Chapter 2: Data handling and Regression models

Werner Bauer

UNIVERSITY OF SURREY

Outline

Loading data files with Pandas

Pandas provides useful functions to import various file types:

- .csv (comma-separated values) file

```
data = pd.read_csv('titanic.csv')
# Output:
     PassengerId  Survived  Pclass  ...     Fare Cabin  Embarked
0              1         0       3  ...   7.2500   NaN         S
1              2         1       1  ...  71.2833   C85         C
2              3         1       3  ...   7.9250   NaN         S
3              4         1       1  ...  53.1000  C123         S
4              5         0       3  ...   8.0500   NaN         S
```

- .txt files (without column names and separated by spaces):

```
data = pd.read_csv('file.txt', sep = "\s+", names = ['column1_name','column2_name']) # use '\s+'
       if spacing is not known, assign names to columns, ....)
```

- excel (.xlsx) files

```
banking = pd.read_excel('banking.xlsx', sheet_name='bank-additional-full')
# Output:
       age         job  marital  ... euribor3m nr_employed   y
0       56   housemaid  married  ...     4.857      5191.0  no
1       57    services  married  ...     4.857      5191.0  no
2       37    services  married  ...     4.857      5191.0  no
```

- Loading files from internet sources via url's

Example: loading the housing.txt file (used later)

We want to read in the following txt file:

```
0.00632  18.00   2.310  0  0.5380  6.5750  65.20  4.0900   1  296.0  15.30 396.90   4.98  24.00
0.02731   0.00   7.070  0  0.4690  6.4210  78.90  4.9671   2  242.0  17.80 396.90   9.14  21.60
0.02729   0.00   7.070  0  0.4690  7.1850  61.10  4.9671   2  242.0  17.80 392.83   4.03  34.70
0.03237   0.00   2.180  0  0.4580  6.9980  45.80  6.0622   3  222.0  18.70 394.63   2.94  33.40
   .        .      .    .    .       .      .       .     .   .       .     .       .     .
   .        .      .    .    .       .      .       .     .   .       .     .       .     .
```

We can use `.read_csv()` with options as shown next:

```
# as file has no header and values separated by whitespaces
df = pd.read_csv('housing.txt', delimiter= '\s+', header = None) # OR
df = pd.read_csv('housing.txt', delim_whitespace=True, header = None) # OR
df = pd.read_fwf('housing.txt', header = None) # fwf for 'fixed-width formatted'
# Output:
          0     1      2   3      4   ...     9     10      11    12    13
0    0.00632  18.0   2.31    0  0.538  ...  296.0  15.3  396.90  4.98  24.0
1    0.02731   0.0   7.07    0  0.469  ...  242.0  17.8  396.90  9.14  21.6

# To set names rather then indices for the COLUMNS:
df.columns = ['CRIM','ZN','INDUS','CHAS','NOX','RM','AGE','DIS','RAD','TAX','PTRATIO','B','LSTAT', 'MEDV'] # OR
df = pd.read_csv('housing.txt', delim_whitespace=True, header = None, names = ['CRIM',..., 'MEDV'])
# Output:
        CRIM    ZN  INDUS  CHAS    NOX  ...    TAX  PTRATIO       B  LSTAT  MEDV
0    0.00632  18.0   2.31     0  0.538  ...  296.0     15.3  396.90   4.98  24.0
1    0.02731   0.0   7.07     0  0.469  ...  242.0     17.8  396.90   9.14  21.6
```

Example (Hands-On ML book 46-47): loading (another) housing data

- We write functions to download and decompress the file:

```python
import os
import tarfile
import urllib
import pandas as pd

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

def fetch_housing_data(housing_url=HOUSING_URL, housing_path=HOUSING_PATH):
    os.makedirs(housing_path, exist_ok=True)
    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()

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

fetch_housing_data() # fetch the data
housing = load_housing_data()
```

-

```
housing.head() # Outputs top 5 lines:
   longitude  latitude  ...  median_house_value  ocean_proximity
0    -122.23     37.88  ...            452600.0         NEAR BAY
1    -122.22     37.86  ...            358500.0         NEAR BAY
2    -122.24     37.85  ...            352100.0         NEAR BAY
3    -122.25     37.85  ...            341300.0         NEAR BAY
4    -122.25     37.85  ...            342200.0         NEAR BAY
```

