: from sklearn.impute import SimpleImputer
imputer = SimpleImputer(strategy="median")
```

"Only the total\_bedrooms attribute had missing values, but we cannot be sure that there won't be any missing values in new data after the system goes live, so it is safer to apply the imputer to all the numerical attributes"

Aside: Scikit-Learn Design

A key aspect you'll notice while working with sklearn--Consistency!

- Estimators - Any object that can estimate some parameters based on a dataset
  - will always have a `fit()` method
- Transformers - Estimators (such as an imputer) that can also transform a dataset.
  - will always have `transform()`, `fit_transform()` methods
- Predictors - Estimators that are capable of making predictions on dataset.
  - will always have `predict()`, `score()` methods

## Handling Text and Categorical Attributes

Only one categorical variable: 'ocean\_proximity'.

Check `value_counts()`, `nunique()`.

Need to encode as a numeric type. Why? "ML algorithms prefer to work with numbers."

Scikit-Learn's OrdinalEncoder class:

```
In [125]: from sklearn.preprocessing import OrdinalEncoder

ordinal_encoder = OrdinalEncoder()
housing_cat_encoded = ordinal_encoder.fit_transform(housing_cat)
housing_cat_encoded[:10]

Out[125]: array([[0.],
 [0.],
 [4.],
 [1.],
 [0.],
 [1.],
 [0.],
 [1.],
 [0.],
 [0.]])
```

**Problem:** "One issue with this representation is that ML algorithms will assume that two nearby values are more similar than two distant values."

Scikit-Learn's OneHotEncoder class:

```
In [127]: from sklearn.preprocessing import OneHotEncoder

cat_encoder = OneHotEncoder()
housing_cat_1hot = cat_encoder.fit_transform(housing_cat)
housing_cat_1hot

Out[127]: <16512x5 sparse matrix of type '<class 'numpy.float64'>'
with 16512 stored elements in Compressed Sparse Row format>
```

## Custom Transformers

Not everything is built-in.

```
In [70]: from sklearn.base import BaseEstimator, TransformerMixin

# column index
rooms_ix, bedrooms_ix, population_ix, households_ix = 3, 4, 5, 6

class CombinedAttributesAdder(BaseEstimator, TransformerMixin):
    def __init__(self, add_bedrooms_per_room=True): # no *args or **kwargs
        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):
        rooms_per_household = X[:, rooms_ix] / X[:, households_ix]
        population_per_household = X[:, population_ix] / X[:, households_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)
```

**Note:** 'self' in python is the same thing as 'this' in C++/Java.

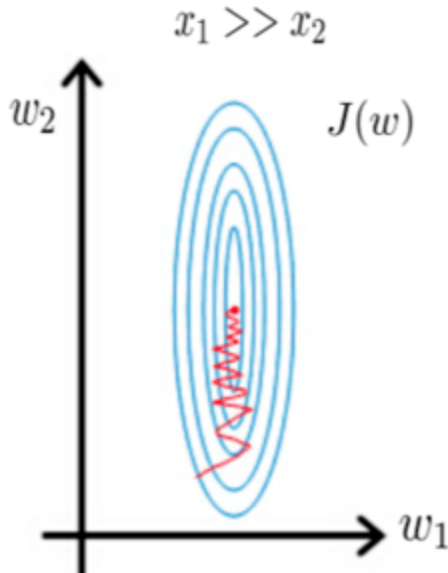
"Scikit-Learn relies on duck typing (not inheritance), all you need to do is create a class and implement three methods: fit() (returning self), transform(), and fit\_transform()."

**Discussion:** Is this really true? The custom class below inherits from BaseEstimator and TransformerMixin...

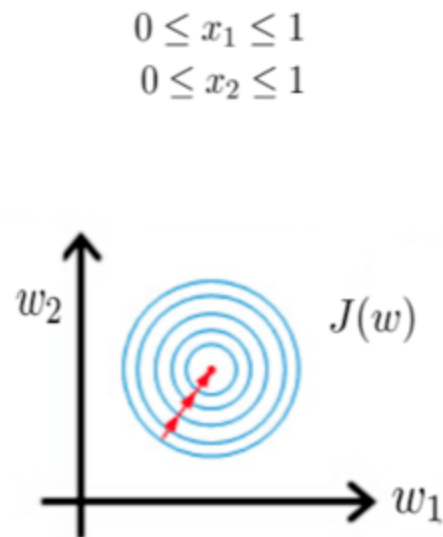
## Feature Scaling

"With few exceptions, Machine Learning algorithms don't perform well when the input numerical attributes have very different scales."

Gradient descent  
without scaling



Gradient descent  
after scaling variables



[https://scikit-learn.org/stable/auto\\_examples/preprocessing/plot\\_all\\_scaling.html#sphx-glr-auto-examples-preprocessing-plot-all-scaling-py](https://scikit-learn.org/stable/auto_examples/preprocessing/plot_all_scaling.html#sphx-glr-auto-examples-preprocessing-plot-all-scaling-py)

Scaling options:

- MinMaxScaler
- StandardScaler
- RobustScaler
- QuantileTransformer
- PowerTransformer

As with all the transformations, fit to training data only.

## Transformation Pipelines

Basically objects that you can compose sequentially.

"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)."

Simple example:

