End-to-End Machine Learning Project

Hands-on Machine Learning with Scikit-Learn and Tensor Flow (Chapter-2)

Dr. Muhammad Usman Ghani
Associate Professor in Department of Computer Science & Engineering

University of Engineering and Technology, Lahore, Pakistan

Chapter division

- This chapter is divided into following parts:
  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 Machine Learning algorithms
  5. Explore many different models and short-list 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

1.Frame the problem

- It includes following points:
  - What exactly is the business objective
  - How does the company expect to use and benefit from this model?
  - What algorithms will be selected
  - What performance measure will be used to evaluate model
  - Frame the problem: is it supervised, unsupervised, or Reinforcement Learning?
  - Is it a classification task, a regression task, or something else?

1.Frame the problem

- It includes following points:
  - How should performance be measured?
  - What would be the minimum performance needed to reach the business objective?
  - How would the problem be solved manually?
  - List the assumptions you (or others) have made so far
  - Verify assumptions if possible

Data Pipeline

- A sequence of data processing components is called a data pipeline
- A piece of information fed to a Machine Learning system is often called a signal

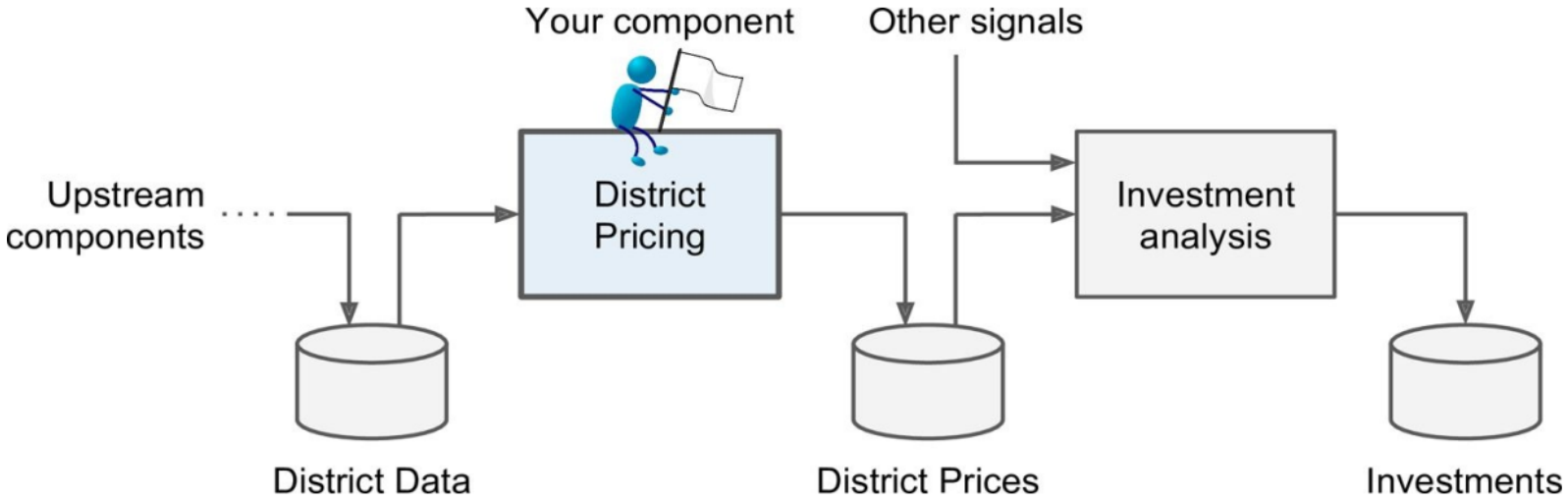


Figure 2-2. A Machine Learning pipeline for real estate investments

Performance Measure

- There are two types of performance measures used in this project for regression problem:
  1. MAE(Mean Absolute error)
  2. RMSE(Root mean squared error)

- Root Mean Square Error (RMSE)also known as Root Mean Square Deviation measures how much error there is between two data sets. In other words, it compares a predicted value and an observed or known value.The smaller an RMSE value, the closer predicted and observed values are
- Mean Absolute Error (MAE) measures how far predicted values are away from observed values.The Mean Absolute Error(MAE) is the average of all absolute errors

Performance Measure

- x(i) is a vector of all the feature values of the ith instance in the dataset, and y(i) is its label and m is the number of instances in the dataset.h is a prediction function, also called a hypothesis. When our system is given an instance's feature vector x(i), it outputs a predicted value $\hat{y}(i) = h(x(i))$ for that instance

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

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

2.Get the data

- The data set used is California Housing Prices dataset from the StatLib repository2.This dataset was based on data from the 1990 California census.
  Getting data includes following points:

1. List the data needed and how much needed
2. Find and document where we can get that data
3. Check how much space it will take
4. Check legal obligations, and get authorization if necessary
5. Get access authorizations
6. Create a workspace (with enough storage space)
7. Get the data and Convert the data to a format you can easily manipulate (without changing the data itself)
8. Ensure sensitive information is deleted or protected
9. Check the size and type of data (time series, sample, geographical, etc.)