Types are not aways correctly recognised:

```
print(titanic) # Output:
     PassengerId  Survived  Pclass  ...     Fare Cabin  Embarked
0              1         0       3  ...   7.2500   NaN         S
1              2         1       1  ...  71.2833   C85         C
2              3         1       3  ...   7.9250   NaN         S
print(titanic.dtypes) # Output:
PassengerId      int64 # OK
Survived         int64 # should be categorical
...

titanic = titanic.astype({'Survived' : 'category'}, copy=True) # DEFAULT: copy=True
print(titanic.dtypes) # Output:
PassengerId       int64 # OK
Survived       category # OK
```

```
print(housing)
       longitude  latitude  ...  median_house_value  ocean_proximity
0        -122.23     37.88  ...            452600.0         NEAR BAY
1        -122.22     37.86  ...            358500.0         NEAR BAY
2        -122.24     37.85  ...            352100.0         NEAR BAY
print(housing.dtypes)
longitude             float64 # OK
...
ocean_proximity         object # Could be any Python object, but repetitive entries suggest it is
      categorical!
housing['ocean_proximity'].value_counts()
<1H OCEAN     9136
INLAND        6551
NEAR OCEAN    2658
NEAR BAY      2290
ISLAND           5
# THEN: modify type as suggested above! Discussed later again!
```

Looking at the data file (housing)

- Use `.head()` to look at top 5 lines
- `.describe()` gives overview of each numerical attribute

```
          longitude      latitude  ...  median_income  median_house_value
count  20640.000000  20640.000000  ...   20640.000000        20640.000000
mean    -119.569704     35.631861  ...       3.870671       206855.816909
std        2.003532      2.135952  ...       1.899822       115395.615874
min     -124.350000     32.540000  ...       0.499900        14999.000000
25%     -121.800000     33.930000  ...       2.563400       119600.000000
50%     -118.490000     34.260000  ...       3.534800       179700.000000
75%     -118.010000     37.710000  ...       4.743250       264725.000000
max     -114.310000     41.950000  ...      15.000100       500001.000000
```

- count, mean, min, max are self-explanatory
- std stands for (standard deviation)
- 25%, 50%, 75% for percentile (e.g. 25% of the districts have a `median_income` lower than 2.563400)

Looking at the data file (housing)

- The `.info()` method gives quick description of data

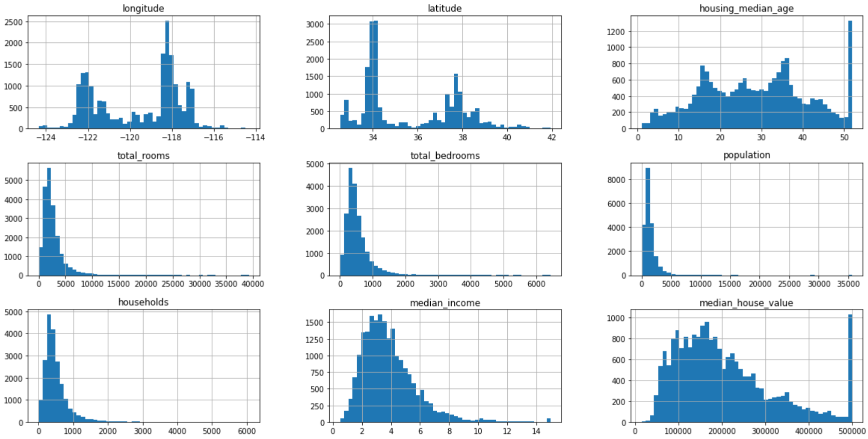
```
housing = load_housing_data()
print(housing.info()) # Output:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 10 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   longitude           20640 non-null  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
None
```

Gives total number of rows and columns, the attribute types and number of non-null values.

- Note: `total_bedrooms` is missing some values (we will need to address this)

- Using a histogram plot to view numerical attributes:

```
import matplotlib.pyplot as plt
housing.hist(bins=50,figsize=(20,15))
```



At this stage is is time to set aside part of the data. Why?

