Lesson 04
Notes on Chapter 2
End-to-End Machine Learning Project

CSC357 Advanced Topics—Machine Learning

14 January 2020

Chapter 1

Notes

- os.mkdirs()
- os.path.join()
- matplotlib.pyplot.savefig()
- matplotlib.pyplot.tight_layout()

- what is my organization's objective?
- model is probably not the end goal
- how will model be used? what benefit from use of model?
- framing the problem will help us answer questions...
  - which algorithms?
  - which measures of performance?
  - how much effort to invest in fine-tuning?
- our model's output → prediction of district's median housing price
- will be of one of several inputs to another system whose output will be a recommendation to invest/not invest
- our model is one component of a data pipeline
- components of pipeline self-contained, independent, communicate through well-defined interfaces, run asychronously
- easier to build and maintain a system that is modularized
- how are we currently solving the problem?

  - by hand
  - using complex rules (to estimate prices when data unavailable)
  - takes a long time
  - less accuracy than we want and think that we can achieve (20% errors)
- input for our model—census data
- Census Bureau's block groups
  - smallest geographical unit for which Bureau publishes data
  - 600–3000 people (typically)
  - we will call them districts
  - population, median income, median housing price, and other features
- nature of our problem suggests...
  - supervised learning (Census data includes housing prices—our label)
  - regression (we are trying to predict prices)
  - multiple regression (we may use several features of a district to predict prices)
  - univariate regression (we are only trying to predict one thing—price)
  - batch learning (no continuous stream of data or frequent updates of data)
- measurements of performance: Root Mean Square Error, Mean Absolute Error

$$\text{RMSE}(\mathbf{X}, h) = \sqrt{\frac{1}{m}\sum_{i=0}^{m-1}(h(\mathbf{x}^i) - y^i)^2}$$

$$\text{MAE}(\mathbf{X}, h) = \frac{1}{m}\sum_{i=0}^{m-1} \mid h(\mathbf{x}^i) - y^i \mid$$

- RMSE is more sensitive to outliers than is MAE
- RMSE uses ℓ_2 norm ($|| \vec{v} ||_2$ or $|| \vec{v} ||$)
- other norms

$$|| \vec{v} ||_k = (\sum_{i=0}^{m-1}(v_i)^k)^{\frac{1}{k}}$$

- automating retrieval of data helpful
  - might want to update data
  - might want to repeat work on other computers
  - might want to give other members of team ability to get data
  - assures data is placed in same place every time
- inspect data using DataFrame's methods
  - head()
  - info()
  - describe()
  - hist() for whole dataset (all numerical features), or one attribute at a time
- look for...
  - scale—are all features measured on the same scale?
  - units—do we know how to interpret the numbers?
  - range—are values capped?
  - shape of distributions—long tails?
- WARNING!!
  - do not look at data too much!
  - we can easily acquire biases! (*data snooping bias*)
  - divide data into training set and test set now (typically 20%)
- want stable training/test set division
  - save test set on first run, reload on subsequent runs
  - seed random number generator to get same shuffle each time
  - both of above solutions do not work if retrieve new datasets
  - attach unique id to each record (hash function), select records with lowest 20% values
  - use row number as id? (must then delete no records and add new records only by appending)
  - combine latitude, longitude to produce id
- use function from scikit-learn
  - sklearn.model_selection.train_test_split
- stratified sampling

  - suppose that we expect very different responses from male, female survey respondents
  - U.S. population is 51.3% female, 48.7% male
  - 12% of random samples of 1000 people will have either $< 49\%$ female or $> 54\%$ female
  - better to sample in a way that guarantees close to 510 women, 490 men
  - women and men are strata (groups homogeneous in some way)
- suppose that we have reason to believe that median income strongly correlated with median housing price
- how to define strata?
- use Scikit-Learn's StratifiedShuffleSplit()
- compare distribution of results with distribution in random (unstratified) sampling
- many people do not give enough attention to how they will create a test set
- methods learned in this example will have application in cross-validation
- again, do not look at test set!
- might want to create an exploration set (a subset of training set, whose smaller size will allow easier, faster experimentation)
- make a copy of training set for experimentation
- use scatter plot (latitude, longitude) to look at geographic distribution
- "alpha" option allows better separation of circles
- "s" option links radius of circles to populations of districts
- "c" and "cmap" options (with pre-defined "jet" spectrum) links colors of circles to prices
- concentrations of population around San Francisco, Los Angeles, San Diego, and cities in Central Valley (Sacramento, Fresno)
- map shows that housing prices are related to population density and nearness to the ocean
- use corr() method to compute standard correlation coefficients (Pearson's r) for every pair of features
- correlation coefficients can take values from -1 to $+1$