Fetch the data

- We use this code to fetch the data from zipped file.
- When we call fetch_housing_data(), it creates a datasets/housing directory in our workspace, downloads the housing.tgz file, and extracts the housing.csv from it in this directory

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

DOWNLOAD_ROOT = "https://raw.githubusercontent.com/ageron/handson-ml/master/"
HOUSING_PATH = "datasets/housing"
HOUSING_URL = DOWNLOAD_ROOT + HOUSING_PATH + "/housing.tgz"

def fetch_housing_data(housing_url=HOUSING_URL, housing_path=HOUSING_PATH):
    if not os.path.isdir(housing_path):
        os.makedirs(housing_path)
    tgz_path = os.path.join(housing_path, "housing.tgz")
    urllib.request.urlretrieve(housing_url, tgz_path)
    housing_tgz = tarfile.open(tgz_path)
    housing_tgz.extractall(path=housing_path)
    housing_tgz.close()
```

Load the Data

- We write following function to load the data using Pandas

```
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)
```

Top five rows in the dataset

- Each row represents one district. There are 10 attributes longitude, latitude, housing_median_age, total_rooms, total_bedrooms, population, households, median_income, median_house_value, and ocean_proximity lines, tabs and more
- The dataset will look like this

```
In [5]: housing = load_housing_data()
        housing.head()
```

Out[5]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	populatio
0	-122.23	37.88	41.0	880.0	129.0	322.0
1	-122.22	37.86	21.0	7099.0	1106.0	2401.0
2	-122.24	37.85	52.0	1467.0	190.0	496.0
3	-122.25	37.85	52.0	1274.0	235.0	558.0
4	-122.25	37.85	52.0	1627.0	280.0	565.0

Figure 2-5. Top five rows in the dataset

Housing info

- We use info() method to get a quick description of the data


```
housing.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 10 columns):
longitude             20640 non-null float64
latitude              20640 non-null float64
housing_median_age    20640 non-null float64
total_rooms           20640 non-null float64
total_bedrooms        20433 non-null float64
population            20640 non-null float64
households            20640 non-null float64
median_income         20640 non-null float64
median_house_value    20640 non-null float64
ocean_proximity       20640 non-null object
dtypes: float64(9), object(1)
memory usage: 1.6+ MB
```

Figure 2-6. Housing info

Summary of each numerical attribute

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