- Having test data is important to avoid overfitting of the ML model
- If your brain detects pattern when looking at the test data, you might realize pattern that influences your choice of ML model
- When you estimate the generalization error using the test set, your estimate will be too optimistic and it does not generalize this well to new data (data snooping bias)

We can use a scikit-learn function to split datasets.

- ```python
  from sklearn.model_selection import train_test_split

  # purely random sampling method:
  train_set, test_set = train_test_split(housing,test_size=0.2,random_state=42)
  ```

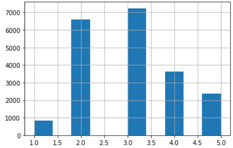
- Potentially with smaller datasets, random sampling introduces bias! Hence:
  we must make sure that the test set is representitive for the various categories. E.g. consider the `median_income` attribute:

```python
housing["income_cat"] = pd.cut(housing["median_income"],
                               bins=[0., 1.5, 3.0, 4.5, 6., np.inf],
                               labels=[1, 2, 3, 4, 5])

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]

# remove income_cut to bring data back to original state:
for set_ in (strat_train_set, strat_test_set):
    set_.drop("income_cat", axis=1, inplace=True)
```



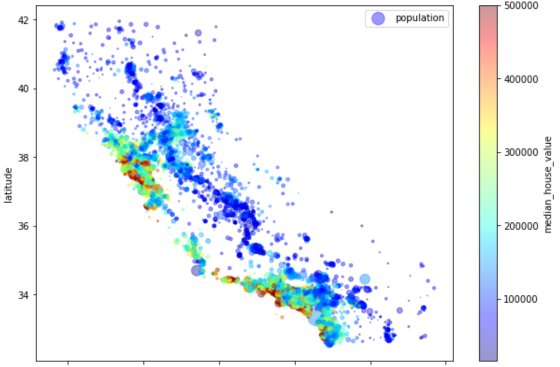

```python
??_train_set['income_cat'].
value_counts()/len(??_train_set)
```

	Overall	Stratified	Random	Rand. %error	Strat. %error
1	0.039826	0.039729	0.040213	0.973236	-0.243309
2	0.318847	0.318798	0.324370	1.732260	-0.015195
3	0.350581	0.350533	0.358527	2.266446	-0.013820
4	0.176308	0.176357	0.167393	-5.056334	0.027480
5	0.114438	0.114583	0.109496	-4.318374	0.127011
```

Outline

Example: scatter plot of housing dataset

```
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()
```



California housing prices: red is expensive, blue is cheap;
large circles for areas with large population

- Housing prices are related very much to location!

Looking for correlations: *standard correlation coefficient*

- If dataset is not too large: calculate standard correlation coefficient (Pearson's r):

```
corr_matrix = housing.corr()
print(corr_matrix) # Output:
                    longitude  latitude  ...  median_income  median_house_value
longitude            1.000000 -0.924478  ...      -0.019615           -0.047466
latitude            -0.924478  1.000000  ...      -0.075146           -0.142673
housing_median_age  -0.105823  0.005737  ...      -0.111315            0.114146
...
households           0.063146 -0.077765  ...       0.010869            0.064590
median_income       -0.019615 -0.075146  ...       1.000000            0.687151
median_house_value  -0.047466 -0.142673  ...       0.687151            1.000000
```

- select an attribute of interest and sort values:

```
corr_matrix['median_house_value'].sort_values(ascending=False)
# Output:
median_house_value    1.000000
median_income         0.687151
total_rooms           0.135140
housing_median_age    0.114146
households            0.064590
total_bedrooms        0.047781
population           -0.026882
longitude            -0.047466
latitude             -0.142673
```

- Range [-1,1]: -1 negative, +1 positive, 0 no correlation
- works only for lin. correlation: see examples in Hands-On ML bookpage 59

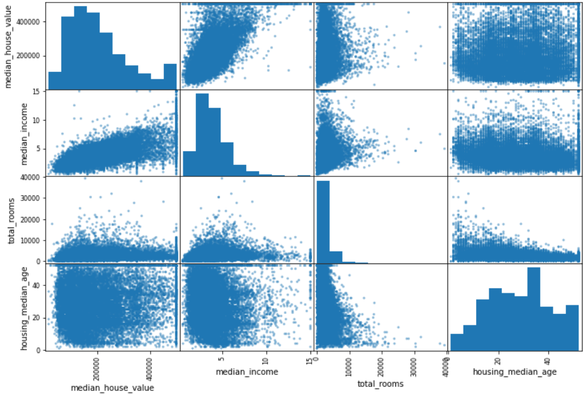
Looking for correlations: pandas *scatter matrix*

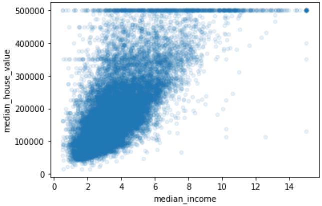
- Plot attributes most correlated to `median_house_value`