```
In [73]: from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

num_pipeline = Pipeline([
    ('imputer', SimpleImputer(strategy="median")),
    ('attribs_adder', CombinedAttributesAdder()),
    ('std_scaler', StandardScaler()),
])

housing_num_tr = num_pipeline.fit_transform(housing_num)
```

Better example via ColumnTransformer:

```
In [75]: from sklearn.compose import ColumnTransformer

num_attribs = list(housing_num)
cat_attribs = ["ocean_proximity"]

full_pipeline = ColumnTransformer([
    ("num", num_pipeline, num_attribs),
    ("cat", OneHotEncoder(), cat_attribs),
])

housing_prepared = full_pipeline.fit_transform(housing)
```

ColumnTransformer - Applies each transformer to the appropriate columns then concatenates.

FeatureUnion - Applies each transform to the *entire* data set then concatenates.

## 5. Select and Train a Model

Suggestion: Start simple, gradually progress to more complex or computationally expensive models.

LinearRegression()

```
In [83]: from sklearn.linear_model import LinearRegression
```

```
lin_reg = LinearRegression()  
lin_reg.fit(housing_prepared, housing_labels)
```

```
Out[83]: LinearRegression()
```

```
In [84]: # let's try the full preprocessing pipeline on a few training instances
```

```
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: [210644.60459286 317768.80697211 210956.43331178  59218.98886849  
189747.55849879]
```

Compare against the actual values:

```
In [85]: print("Labels:", list(some_labels))
```

```
Labels: [286600.0, 340600.0, 196900.0, 46300.0, 254500.0]
```

Works, but it's pretty bad.

"most districts' median\_housing\_values range between \$120,000 and \$265,000, so a typical prediction error of \$68,628 is not very satisfying. "

What's happening here? Underfitting.

- features do not provide enough information to make good predictions
- that the model is not powerful enough.

## DecisionTreeRegressor()