```
housing.describe()
```

	longitude	latitude	housing_median_age	total_rooms	total_bedr
count	20640.000000	20640.000000	20640.000000	20640.000000	20433.000
mean	-119.569704	35.631861	28.639486	2635.763081	537.870553
std	2.003532	2.135952	12.585558	2181.615252	421.385070
min	-124.350000	32.540000	1.000000	2.000000	1.000000
25%	-121.800000	33.930000	18.000000	1447.750000	296.000000
50%	-118.490000	34.260000	29.000000	2127.000000	435.000000
75%	-118.010000	37.710000	37.000000	3148.000000	647.000000
max	-114.310000	41.950000	52.000000	39320.000000	6445.0000

Figure 2-7. Summary of each numerical attribute

A histogram for each numerical attribute

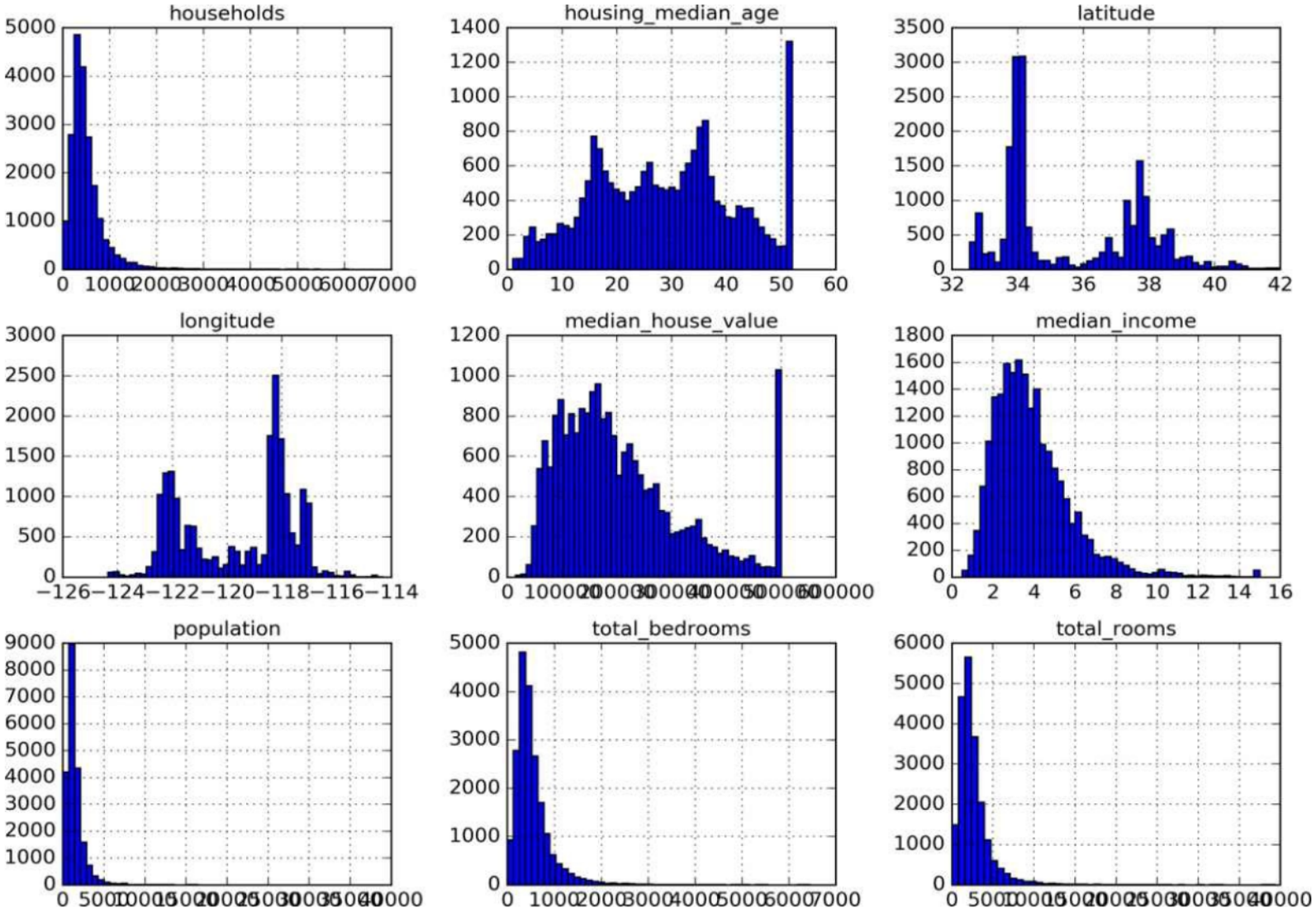


Figure 2-8. A histogram for each numerical attribute

Create a test set

- We create a test set using following code

```
import numpy as np

def split_train_test(data, test_ratio):
    shuffled_indices = np.random.permutation(len(data))
    test_set_size = int(len(data) * test_ratio)
    test_indices = shuffled_indices[:test_set_size]
    train_indices = shuffled_indices[test_set_size:]
    return data.iloc[train_indices], data.iloc[test_indices]
```

Split dataset

- Scikit-Learn provides a few functions to split datasets into multiple subsets in various ways. The simplest function is train_test_split, which does pretty much the same thing as the function

```
from sklearn.model_selection import train_test_split

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

Sampling Methods

- Sampling is done because we usually cannot gather data from the entire population. Even in relatively small populations, the data may be needed urgently, and including everyone in the population in our data collection may take too long
- Two sampling methods are used:

1. Purely random sampling
2. Stratified sampling

Purely Random Sampling

- A random sample is a subset of a statistical population in which each member of the subset has an equal probability of being chosen
- Random samples involve the random selection of data from the entire population so each possible sample is equally likely to occur