```python
from pandas.plotting import scatter_matrix

attributes = ["median_house_value", "median_income", "total_rooms", "housing_median_age"]
scatter_matrix(housing[attributes], figsize=(12, 8)) # LEFT

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





- Note: diagonal shows histograms rather than correlation

Testing out attribute combinations

- Try out some attribute combinations to increase correlation with the target attribute, here `median_house_value`.
- Example: total number of rooms in a district not very useful without knowing how many households there are!
- Following attribute combinations seem promising:

```
housing["rooms_per_household"] = housing["total_rooms"]/housing["households"]
housing["bedrooms_per_room"] = housing["total_bedrooms"]/housing["total_rooms"]
housing["population_per_household"]=housing["population"]/housing["households"]

corr_matrix = housing.corr()
corr_matrix["median_house_value"].sort_values(ascending=False)
# Output:
median_house_value          1.000000
median_income               0.687151
rooms_per_household         0.146255
total_rooms                 0.135140
housing_median_age          0.114146
households                  0.064590
total_bedrooms              0.047781
population_per_household   -0.021991
population                 -0.026882
longitude                  -0.047466
latitude                   -0.142673
bedrooms_per_room          -0.259952
```

We found via `bedrooms_per_room`: houses with lower bedroom to room ratio are more expensive!

Outline

Steps to prepare data for Machine Learning (ML)

- Most ML algorithms cannot work with missing values!
- Most ML prefer to work with numerical rather than categorical attributes!
- Most ML don't perform well when the input numerical attributes have very different scale!

$\Rightarrow$ usually we must perform preparation steps:

1. Split labels from predictors
2. Data Cleaning; either
    - get rid of the corresponding observations/instances (rows),
    - get rid of whole attributes (columns), or
    - set the values to some value (zero, mean, median ...)
3. Converting categories from text to numbers (e.g. ordinal encoder, one-hot encoder)
4. Feature scaling (min-max scaling or standardisation)

Step 1: Split labels from predictors

1. Labels: these are the target values, the ones we want to actually predict
2. Predictors: all other attributes from the dataset used for the prediction

```
housing = strat_train_set.drop("median_house_value", axis=1)
# drop() creates a copy of the data and does not affect strat_train_set
housing_labels = strat_train_set["median_house_value"].copy()
```

Note:

- we split target value from predictors which will allow us to perform different transformations to them
- we start with clean training set by copying `strat_train_set`

Step 2: Data cleaning

1. We can use DataFrame methods `dropna()`,`drop()`,`fillna()`:

```
housing.dropna(subset = ['total_bedrooms']) # drops instances (rows)
housing.drop(['total_bedrooms'],axis = 1) # drops whole attribute 'total_bedrooms'
median = housing['total_bedrooms'].median() # define median, used later also for test set
housing['total_bedrooms'].fillna(value = median, inplace=True) # fill NaNs with column median
```

2. For automatisation of this process, `sklearn.impute` provides the `SimpleImputer` class:

```
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(strategy='median') # create instance and define strategy

# split of category column as imputer only works for numerical values:
# we take care of category attribute later:
housing_num = housing.drop("ocean_proximity", axis=1)

imputer.fit(housing_num)

print(imputer.statistics_) # gives same output as print(housing_num.median().values)
# [-118.51      34.26      29.       2119.       433.       1164.       408.         3.54155]

# NOTE: X is plain NumpyArray with transformed features:
X = imputer.transform(housing_num)

# To obtain DataFrame do:
housing_tr = pd.DataFrame(X, columns=housing_num.columns,
                          index=housing_num.index)
```

1. Separating categorical attributes from others:

```
housing_cat = housing[['ocean_proximity']] # gives DataFrame with column name 'ocean_proximity';
       interior bracket for 'list', outside bracket is indexing operators.
# Note: [ ] returns Series, [[ ]]  returns DataFrame
print(housing_cat.head(6))
      ocean_proximity
12655          INLAND
15502      NEAR OCEAN
2908           INLAND
14053      NEAR OCEAN
20496       <1H OCEAN
1481         NEAR BAY
```

2. Ordinal encoder: useful when nearby values are closer related than distinct ones (e.g. 'bad', 'average', 'good')

```
from sklearn.preprocessing import OrdinalEncoder

ordinal_encoder = OrdinalEncoder() # create instance
housing_cat_encoded = ordinal_encoder.fit_transform(housing_cat)
print(housing_cat_encoded[:6]) # use numpy indexing here!
[[1.]
 [4.]
 [1.]
 [4.]
 [0.]
 [3.]]

# Output corresponding categories with:
print(ordinal_encoder.categories_) # Output:
[array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'],
      dtype=object)]
```