```
In [89]: from sklearn.tree import DecisionTreeRegressor
```

```
tree_reg = DecisionTreeRegressor(random_state=42)  
tree_reg.fit(housing_prepared, housing_labels)
```

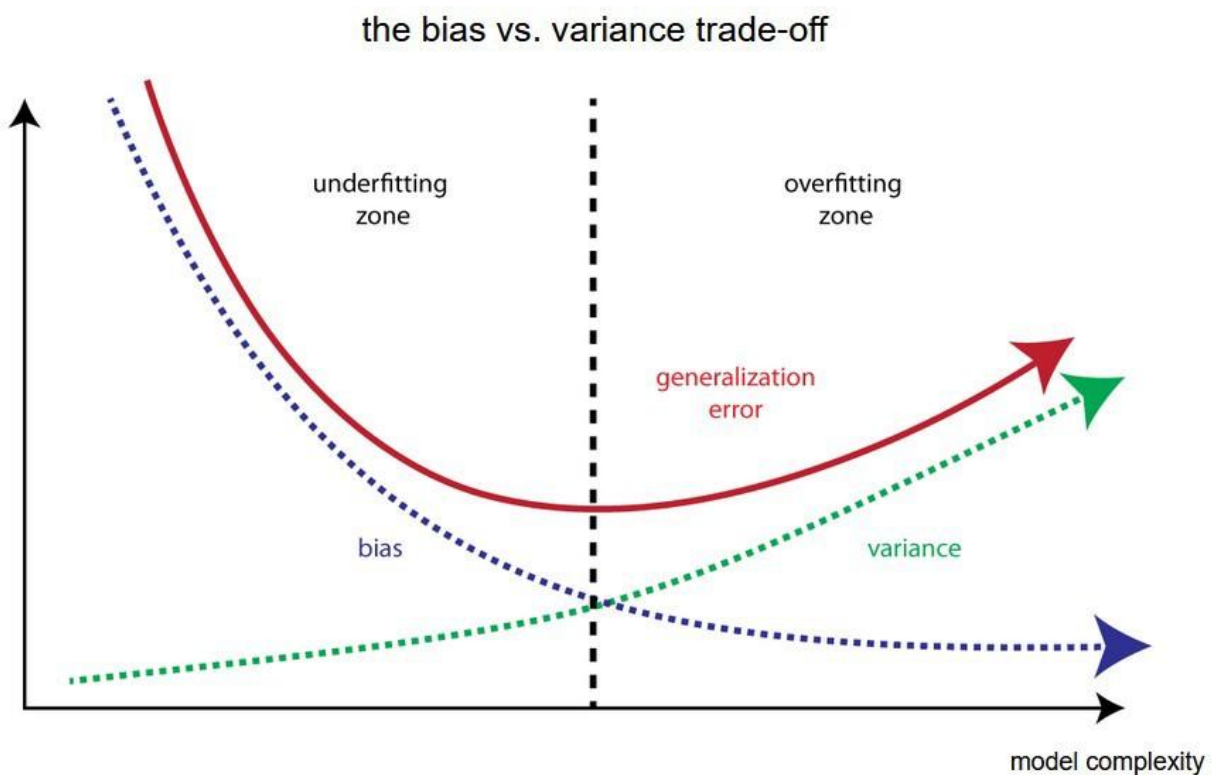
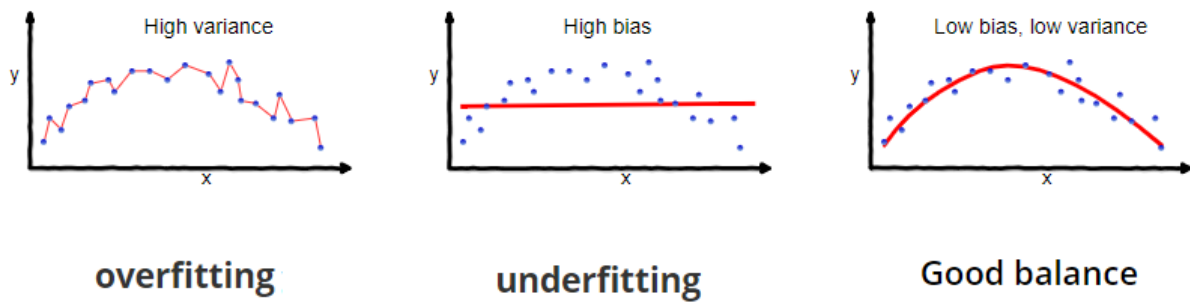
```
Out[89]: DecisionTreeRegressor(random_state=42)
```

```
In [90]: housing_predictions = tree_reg.predict(housing_prepared)  
tree_mse = mean_squared_error(housing_labels, housing_predictions)  
tree_rmse = np.sqrt(tree_mse)  
tree_rmse
```

```
Out[90]: 0.0
```

Weird! Overfitting. What does this mean? The model has effectively memorized all the training examples. If you were to feed it data it hasn't seen before, it would most likely crap the bed.

Recall: Bias-Variance Tradeoff



## Better Evaluation Using Cross-Validation

Rather than `train_test_split()`, we could use Scikit-Learn's K-fold cross-validation class.

Note: Number of folds is somewhat arbitrary; the appropriate value depends on the data.

Scikit-Learn's `cross_val_score` convenience class:

```
In [91]: 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)
```

RandomForestRegressor() - Prototypical example of a "bagging" ensemble model.