Stratified sampling

- The population is divided into homogeneous subgroups called strata, and the right number of instances is sampled from each stratum to guarantee that the test set is representative of the overall population
- Random samples are then selected from each stratum

Sampling bias comparison of stratified versus purely random sampling

- The test set generated using stratified sampling has income category proportions almost identical to those in the full dataset, whereas the test set generated using purely random sampling is quite skewed

	Overall	Random	Stratified	Rand. %error	Strat. %error
1.0	0.039826	0.040213	0.039738	0.973236	-0.219137
2.0	0.318847	0.324370	0.318876	1.732260	0.009032
3.0	0.350581	0.358527	0.350618	2.266446	0.010408
4.0	0.176308	0.167393	0.176399	-5.056334	0.051717
5.0	0.114438	0.109496	0.114369	-4.318374	-0.060464

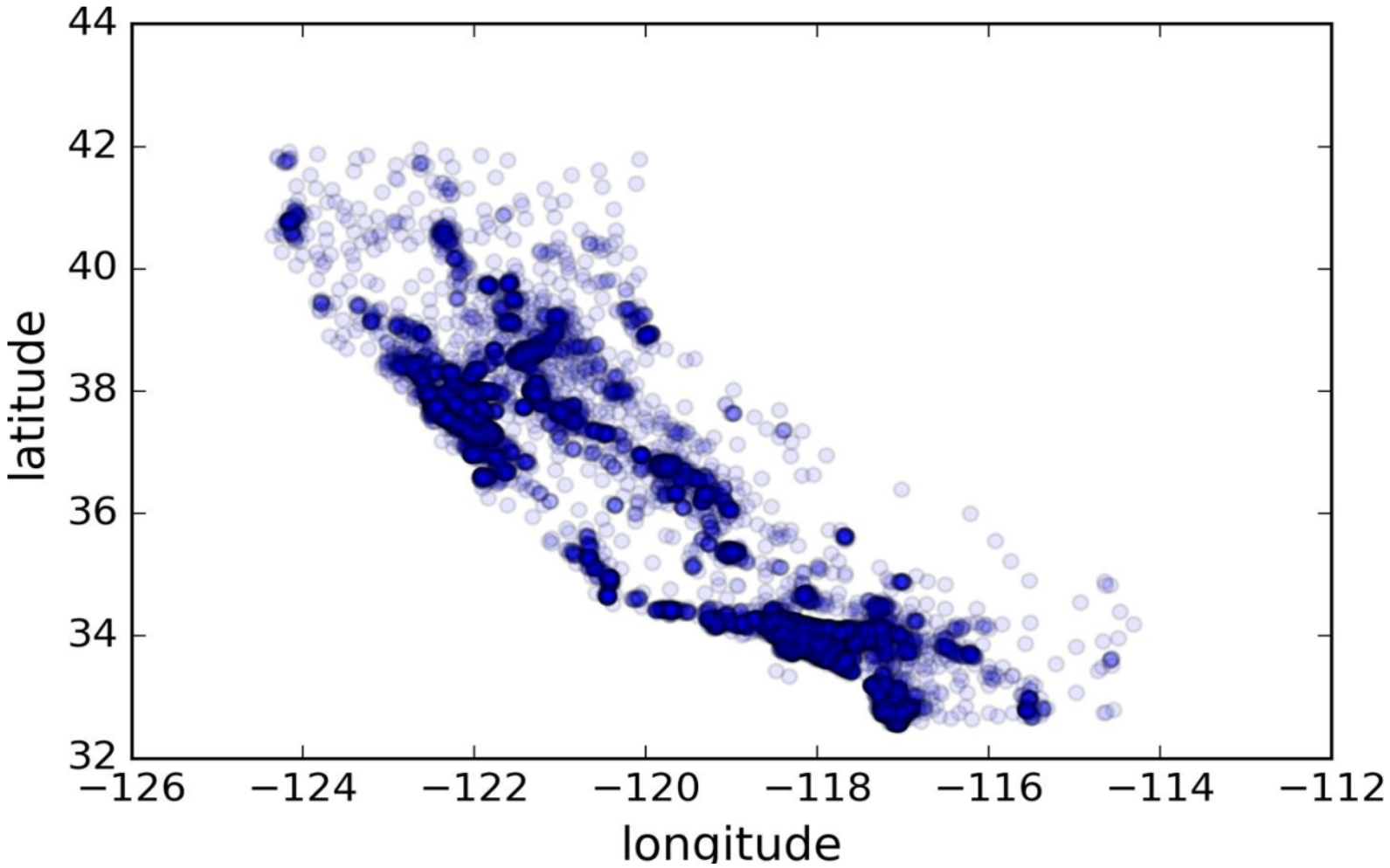
Figure 2-10. Sampling bias comparison of stratified versus purely random sampling