1. If there is not such ordering (as in `ocean_proximity`), one-hot encoder could be used:

```
from sklearn.preprocessing import OneHotEncoder

cat_encoder = OneHotEncoder() # create instance
housing_cat_1hot = cat_encoder.fit_transform(housing_cat) # creates sparse matrix

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

print(housing_cat_1hot) # Output (only first 3 lines):
(0, 1)  1.0
(1, 4)  1.0
(2, 1)  1.0
...

housing_cat_1hot.toarray() # sparse to dense mapping
# Output:
array([[0., 1., 0., 0., 0.],
       [0., 0., 0., 0., 1.],
       [0., 1., 0., 0., 0.],
       ...,
       [1., 0., 0., 0., 0.],
       [1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.]])

# get list of categories with .categories_
cat_encoder.categories_
# Output:
[array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'],
       dtype=object)]
```

Step 4: Feature Scaling

1. Min-max scaling: values are shifted and rescaled to range from 0 to 1 (outliers have strong impact on this scaling)

```
from sklearn.preprocessing import MinMaxScaler

minmax_scale = MinMaxScaler()
housing_minmax =  minmax_scale.fit_transform(housing_num) # normalizes data

print(housing_minmax)
# Output:
[[0.28784861 0.63549416 0.54901961 ... 0.06261386 0.13144137 0.11542599]
 [0.70916335 0.05844846 0.11764706 ... 0.05639172 0.14301718 0.40257376]
 ...
```

2. Standardization: subtracts the mean value (standardized values have zero mean) and divides by standard deviation (resulting distribution has unit variance). Here, the data is much less affected by outliers!

```
from sklearn.preprocessing import StandardScaler

std_scale = StandardScaler()
housing_std =  std_scale.fit_transform(housing_num) # does the standardization

print(housing_std)
# Output:
[[-0.94135046  1.34743822  0.02756357 ...  0.73260236  0.55628602
  -0.8936472 ]
 [ 1.17178212 -1.19243966 -1.72201763 ...  0.53361152  0.72131799
   1.292168  ]
```

Step 5: Custom Transformers

Scikit-Learn Design (extract), cf. Hands-On ML book page 64-65

- Estimators (`.fit()`): can estimate parameters (e.g. Imputer)
- Transformers (`.transform()`): applies estimates to data
- Predictors (`.predict()`): e.g. linear regression model (below)

1. Combined attributes need a Custom transformer
2. Transformers need: `fit()`, `transform()`, `fit_transform()`

```python
from sklearn.base import BaseEstimator, TransformerMixin

rooms_ix, bedrooms_ix, population_ix, households_ix = 3, 4, 5, 6 # indices for attribute columns

class CombinedAttributesAdder (BaseEstimator, TransformerMixin): # set up customer transformers:
    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):
        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]
    # fit_transform() method is inherited from base class TransformerMixin
```

A 'Transformation pipeline' summarizes these steps!

We want to automatise the above steps. Scikit-Learn provides the classes `Pipeline, ColumnTransformer` to do this.

1. We first define the numerical pipeline:

```
from sklearn.pipeline import Pipeline

# pipeline takes list of tupes
num_pipeline = Pipeline([
    ('imputer', SimpleImputer(strategy='median')),
    ('attribs_adder',CombinedAttributesAdder(add_bedrooms_per_room=True)),
    ('std_scaler', StandardScaler()), ])

# if one wants to transform the numerical values only:
housing_num_tr = num_pipeline.fit_transform(housing_num)
```

2. Next, we combine it with the category attributes:

```
from sklearn.compose import ColumnTransformer

num_attributs = list(housing_num) # give list of column names
cat_attributs = ['ocean_proximity'] # define category names

full_pipeline = ColumnTransformer([('num', num_pipeline, num_attributs),
    ('cat', OneHotEncoder(), cat_attributs), ])
```

3. All at once: applying `full_pipeline` directly to `housing`

```
housing_prepared = full_pipeline.fit_transform(housing) #  this is numpy array!!!
```