```
In [95]: housing_predictions = forest_reg.predict(housing_prepared)
forest_mse = mean_squared_error(housing_labels, housing_predictions)
forest_rmse = np.sqrt(forest_mse)
forest_rmse
```

```
Out[95]: 18603.515021376355
```

```
In [96]: from sklearn.model_selection import cross_val_score

forest_scores = cross_val_score(forest_reg, housing_prepared, housing_labels, scoring="neg_mean_squared_error", cv=10)
forest_rmse_scores = np.sqrt(-forest_scores)
display_scores(forest_rmse_scores)

Scores: [49519.80364233 47461.9115823 50029.02762854 52325.28068953
 49308.39426421 53446.37892622 48634.8036574 47585.73832311
 53490.10699751 50021.5852922 ]
Mean: 50182.303100336096
Standard deviation: 2097.0810550985693
```

Author's tip: Saving models via pickle.

- Warning: pickling and similar data serialization routines can introduce vulnerabilities in your code!
- joblib is supposed to be the replacement for pickle; more efficient for ndarrays.
- Another, less flexible, but more space-efficient: keep the validation setup fixed, and for each model store the parameters (in case you need to re-train it), and oof\_predictions, in case you want to use that output down the line for further ensembling (blending/stacking, weighted averages of models, etc).

## 6. Fine-Tune Your Model

### Grid Search

Evaluate all the possible combinations of hyperparameter values.

```
In [99]: from sklearn.model_selection import GridSearchCV

param_grid = [
    # try 12 (3x4) combinations of hyperparameters
    {'n_estimators': [3, 10, 30], 'max_features': [2, 4, 6, 8]},
    # then try 6 (2x3) combinations with bootstrap set as False
    {'bootstrap': [False], 'n_estimators': [3, 10], 'max_features': [2, 3, 4]},
]

forest_reg = RandomForestRegressor(random_state=42)
# train across 5 folds, that's a total of (12+6)*5=90 rounds of training
grid_search = GridSearchCV(forest_reg, param_grid, cv=5,
                           scoring='neg_mean_squared_error',
                           return_train_score=True)
grid_search.fit(housing_prepared, housing_labels)

Out[99]: GridSearchCV(cv=5, estimator=RandomForestRegressor(random_state=42),
  param_grid=[{'max_features': [2, 4, 6, 8],
    'n_estimators': [3, 10, 30]},
    {'bootstrap': [False], 'max_features': [2, 3, 4],
    'n_estimators': [3, 10]}],
  return_train_score=True, scoring='neg_mean_squared_error')
```



### Key attributes:

- .best\_params\_
- .best\_estimator\_
- .cv\_results\_

Author's tip: "When you have no idea what value a hyperparameter should have, a simple approach is to try out consecutive powers of 10." a.k.a. logspace.

Note: Once you become familiar with particular machine learning algorithms, you'll better be able to tell what parameter values are odd... good to sanity check hyperparameter optimization.

Tip: Setting `n_jobs=-1` is handy for many sklearn objects.

- -1 means using all processors.
- Or, `n_jobs=n_cpus - 1`, in order to avoid that the machine gets stuck.

## Randomized Search

Preferable when hyperparameter search space is very large.

```
In [104]: from sklearn.model_selection import RandomizedSearchCV
          from scipy.stats import randint

          param_distributions = {
              'n_estimators': randint(low=1, high=200),
              'max_features': randint(low=1, high=8),
          }

          forest_reg = RandomForestRegressor(random_state=42)
          rnd_search = RandomizedSearchCV(forest_reg, param_distributions=param_distributions,
                                         n_iter=10, cv=5, scoring='neg_mean_squared_error', random_state=42)
          rnd_search.fit(housing_prepared, housing_labels)

Out[104]: RandomizedSearchCV(cv=5, estimator=RandomForestRegressor(random_state=42),
                             param_distributions={'max_features': <scipy.stats._distn_infrastructure.rv_frozen object at 0x7fd1b96682d0>,
                                                  'n_estimators': <scipy.stats._distn_infrastructure.rv_frozen object at 0x7fd1b9668b10>},
                             random_state=42, scoring='neg_mean_squared_error')
```

### Main benefits:

- May yield a good configuration of hyperparameters in less time/compute than a really extensive gridsearch.
- Some marginal control over computing budget via `n_iter`.

### Other Hip/Trendy Options (via external packages):

- hyperopt
- optuna

2.2.2 LGBM Study 2; max\_depth\_upper=8, min\_data\_in\_leaf\_lower=150