- strong positive correlation between housing prices and incomes in district
- weak negative correlation between housing prices and latitude
- correlation coefficient only measure linear correlations!
- correlation coefficient does not indicate slope
- here we are asking how sure we are that one value will increase when anothe increases (not how fast it will increase!)
- pandas' scatter_matrix() method is another way to look at correlations
- do not want all $11 \times 11 = 121$ graphs!
- let's look at just 4 features (16 graphs)
- median_house_value, median_income, total_rooms, housing_median_age
- scatter_matrix() method gives histogram (rather than pointless straight line) when plotting a variable against itself
- look more closely at median_income vs. median_house_value (most promising)
- note horizontal lines at \$500K, \$450K, \$350K, maybe \$280K and elsewhere
- room data points on horizontal lines to avoid recreating artifact in model?
- transform tail-heavy distributions with logarithm?
- compute new features from existing features
  - total_rooms / households
  - total_bedrooms / total_rooms
  - population / households
- how strong is the correlation between these variables and price?
- total_bedrooms is missing some values
  - delete this feature from the dataset (DataFrame's drop() method)
  - delete records in which this value is missing (DataFrame's dropna() method)
  - fill missing values (with zero, mean, median, or something else) (DataFrame's fillna() method—use median and remember its value to use later in test set)
- Scikit-Learn's Imputer is a way to replace missing values

```
imputer = SimpleImputer(strategy="median")
```

- can only compute median on numerical attributes—create a copy of training set from which ocean proximity (a non-numerical attribute) has been removed

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

- compute values of all numerical attributes

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

- we have computed all medians, even though values are missing only in the total_bedrooms column—but no harm done
- similarly, we will ask the program to replace missing values in all columns, even knowing that in this case that really means replacing values in only one column
- replace missing values with median

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

- put data back in a DataFrame

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

- design principles of Scikit-Learn's API
  - consistency
    - estimators—parameter is dataset (or dataset and labels), hyperparameters (strategy=''median''), fit() method
    - transformers—transform() method
    - predictors—e.g., LinearPrediction, predict() and score() methods
  - inspection
    - estimator's hyperparameters are instance variables
    - estimator's learned parameters are instance variables whose identifiers end with an underscore
  - non-proliferation of classes
    - data representation with NumPy arrays or SciPy sparse matrices (rather than custom classes)
    - hyperparameters are strings or numbers
  - composition—create a Pipeline estimator by chaining together Estimators
  - sensible defaults

- ocean_proximity is a categorical variable
- create a DataFrame that contains only the ocean_proximity column

```
housing_cat = housing[["ocean_proximity"]]
```

- machine learning algorithms might work better with numerical variables
- can convert categorical to numerical with Scikit-Learn's OrdinalEncoder class

```
ordinal_encoder = OrdinalEncoder()
housing_cat_encoded = ordinal_encoder.fit_transform(housing_cat)
```

- examine transformations of first few samples

```
housing_cat_encoded[:10]
```

- examine categories

```
ordinal_encoder.categories_
```

- see that 'INLAND' follows '1H OCEAN' in listing of categories
- but '1H OCEAN' more like 'NEAR OCEAN' than like 'INLAND'!
- replace categorical variable with a binary vector–'one hot encoding'
  - length of vector equal to number of possible values for categorical variable
  - each vector contains just one '1' (all other elements are '0')
- method creates a sparse matrix

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

- this approach might not be practical if there are very large number of possible values for the categorical variable
- might be able to substitute relevant numerical variables for the categorical variable (e.g., replace a country code with population and GDP)
- might want to write custom transformers
  - Scikit-Learn uses duck typing (not inheritance)
  - enough to define fit(), transform(), and fit_transform() methods
  - TransformerMixin and BaseEstimator are useful base classes

- feature scaling
  - total number of rooms ranges from 6 to 39320
  - median incomes range from 0 to 15
  - very different scales!
  - (do not often want to scale target values)
  - min-max scaling (normalization) puts all values in 0.0–1.0 range

$$x = \frac{x - min}{max - min}$$

  - standardization establishes $mean = 0.0$ and $variance = 1.0$

$$x = \frac{x - \bar{\mathbf{X}}}{\sigma(\mathbf{X})}$$

  - standardization is much less affected by outliers
  - use just the training data to create the scaling functions
  - then apply scaling functions to training set, test set, and new data
- create a pipeline
  - constructor takes pairs—each pair a name and an estimator
  - each name is unique
  - names do not contain double underscores
  - all but last estimator must be a transformer (has a fit_transform() method)

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

housing_num_tr = num_pipeline.fit_transform(housing_num)
```

- pipeline has same methods as its last estimator
- in this example, last estimator is a transform, so pipeline has a fit_transform() method
- better to handle all columns together (rather than handle numerical and categorical features separately—as done so far in this example)
- use Scikit-Learn's ColumnTransformer

  - constructor takes tuples
  - each tuple contains a name, a transformat, and a list of names or indices of columns
  - applies transformers to appropriate columns
  - concatenates outputs of transformers (builds rows)
  - number of rows returned by each transformer must be the same
  - ColumnTransformer will return dense or sparse matrix, depending on outputs of transformers and a specified threshold
  - specify "drop" to leave out column (or columns)
  - specify "passthrough" to leave column (or columns) unchanged

```
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)
```

- **WE HAVE NOW...**
  - framed the problem
  - obtained data
  - explored the data
  - created a training set and a test set
  - constructed a pipeline of transformations
- **WE ARE READY TO SELECT AND TRAIN AN ML MODEL**
- let's use linear regression
- use Scikit-Learn's LinearRegression class
- see example
- train a linear regression model

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

- let's see how well it works...