The 'Transformation pipeline' on one page!

```python
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OneHotEncoder
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer

housing = strat_train_set.drop("median_house_value", axis=1) # separate target from predictors
housing_labels = strat_train_set["median_house_value"].copy()
housing_num = housing.drop("ocean_proximity", axis=1) # drop category attributes:

rooms_ix, bedrooms_ix, population_ix, households_ix = 3, 4, 5, 6 # indices for attribute columns
class CombinedAttributesAdder (BaseEstimator, TransformerMixin): # set up customer transformers:
    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):
        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]

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

num_attributs = list(housing_num) # give list of column names
cat_attributs = ['ocean_proximity'] # define category names
full_pipeline = ColumnTransformer([('num', num_pipeline, num_attributs),
    ('cat', OneHotEncoder(), cat_attributs),])

housing_prepared = full_pipeline.fit_transform(housing) # full transformation as numpy array
```

Outline

Train a linear regression (LR) model!

1. Training a ML model is now comparable easy:

```
# we load a linear regression model:
from sklearn.linear_model import LinearRegression

lin_reg = LinearRegression() # create instance
lin_reg.fit(housing_prepared, house_labels) # train the model
```

2. Make predictions with `.predict()`:

```
# 1): select say the first 6 istances:
some_instances = housing.iloc[:5]
some_labels = housing_labels.iloc[:5]

# apply transform, but no fit (fit always only to training set)!
some_instances_prep = full_pipeline.transform(some_instances)
predictions = lin_reg.predict(some_instances_prep)

print('Predictions:',  predictions)
Predictions: [ 85658. 305493. 152056. 186096. 244551.]
print('Labels:',  list(some_labels))
Labels: [72100.0, 279600.0, 82700.0, 112500.0, 238300.0]
# NOTE: there are errors larger than 30 %
```

About 30 % off doesn't look too good! **The LR model is underfitting the training data!**

Evaluation: RMSE and cross-validation!

- Error evaluation via root-mean-square-error (rmse):

```
from sklearn.metrics import mean_squared_error
housing_predictions = lin_reg.predict(housing_prepared)
lin_rmse = np.sqrt(mean_squared_error(housing_labels,housing_predictions))
print(lin_rmse)
# Output:
68627.87390018745
```

More information with k-fold Cross-validation (CV):

Training data randomly split into k distinct subsets (folds): CV outputs k eval. scores by picking 1 different fold for evaluation every time while using the other $k-1$ folds for training

```
from sklearn.model_selection import cross_val_score
def display_scores(scores):
    print ("Scores:", scores)
    print ("Mean:", scores.mean())
    print ("Standard deviation:", scores.std())

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.round()) # Output:
Scores: [71763. 64115. 67771. 68635. 66846. 72528. 73997. 68802. 66443. 70140.]
Mean: 69104.0
Standard deviation: 2880.389938879804
```

Use a more complex model: Decision Tree (DT)

- **DT** more powerful as it allows for complex nonlinear relationships

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

# We can evalute this trained model on the training set:
housing_predictions = tree_reg.predict(housing_prepared)
tree_rmse = np.sqrt(mean_squared_error(housing_labels, housing_predictions))
tree_rmse
# Output:
0.0
```

- Evalute DT with cross-validation

```
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)

display_scores(tree_rmse_scores.round())
# Output:
Scores: [72926. 71205. 67809. 71970. 70258. 78120. 69846. 74363. 68414. 71595.]
Mean: 71650.6
Standard deviation: 2856.0453147665567
```

Score on training set much lower that on evaluation set: **DT is strongly overfitting the training set!**

Underfitting or overfitting the training set

Reasons for models underfitting training data

- model not powerful enough
- features in data do not provide enough information
- constraints on model (e.g. via regularization terms) to reduce numbers of parameters

Reasons for overfitting (and potential remedies)

- model to complex (too many parameters) for the information provided (maybe regularize it or take simpler one)
- model has been trained too extensively (early stopping)
- to little data (if possible use more training data)

Some metrics for Regression models!

Denote predictions with y_i, true value with x_i, and the number of data points with n.

- Mean absolute error (MAE):

$$MAE = \frac{1}{n} \sum_i |y_i - x_i|.$$

  The MAE is always positive and similar to the RMSE, but less sensitive to outliers. *Look for smaller values of the MAE.*

- Mean squared error (MSE):

$$MSE = \frac{1}{n} \sum_i (y_i - x_i)^2.$$

  The MSE is the square of the RMSE. *Look for smaller values of the MSE.*

- Root mean squared error (RMSE);

$$RMSE = \sqrt{MSE}.$$

  The RMSE is always positive and its units match the units of your response. *Look for smaller values of the RMSE.*