```
In [35]: DEBUG_MODE = False
MAX_TIMEOUT = 4*60*60
lgbm_study_2_8_150 = optuna.create_study(direction='minimize')
lgbm_study_2_8_150.optimize(lambda trial: objective_lgbm_v1(trial,
    X_train_quantiled,
    y_train_unscaled,
    n_folds=N_FOLDS,
    random_seed=RANDOM_SEED,
    static_params=lgbm_static_params_base,
    max_depth_upper=8,
    min_data_in_leaf_lower=150,)),
    n_trials=10 if DEBUG_MODE else None,
    timeout=None if DEBUG_MODE else MAX_TIMEOUT,
    show_progress_bar=False,
    callbacks=[logging_callback_best],)

...

In [36]: print(lgbm_study_2_8_150.best_value)
print(lgbm_study_2_8_150.best_trial.params)

7.8378488771033785
{'learning_rate': 0.0035864329699057187, 'max_depth': 8, 'num_leaves': 173, 'lambda_l1': 4.926374723265517e-07, 'lambda_l2': 0.011399693220695693, 'max_bin': 132, 'bagging_fraction': 0.5131388059904235, 'bagging_freq': 5, 'feature_fraction': 0.14867619007859442, 'min_data_in_leaf': 754, 'extra_trees': False, 'min_sum_hessian_in_leaf': 0.006947644542162502}
```

## Ensemble Methods

"Another way to fine-tune your system is to try to combine the models that perform best... especially if the individual models make very different types of errors."

Common types of ensembling:

- Bagging or weighted averages
- Boosting
- Stacking/Blending

## Analyze the Best Models and Their Errors

grid\_search.best\_estimator\_.feature\_importances\_

Illustrates the importance of diagnostic models!

Further options for improvement, which basically amount to removing noise:

- Drop bad features
- Remove outliers

Try to find patterns in the errors. If you can spot a pattern to exploit, chances are you can finagle with the features and hyperparameters so that the model can exploit the pattern too.

"You should also look at the specific errors that your system makes, then try to understand why it makes them and what could fix the problem (adding extra features or getting rid of uninformative ones, cleaning up outliers, etc.)."

## Evaluate Your System on the Test Set

"Run your full\_pipeline to transform the data (call transform(), not fit\_transform()—you do not want to fit the test set!), and evaluate the final model on the test set"