3.Explore the data to gain insights

- Create a copy of the data for exploration
- Study each attribute and its characteristics
  - Name
  - Type (categorical, int/float, bounded/unbounded, text, structured, etc.)
  - % of missing values
  - Noisiness and type of noise (stochastic, outliers, rounding errors, etc.)
  - Filter out Stop Words (and Pipeline)
  - Stem Words
- Visualize the data
- For supervised learning tasks, identify the target attribute(s)
- Study the correlations between attributes

Visualize the geographical data

- To visualize the data,we create a scatter plot



California housing prices

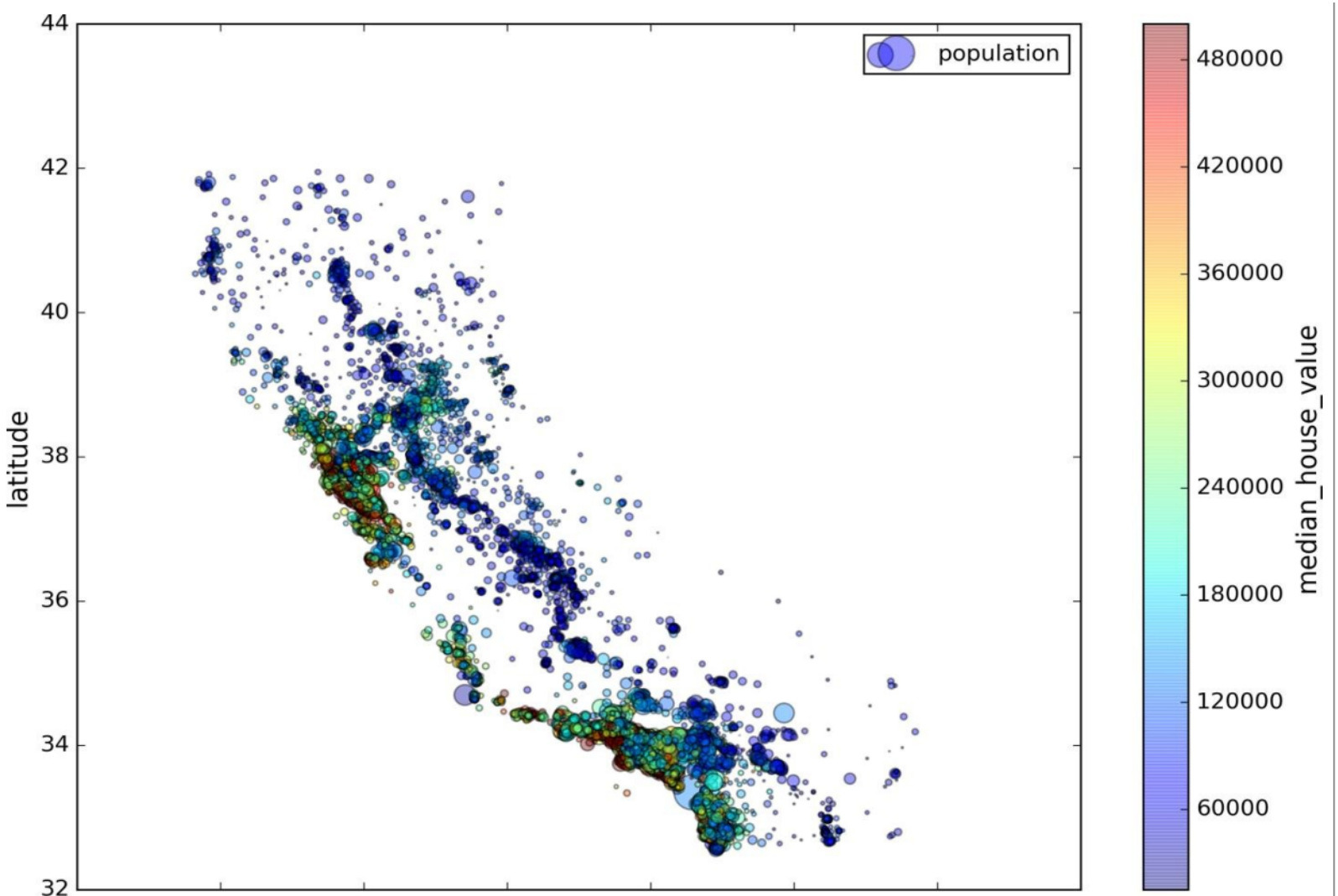


Figure 2-13. California housing prices

how much each attribute correlates with the median house value


```
>>> corr_matrix["median_house_value"].sort_values(ascending=False)
median_house_value    1.000000
median_income         0.687170
total_rooms           0.135231
housing_median_age    0.114220
households            0.064702
total_bedrooms        0.047865
population           -0.026699
longitude            -0.047279
latitude             -0.142826
Name: median_house_value, dtype: float64
```

Looking for Correlations

- We compute the standard correlation coefficient (also called Pearson's r) between every pair of attributes using the corr() method
- The correlation coefficient ranges from –1 to 1
- When it is close to 1, it means that there is a strong positive correlation; for example, the median house value tends to go up when the median income goes up
- When the coefficient is close to minus 1, it means that there is a strong negative correlation; we can see a small negative correlation between the latitude and the median house value (i.e., prices have a slight tendency to go down when we go to north)

4.Prepare the data for machine learning

- Work on copies of the data (keep the original dataset intact)
- Write functions for all data transformations you apply

1. Data cleaning:
   - Fix or remove outliers (optional)
   - Fill in missing values (e.g., with zero, mean, median...) or drop their rows (or columns)
2. Feature selection (optional):
   - Drop the attributes that provide no useful information for the task
3. Feature engineering, where appropriate
   - Decompose features (e.g., categorical, date/time, etc.)
   - Add promising transformations of features (e.g., log(x), sqrt(x), x2, etc.)

Data Cleaning

- Most Machine Learning algorithms cannot work with missing features so first fix them
  - Get rid of the corresponding districts
  - Get rid of the whole attribute
  - Set the values to some value (zero, the mean, the median, etc.)