```
# first 5 samples
some_data = housing.iloc[:5]
some_labels = housing_labels.iloc[:5]
some_data_prepared = full_pipeline.transform(some_data)

# print the predictions
print("Predictions:", lin_reg.predict(some_data_prepared))

# Predictions: [ 210644.6045   317768.8069   210956.4333
#      59218.9888   189747.5584]

# print the known values
print("Labels:", list(some_labels))

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

- big difference between prediction and known price for first sample! (predict \$210K, but actual is \$286K)
- measure regression model's RMSE...

```
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
```

- $rmse = 68628.19819848922$
- most house prices in $[\$120K, \$265K]$
- typical error $> \$68K$—would like to do better!!
- our model is underfitting the training data
  - maybe model is not sufficiently powerful?
  - maybe features do not carry enough information to allow good predictions?
- possible responses...
  - select different features
  - select different model
  - reduce constraints (not an option here, since model is not regularized)
- let's try a different model, DecisionTreeRegressor

```
from sklearn.tree import DecisionTreeRegressor

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

- see what we get with this model...

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

- result is $tree\_rmse = 0.0$! (perfect predictions unlikely!)
- now we are overfitting the data!
- try cross-validation—from training set, produce a smaller training set and a validation set
- could use train_test_split(), but better to use Scikit-Learn's K-fold cross validation
- here, divide training set into 10 "folds"
- 10 times, pick one fold for evaluation and train on the other 9 folds

```
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)
```

- notice the minus sign in front of scores—cross-validation uses a utility function, not a cost function
  - a bigger value means a better result in case of utility function
  - a smaller value means a better result in case of cost function
- here's how to check the results...

```
def display_scores(scores):
    print("Scores:", scores)
    print("Mean:", scores.mean())
    print("Standard_deviation:", scores.std())

display_scores(tree_rmse_scores)
```

- ...and here are the results (roughly, $\$71.4K \pm \$2.4K$) ...

```
Scores: [70194.33680785
         66855.16363941
         72432.58244769
         70758.73896782
         71115.88230639
         75585.14172901
         70262.86139133
         70273.6325285
         75366.87952553
         71231.65726027]
Mean: 71407.68766037929
Standard deviation: 2439.4345041191004
```

- is the decision tree model better than the linear regression model? let's see...

```
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)
```

- look at these results...

```
display_scores(lin_rmse_scores)
```

- here they are (roughly $\$69.0K \pm \$2.7K$) ...

```
Scores: [66782.73843989
         66960.118071
         70347.95244419
         74739.57052552
         68031.13388938
         71193.84183426
         64969.63056405
         68281.61137997
         71552.91566558
         67665.10082067]
Mean: 69052.46136345083
Standard deviation: 2731.674001798348
```

- in this case, decision tree model worse than linear regression model
- try one last model—RandomForestRegressor
  - train many decision trees on random subset of features
  - average the predictions
  - an example of *ensemble learning*

- code is like before (roughly $\$50.2K \pm \$2.1K$) ...

```
from sklearn.ensemble import RandomForestRegressor

forest_reg = RandomForestRegressor()
forest_reg.fit(housing_prepared, housing_labels)

forest_rmse

18603.515021376355

display_scores(forest_rmse_scores)Scores: [49519.80364233 47461.9115
        50029.02762854 52325.28068953 49308.39426421 53446.37892622
        48634.8036574  47585.73832311 53490.10699751 50021.5852922 ]
Mean: 50182.303100336096
Standard deviation: 2097.0810550985693
```

- a better result, however...
  - score on training set << score on validation set → still overfitting
  - could simplify model
  - could constrain (regularize) model
  - could get more training data

Mathematics

Arithmetic mean

$$\bar{x} = \frac{1}{n}\sum_{i=0}^{n} x_i$$

$$\bar{y} = \frac{1}{n}\sum_{i=0}^{n} y_i$$

$$\sum x_i = n \cdot \bar{x}$$

$$\sum y_i = n \cdot \bar{y}$$

Variance and covariance

$$
\begin{aligned}
ss_{xx} &= \sum (x_i - \bar{x})^2 \\
&= \sum x_i^2 - 2\bar{x} \sum x_i + \sum \bar{x}^2 \\
&= \left(\sum x_i^2 \right) - 2n\bar{x}^2 + n\bar{x}^2 \\
&= \left(\sum x_i^2 \right) - n\bar{x}^2 \\
&= n \cdot Var(x)
\end{aligned}
$$

$$
\begin{aligned}
ss_{yy} &= \sum (y_i - \bar{y})^2 \\
&= \sum y_i^2 - 2\bar{y} \sum y_i + \sum \bar{y}^2 \\
&= \left(\sum y_i^2 \right) - 2n\bar{y}^2 + n\bar{y} \\
&= \left(\sum y_i^2 \right) - n\bar{y}^2 \\
&= n \cdot Var(y)
\end{aligned}
$$

$$
\begin{aligned}
ss_{xy} &= \sum (x_i - \bar{x})(y_i - \bar{y}) \\
&= \sum (x_i x_i - \bar{x} y_i - x_i \bar{y} + \bar{x}\bar{y}) \\
&= \left(\sum x_i y_i \right) - n\bar{x}\bar{y} - n\bar{x}\bar{y} + n\bar{x}\bar{y} \\
&= \left(\sum x_i y_i \right) - n\bar{x}\bar{y} \\
&= n \cdot Cov(x, y)
\end{aligned}
$$

Least squares fit and correlation

$$r^2 = \frac{ss_{xy}^2}{ss_{xx} \cdot ss_{yy}}$$

$$y = a + b \cdot x$$

$$b = \frac{ss_{xy}}{ss_{xx}}$$

$$a = \bar{y} - b \cdot \bar{x}$$

Chapter 2

Code

2.1 Chapter 2 End-to-end Machine Learning project

Welcome to Machine Learning Housing Corp.! Your task is to predict median house values in Californian districts, given a number of features from these districts.

This notebook contains all the sample code and solutions to the exercises in chapter 2.

2.2 Setup

First, let's import a few common modules, ensure MatplotLib plots figures inline and prepare a function to save the figures. We also check that Python 3.5 or later is installed (although Python 2.x may work, it is deprecated so we strongly recommend you use Python 3 instead), as well as Scikit-Learn $\geq$ 0.2.

In[1]:

```
# Python 3.5 is required
import sys
assert sys.version_info >= (3, 5)

# Scikit-Learn 0.20 is required
import sklearn
```

```
assert sklearn.__version__ >= "0.20"

# Common imports
import numpy as np
import os

# To plot pretty figures
get_ipython().run_line_magic('matplotlib', 'inline')
import matplotlib as mpl
import matplotlib.pyplot as plt
mpl.rc('axes', labelsize=14)
mpl.rc('xtick', labelsize=12)
mpl.rc('ytick', labelsize=12)

# Where to save the figures
PROJECT_ROOT_DIR = "."
CHAPTER_ID = "end_to_end_project"
IMAGES_PATH = os.path.join(PROJECT_ROOT_DIR, "images", CHAPTER_ID)
os.makedirs(IMAGES_PATH, exist_ok=True)

def save_fig(fig_id, tight_layout=True,
             fig_extension="png", resolution=300):
    path = os.path.join(IMAGES_PATH, fig_id + "." + fig_extension)
    print("Saving figure", fig_id)
    if tight_layout:
        plt.tight_layout()
    plt.savefig(path, format=fig_extension, dpi=resolution)

# Ignore useless warnings (see SciPy issue #5998)
import warnings
warnings.filterwarnings(action="ignore", message="^internal gelsd")
```

2.3 Get the data

In[1]:

```
import os
import tarfile
import urllib

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):
    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()
```

In[2]:

```
fetch_housing_data()
```

In[3]:

```
import pandas as pd

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

In[4]:

```
housing = load_housing_data()
housing.head()
```

In[5]:

```
housing.info()
```

In[7]:

```
housing["ocean_proximity"].value_counts()
```

In[8]:

```
housing.describe()
```

In[9]:

```
get_ipython().run_line_magic('matplotlib', 'inline')
import matplotlib.pyplot as plt
housing.hist(bins=50, figsize=(20,15))
save_fig("attribute_histogram_plots")
plt.show()
```

In[10]:

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

In[11]:

```
import numpy as np

# For illustration only. Sklearn has train_test_split()
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]
```

In[12]:

```
train_set, test_set = split_train_test(housing, 0.2)
len(train_set)
```

In[13]:

```
len(test_set)
```

In[14]:

```
from zlib import crc32

def test_set_check(identifier, test_ratio):
    return crc32(np.int64(identifier)) & 0xffffffff < test_ratio * 2**32

def split_train_test_by_id(data, test_ratio, id_column):
    ids = data[id_column]
    in_test_set = ids.apply(lambda id_: test_set_check(id_, test_ratio))
    return data.loc[~in_test_set], data.loc[in_test_set]
```

The implementation of ‘test_set_check () above works fine in both Python 2 and Python 3. In earlier releases, the following implementation was proposed, which supported any hash function, but was much slower and did not support Python 2:

In[15]:

```
import hashlib

def test_set_check(identifier, test_ratio, hash=hashlib.md5):
    return hash(np.int64(identifier)).digest()[-1] < 256 * test_ratio
```

If you want an implementation that supports any hash function and is compatible with both Python 2 and Python 3, here is one:

In[16]:

```
def test_set_check(identifier, test_ratio, hash=hashlib.md5):
    return bytearray(hash(np.int64(identifier)).digest())[-1] \
           < 256 * test_ratio
```

In[17]:

```
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")
```

In[18]:

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

In[19]:

```
test_set.head()
```

In[20]:

```
from sklearn.model_selection import train_test_split

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

In[21]:

```
test_set.head()
```

In[22]:

```
housing["median_income"].hist()
```

In[23]:

```
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])
```

In[24]:

```
housing["income_cat"].value_counts()
```

In[25]:

```
housing["income_cat"].hist()
```

In[26]:

```
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]
```

In[27]:

```
strat_test_set["income_cat"].value_counts() / len(strat_test_set)
```

In[28]:

```
housing["income_cat"].value_counts() / len(housing)
```

In[29]:

```
def income_cat_proportions(data):
    return data["income_cat"].value_counts() / len(data)

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

compare_props = pd.DataFrame({
    "Overall": income_cat_proportions(housing),
    "Stratified": income_cat_proportions(strat_test_set),
    "Random": income_cat_proportions(test_set),
}).sort_index()
compare_props["Rand. %error"] = \
        100 * compare_props["Random"] / \
        compare_props["Overall"] - 100
compare_props["Strat. %error"] = \\
        100 * compare_props["Stratified"] / \
        compare_props["Overall"] - 100
```

In[30]:

```
compare_props
```

In[31]:

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

2.4 Discover and visualize the data to gain insights

In[32]:

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

In[33]:

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

In[34]:

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

The argument sharex=False fixes a display bug (the x-axis values and legend were not displayed). This is a temporary fix (see: https://github.com/pandas-dev/pandas/issues/10611). Thanks to Wilmer Arellano for pointing it out.

In[35]:

```
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,
    sharex=False)
plt.legend()
save_fig("housing_prices_scatterplot")
```

In[36]:

```
# Download the California image
images_path = \
    os.path.join(PROJECT_ROOT_DIR, "images", "end_to_end_project")
os.makedirs(images_path, exist_ok=True)
DOWNLOAD_ROOT = \
    "https://raw.githubusercontent.com/ageron/handson-ml2/master/"
filename = "california.png"
print("Downloading", filename)
url = DOWNLOAD_ROOT + "images/end_to_end_project/" + filename
urllib.request.urlretrieve(url, os.path.join(images_path, filename))
```

In[37]:

```
import matplotlib.image as mpimg
california_img=mpimg.imread(os.path.join(images_path, filename))
ax = housing.plot(kind="scatter", x="longitude", y="latitude",
                  figsize=(10,7),
                  s=housing['population']/100, label="Population",
                  c="median_house_value", cmap=plt.get_cmap("jet"),
                  colorbar=False, alpha=0.4,
                      )
plt.imshow(california_img,
           extent=[-124.55, -113.80, 32.45, 42.05],
           alpha=0.5,
           cmap=plt.get_cmap("jet"))
plt.ylabel("Latitude", fontsize=14)
```

```
plt.xlabel("Longitude", fontsize=14)

prices = housing["median_house_value"]
tick_values = np.linspace(prices.min(), prices.max(), 11)
cbar = plt.colorbar()
cbar.ax.set_yticklabels(["$%dk"%(round(v/1000)) for v in tick_values],
                        fontsize=14)
cbar.set_label('Median House Value', fontsize=16)

plt.legend(fontsize=16)
save_fig("california_housing_prices_plot")
plt.show()
```

In[38]:

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

In[39]:

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

In[40]:

```
# from pandas.tools.plotting import scatter_matrix
# For older versions of Pandas
from pandas.plotting import scatter_matrix

attributes = ["median_house_value", "median_income", "total_rooms",
              "housing_median_age"]
scatter_matrix(housing[attributes], figsize=(12, 8))
save_fig("scatter_matrix_plot")
```

In[41]:

```
housing.plot(kind="scatter", x="median_income", y="median_house_value",
             alpha=0.1)
plt.axis([0, 16, 0, 550000])
save_fig("income_vs_house_value_scatterplot")
```

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)
```

In[44]:

```
housing.plot(kind="scatter",
             x="rooms_per_household", y="median_house_value",
             alpha=0.2)
plt.axis([0, 5, 0, 520000])
plt.show()
```

In[45]:

```
housing.describe()
```

2.5 Prepare the data for Machine Learning algorithms

In[46]:

```
# drop labels for training set
housing = strat_train_set.drop("median_house_value", axis=1)
housing_labels = strat_train_set["median_house_value"].copy()
```

In[47]:

```
sample_incomplete_rows = \
        housing[housing.isnull().any(axis=1)].head()
sample_incomplete_rows
```

In[48]:

```
# option 1
sample_incomplete_rows.dropna(subset=["total_bedrooms"])
```

In[49]:

```
# option 2
sample_incomplete_rows.drop("total_bedrooms", axis=1)
```

In[50]:

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

In[51]:

```
sample_incomplete_rows
```

In[52]:

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

Remove the text attribute because median can only be calculated on numerical attributes:

In[53]:

```
housing_num = housing.drop("ocean_proximity", axis=1)
# alternatively: housing_num = \
#       housing.select_dtypes(include=[np.number])
```

In[54]:

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

In[55]:

```
imputer.statistics_
```

Check that this is the same as manually computing the median of each attribute:

In[56]:

```
housing_num.median().values
```

Transform the training set:

In[57]:

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

In[58]:

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

In[59]:

```
housing_tr.loc[sample_incomplete_rows.index.values]
```

In[60]:

```
imputer.strategy
```

In[61]:

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

In[62]:

```
housing_tr.head()
```

Now let's preprocess the categorical input feature, 'ocean_proximity':

In[63]:

```
housing_cat = housing[["ocean_proximity"]]
housing_cat.head(10)
```

In[64]:

```
from sklearn.preprocessing import OrdinalEncoder

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

In[65]:

```
ordinal_encoder.categories_
```

In[66]:

```
from sklearn.preprocessing import OneHotEncoder

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

By default, the 'OneHotEncoder' class returns a sparse array, but we can convert it to a dense array if needed by calling the 'toarray()' method:

In[67]:

```
housing_cat_1hot.toarray()
```

Alternatively, you can set sparse=False when creating the OneHotEncoder:

In[68]:

```
cat_encoder = OneHotEncoder(sparse=False)
housing_cat_1hot = cat_encoder.fit_transform(housing_cat)
housing_cat_1hot
```

In[69]:

```
cat_encoder.categories_
```

Let's create a custom transformer to add extra attributes:

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):
    # no *args or **kargs
    def __init__(self, add_bedrooms_per_room = True):
        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)
```

In[71]:

```
housing_extra_attribs = pd.DataFrame(
    housing_extra_attribs,
    columns=list(housing.columns) + \
            ["rooms_per_household", "population_per_household"],
    index=housing.index)
housing_extra_attribs.head()
```

Now let's build a pipeline for preprocessing the numerical attributes:

In[72]:

```
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)
```

In[73]:

```
housing_num_tr
```

In[74]:

```
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)
```

In[75]:

```
housing_prepared
```

In[76]:

```
housing_prepared.shape
```

For reference, here is the old solution based on a DataFrameSelector transformer (to just select a subset of the Pandas 'DataFrame' columns), and a FeatureUnion:

In[77]:

```
from sklearn.base import BaseEstimator, TransformerMixin

# Create a class to select numerical or categorical columns
class OldDataFrameSelector(BaseEstimator, TransformerMixin):
    def __init__(self, attribute_names):
```

```
        self.attribute_names = attribute_names
    def fit(self, X, y=None):
        return self
    def transform(self, X):
        return X[self.attribute_names].values
```

Now let's join all these components into a big pipeline that will preprocess both the numerical and the categorical features:

In[78]:

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

old_num_pipeline = Pipeline([
        ('selector', OldDataFrameSelector(num_attribs)),
        ('imputer', SimpleImputer(strategy="median")),
        ('attribs_adder', CombinedAttributesAdder()),
        ('std_scaler', StandardScaler()),
    ])

old_cat_pipeline = Pipeline([
        ('selector', OldDataFrameSelector(cat_attribs)),
        ('cat_encoder', OneHotEncoder(sparse=False)),
    ])
```

In[79]:

```
from sklearn.pipeline import FeatureUnion

old_full_pipeline = FeatureUnion(transformer_list=[
        ("num_pipeline", old_num_pipeline),
        ("cat_pipeline", old_cat_pipeline),
    ])
```

In[80]:

```
old_housing_prepared = old_full_pipeline.fit_transform(housing)
```

```
old_housing_prepared
```

The result is the same with the ColumnTransformer:

In[81]:

```
np.allclose(housing_prepared, old_housing_prepared)
```

2.6 Select and train a model

In[82]:

```
from sklearn.linear_model import LinearRegression

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

In[83]:

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))
```

Compare against the actual values:

In[84]:

```
print("Labels:", list(some_labels))
```

In[85]:

```
some_data_prepared
```

In[86]:

```
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
```

In[87]:

```
from sklearn.metrics import mean_absolute_error

lin_mae = mean_absolute_error(housing_labels, housing_predictions)
lin_mae
```

In[88]:

```
from sklearn.tree import DecisionTreeRegressor

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

In[89]:

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

2.7 Fine-tune your model

In[90]:

```
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)
```

In[91]:

```
def display_scores(scores):
    print("Scores:", scores)
    print("Mean:", scores.mean())
    print("Standard_deviation:", scores.std())

display_scores(tree_rmse_scores)
```

In[92]:

```
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)
```

Note: we specify n_estimators=100 to be future-proof since the default value is going to change to 100 in Scikit-Learn 0.22 (for simplicity, this is not shown in the book).

In[93]:

```
from sklearn.ensemble import RandomForestRegressor
```

```
forest_reg = RandomForestRegressor(n_estimators=100, random_state=42)
forest_reg.fit(housing_prepared, housing_labels)
```

In[94]:

```
housing_predictions = forest_reg.predict(housing_prepared)
forest_mse = mean_squared_error(housing_labels, housing_predictions)
forest_rmse = np.sqrt(forest_mse)
forest_rmse
```

In[95]:

```
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)
```

In[96]:

```
scores = cross_val_score(lin_reg, housing_prepared,
                         housing_labels,
                         scoring="neg_mean_squared_error",
                         cv=10)
pd.Series(np.sqrt(-scores)).describe()
```

In[97]:

```
from sklearn.svm import SVR

svm_reg = SVR(kernel="linear")
svm_reg.fit(housing_prepared, housing_labels)
housing_predictions = svm_reg.predict(housing_prepared)
```

```
svm_mse = mean_squared_error(housing_labels, housing_predictions)
svm_rmse = np.sqrt(svm_mse)
svm_rmse
```

In[98]:

```
from sklearn.model_selection import GridSearchCV

param_grid = [
    # try 12 (3  4 ) combinations of hyperparameters
    {'n_estimators': [3, 10, 30], 'max_features': [2, 4, 6, 8]},
    # then try 6 (2  3 ) 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)
```

The best hyperparameter combination found:

In[99]:

```
grid_search.best_params_
```

In[100]:

```
grid_search.best_estimator_
```

Let's look at the score of each hyperparameter combination tested during the grid search:

In[101]:

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

In[102]:

```
pd.DataFrame(grid_search.cv_results_)
```

In[103]:

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

param_distribs = {
        '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_distribs,
                           n_iter=10, cv=5,
                           scoring='neg_mean_squared_error',
                           random_state=42)
rnd_search.fit(housing_prepared, housing_labels)
```

In[104]:

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

In[105]:

```
feature_importances = grid_search.best_estimator_.feature_importances_
feature_importances
```

In[106]:

```
extra_attribs = ["rooms_per_hhold", "pop_per_hhold", "bedrooms_per_room"]
#cat_encoder = cat_pipeline.named_steps["cat_encoder"] # old solution
cat_encoder = full_pipeline.named_transformers_["cat"]
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)
```