```
In [108]: 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)
```

```
In [109]: final_rmse
```

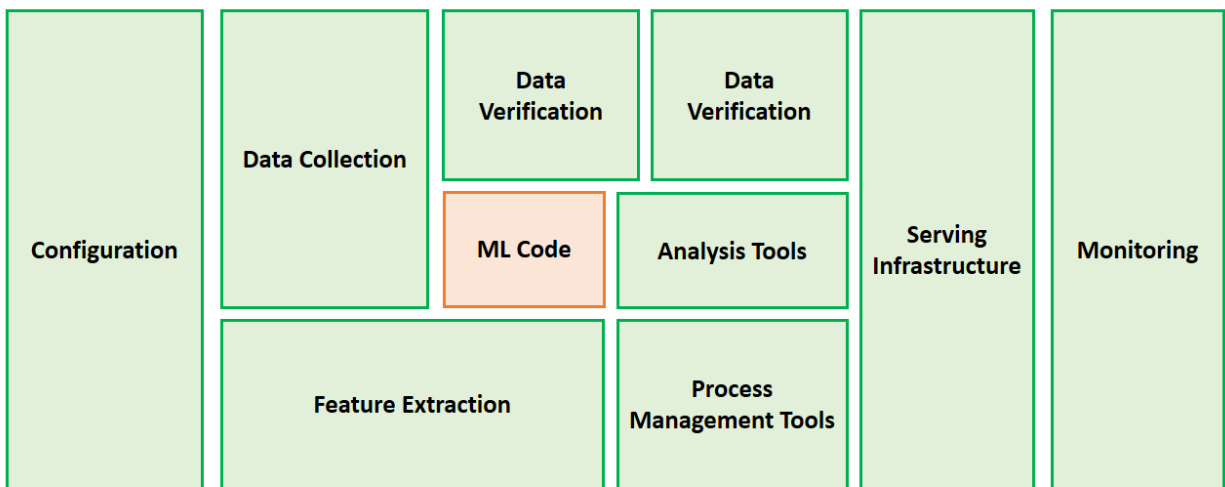
```
Out[109]: 47730.22690385927
```

## 8. Launch, Monitor, and Maintain Your System

Probably better saved for the dedicated chapters with fleshed-out examples.

Hojillions of options of varying complexity; on-prem vs cloud, managed services vs not so managed.

The Reality (Andrew Ng's Coursera stuff):



Monica Rogati (<https://hackernoon.com/the-ai-hierarchy-of-needs-18f111fcc007>)

# THE DATA SCIENCE HIERARCHY OF NEEDS

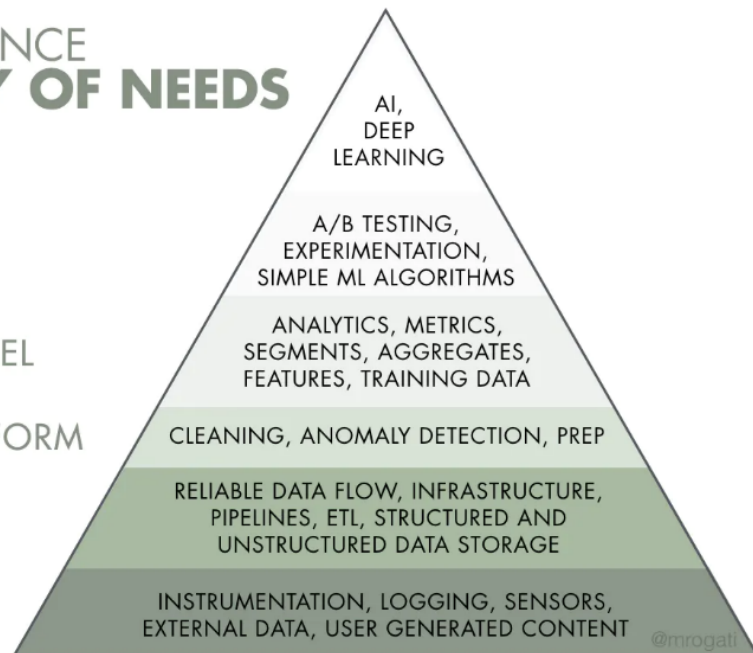
LEARN/OPTIMIZE

AGGREGATE/LABEL

EXPLORE/TRANSFORM

MOVE/STORE

COLLECT



"Data-Centric AI" sentiment periodically resurfaces--or maybe it never really went anywhere?  
(Andrew Ng's Coursera stuff, yet again):

	Steel defect detection	Solar panel	Surface inspection
Baseline	76.2%	75.68%	85.05%
Model-centric	+0% (76.2%)	+0.04% (75.72%)	+0.00% (85.05%)
Data-centric	+16.9% (93.1%)	+3.06% (78.74%)	+0.4% (85.45%)

## (What happened to... ) 7.Present Your Solution

- What you have learned.
- What worked, what did not.
- What assumptions were made.
- What your system's limitations are.
- Document everything!!!
- Create nice presentations with clear visualizations and easy- to-remember statements (e.g., "the median income is the number one predictor of housing prices").

Conclusion: "In this California housing example, the final performance of the system is not better than the experts' price estimates, which were often off by about 20%, but it may still be a good idea to launch it, especially if this frees up some time for the experts so they can work on more interesting and productive tasks."

### Things that don't work:

- Emailing your tech-illiterate CEO a ghastly, unformatted MS Excel spreadsheet with lots of numbers.
- Skipping the presentation all together, appealing to your authority in all things data: "Just trust me, bro".
- Getting distracted by other fires, subsequently neglecting documentation. Ideally your notebooks and code should be at least *somewhat* self-documenting.