In[107]:

```
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[108]:

```
final_rmse
```

We can compute a 95% confidence interval for the test RMSE:

In[109]:

```
from scipy import stats

confidence = 0.95
squared_errors = (final_predictions - y_test) ** 2
np.sqrt(stats.t.interval(confidence, len(squared_errors) - 1,
                         loc=squared_errors.mean(),
                         scale=stats.sem(squared_errors)))
```

We could compute the interval manually like this:

In[110]:

```
m = len(squared_errors)
mean = squared_errors.mean()
tscore = stats.t.ppf((1 + confidence) / 2, df=m - 1)
tmargin = tscore * squared_errors.std(ddof=1) / np.sqrt(m)
np.sqrt(mean - tmargin), np.sqrt(mean + tmargin)
```

Alternatively, we could use a z-scores rather than t-scores:

In[111]:

```
zscore = stats.norm.ppf((1 + confidence) / 2)
zmargin = zscore * squared_errors.std(ddof=1) / np.sqrt(m)
np.sqrt(mean - zmargin), np.sqrt(mean + zmargin)
```

2.8 Extra material

2.8.1 A full pipeline with both preparation and prediction

In[112]:

```
full_pipeline_with_predictor = Pipeline([
        ("preparation", full_pipeline),
```

```
        ("linear", LinearRegression())
    ])

full_pipeline_with_predictor.fit(housing, housing_labels)
full_pipeline_with_predictor.predict(some_data)
```

2.8.2 Model persistence using joblib

In[113]:

```
my_model = full_pipeline_with_predictor
```

In[114]:

```
import joblib
joblib.dump(my_model, "my_model.pkl") # DIFF
#...
my_model_loaded = joblib.load("my_model.pkl") # DIFF
```

2.8.3 Example SciPy distributions for 'RandomizedSearchCV'

In[115]:

```
from scipy.stats import geom, expon
geom_distrib=geom(0.5).rvs(10000, random_state=42)
expon_distrib=expon(scale=1).rvs(10000, random_state=42)
plt.hist(geom_distrib, bins=50)
plt.show()
plt.hist(expon_distrib, bins=50)
plt.show()
```

2.9 Exercise solutions

2.9.1 1.

Question: Try a Support Vector Machine regressor ('sklearn.svm.SVR'), with various hyperparameters such as kernel="linear" (with various values for the 'C' hyperparameter) or kernel="rbf" (with various values for the 'C' and 'gamma' hyperparameters). Don't worry about what these hyperparameters mean for now. How does the best 'SVR' predictor perform?

In[116]:

```
from sklearn.model_selection import GridSearchCV

param_grid = [
        {'kernel': ['linear'],
         'C': [10., 30., 100., 300., 1000., 3000., 10000., 30000.0]},
        {'kernel': ['rbf'], 'C': [1.0, 3.0, 10., 30., 100., 300., 1000.0],
         'gamma': [0.01, 0.03, 0.1, 0.3, 1.0, 3.0]},
    ]

svm_reg = SVR()
grid_search = GridSearchCV(svm_reg, param_grid, cv=5,
                           scoring='neg_mean_squared_error', verbose=2)
grid_search.fit(housing_prepared, housing_labels)
```

The best model achieves the following score (evaluated using 5-fold cross validation):

In[117]:

```
negative_mse = grid_search.best_score_
rmse = np.sqrt(-negative_mse)
rmse
```

That's much worse than the 'RandomForestRegressor.' Let's check the best hyperparameters found:

In[118]:

```
grid_search.best_params_
```

The linear kernel seems better than the RBF kernel. Notice that the value of 'C' is the maximum tested value. When this happens you definitely want to launch the grid search again with higher values for 'C' (removing the smallest values), because it is likely that higher values of 'C' will be better.

2.9.2 2.

Question: Try replacing 'GridSearchCV' with 'RandomizedSearchCV.'

In[119]:

```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import expon, reciprocal

# see https://docs.scipy.org/doc/scipy/reference/stats.html
# for `expon()` and `reciprocal()` documentation and more
# probability distribution functions.

# Note: gamma is ignored when kernel is "linear"
param_distribs = {
        'kernel': ['linear', 'rbf'],
        'C': reciprocal(20, 200000),
        'gamma': expon(scale=1.0),
    }

svm_reg = SVR()
rnd_search = RandomizedSearchCV(svm_reg,
                                param_distributions=param_distribs,
                                n_iter=50, cv=5,
                                scoring='neg_mean_squared_error',
                                verbose=2, random_state=42)
rnd_search.fit(housing_prepared, housing_labels)
```

The best model achieves the following score (evaluated using 5-fold cross validation):

In[120]:

```
negative_mse = rnd_search.best_score_
rmse = np.sqrt(-negative_mse)
rmse
```

Now this is much closer to the performance of the 'RandomForestRegressor' (but not quite there yet). Let's check the best hyperparameters found:

In[121]:

```
rnd_search.best_params_
```

This time the search found a good set of hyperparameters for the RBF kernel. Randomized search tends to find better hyperparameters than grid search in the same amount of time.

Let's look at the exponential distribution we used, with scale=1.0. Note that some samples are much larger or smaller than 1.0, but when you look at the log of the distribution, you can see that most values are actually concentrated roughly in the range of exp(-2) to exp(+2), which is about 0.1 to 7.4.

In[122]:

```
expon_distrib = expon(scale=1.)
samples = expon_distrib.rvs(10000, random_state=42)
plt.figure(figsize=(10, 4))
plt.subplot(121)
plt.title("Exponential distribution (scale=1.0)")
plt.hist(samples, bins=50)
plt.subplot(122)
plt.title("Log of this distribution")
plt.hist(np.log(samples), bins=50)
plt.show()
```

The distribution we used for 'C' looks quite different: the scale of the samples is picked from a uniform distribution within a given range, which is why the right graph, which represents the log of the samples, looks roughly constant. This distribution is useful when you don't have a clue of what the target scale is:

In[123]:

```
reciprocal_distrib = reciprocal(20, 200000)
samples = reciprocal_distrib.rvs(10000, random_state=42)
plt.figure(figsize=(10, 4))
plt.subplot(121)
plt.title("Reciprocal distribution (scale=1.0)")
plt.hist(samples, bins=50)
plt.subplot(122)
plt.title("Log of this distribution")
plt.hist(np.log(samples), bins=50)
plt.show()
```

The reciprocal distribution is useful when you have no idea what the scale of the hyperparameter should be (indeed, as you can see on the figure on the right, all scales are equally likely, within the given range), whereas the exponential distribution is best when you know (more or less) what the scale of the hyperparameter should be.

2.9.3 3.

Question: Try adding a transformer in the preparation pipeline to select only the most important attributes.

In[124]:

```
from sklearn.base import BaseEstimator, TransformerMixin

def indices_of_top_k(arr, k):
    return np.sort(np.argpartition(np.array(arr), -k)[-k:])

class TopFeatureSelector(BaseEstimator, TransformerMixin):
    def __init__(self, feature_importances, k):
        self.feature_importances = feature_importances
        self.k = k
    def fit(self, X, y=None):
        self.feature_indices_ = \
                indices_of_top_k(self.feature_importances, self.k)
        return self
    def transform(self, X):
        return X[:, self.feature_indices_]
```

Note: this feature selector assumes that you have already computed the feature importances somehow (for example using a 'RandomForestRegressor'). You may be tempted to compute them directly in the 'TopFeatureSelector' 's 'fit()' method, however this would likely slow down grid/randomized search since the feature importances would have to be computed for every hyperparameter combination (unless you implement some sort of cache).

Let's define the number of top features we want to keep:

In[125]:

```
k = 5
```

Now let's look for the indices of the top k features:

In[126]:

```
top_k_feature_indices = indices_of_top_k(feature_importances, k)
top_k_feature_indices
```

In[127]:

```
np.array(attributes)[top_k_feature_indices]
```

Let's double check that these are indeed the top k features:

In[128]:

```
sorted(zip(feature_importances, attributes), reverse=True)[:k]
```

Looking good... Now let's create a new pipeline that runs the previously defined preparation pipeline, and adds top k feature selection:

In[129]:

```
preparation_and_feature_selection_pipeline = Pipeline([
    ('preparation', full_pipeline),
    ('feature_selection', TopFeatureSelector(feature_importances, k))
])
```

In[130]:

```
housing_prepared_top_k_features = \
        preparation_and_feature_selection_pipeline.fit_transform(housing)
```

Let's look at the features of the first 3 instances:

In[131]:

```
housing_prepared_top_k_features[0:3]
```

Now let's double check that these are indeed the top k features:

In[132]:

```
housing_prepared[0:3, top_k_feature_indices]
```

Works great! :)

2.9.4 4.

Question: Try creating a single pipeline that does the full data preparation plus the final prediction.

In[133]:

```
prepare_select_and_predict_pipeline = Pipeline([
    ('preparation', full_pipeline),
    ('feature_selection', TopFeatureSelector(feature_importances, k)),
    ('svm_reg', SVR(**rnd_search.best_params_))
```

```
])
```

In[134]:

```
prepare_select_and_predict_pipeline.fit(housing, housing_labels)
```

Let's try the full pipeline on a few instances:

In[135]:

```
some_data = housing.iloc[:4]
some_labels = housing_labels.iloc[:4]

print("Predictions:\t",
      prepare_select_and_predict_pipeline.predict(some_data))
print("Labels:\t\t", list(some_labels))
```

Well, the full pipeline seems to work fine. Of course, the predictions are not fantastic: they would be better if we used the best 'RandomForestRegressor' that we found earlier, rather than the best 'SVR.'

2.9.5 5.

Question: Automatically explore some preparation options using 'GridSearchCV.'

In[136]:

```
param_grid = [{
    'preparation__num__imputer__strategy': ['mean', 'median', 'most_frequent'],
    'feature_selection__k': list(range(1, len(feature_importances) + 1))
}]

grid_search_prep = GridSearchCV(prepare_select_and_predict_pipeline,
                                param_grid, cv=5,
                                scoring='neg_mean_squared_error', verbose=2)
grid_search_prep.fit(housing, housing_labels)
```

In[137]:

```
grid_search_prep.best_params_
```

The best imputer strategy is 'most_frequent' and apparently almost all features are useful (15 out of 16). The last one ('ISLAND') seems to just add some noise.

Congratulations! You already know quite a lot about Machine Learning. :)