- we can accomplish this by using DataFrame's dropna(), drop(), and fillna() methods

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

Data Cleaning

- We should compute the median value on the training set, and use it to fill the missing values in the training set
- Scikit-Learn provides a handy class to take care of missing values: Imputer First, we need to create an Imputer instance, specifying that we want to replace each attribute's missing values with the median of that attribute

```
from sklearn.preprocessing import Imputer

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

Data Cleaning

- Fit the imputer instance to the training data using the fit() method
- imputer.fit(housing_num)
- The imputer has simply computed the median of each attribute and stored the result in its statistics_ instance variable
- imputer.statistics_

Data Cleaning

- We can use this "trained" imputer to transform the training set by replacing missing values by the learned medians
- X = imputer.transform(housing_num)

Handling text data and categorical attributes

- Ocean_proximity is a categorical attribute so we 'll convert it into numbers by using transformer called LabelEncoder

```
>>> from sklearn.preprocessing import LabelEncoder
>>> encoder = LabelEncoder()
>>> housing_cat = housing["ocean_proximity"]
>>> housing_cat_encoded = encoder.fit_transform(housing_cat)
>>> housing_cat_encoded
array([0, 0, 4, ..., 1, 0, 3])
```

Handling text data and categorical attributes

- Create one binary attribute per category: one attribute equal to 1 when the category is "<1H OCEAN" (and 0 otherwise), another attribute equal to 1 when the category is "INLAND" (and 0 otherwise), and so on. This is called one-hot encoding

```
>>> from sklearn.preprocessing import OneHotEncoder
>>> encoder = OneHotEncoder()
>>> housing_cat_1hot = encoder.fit_transform(housing_cat_encoded.reshape(-1,1))
>>> housing_cat_1hot
<16512x5 sparse matrix of type '<class 'numpy.float64'>'
        with 16512 stored elements in Compressed Sparse Row format>
```

Handling text data and categorical attributes

- We can apply both transformations (from text categories to integer categories, then from integer categories to one-hot vectors) in one shot using the LabelBinarizer class

```
>>> from sklearn.preprocessing import LabelBinarizer
>>> encoder = LabelBinarizer()
>>> housing_cat_1hot = encoder.fit_transform(housing_cat)
>>> housing_cat_1hot
array([[1, 0, 0, 0, 0],
       [1, 0, 0, 0, 0],
       [0, 0, 0, 0, 1],
       ...,
       [0, 1, 0, 0, 0],
       [1, 0, 0, 0, 0],
       [0, 0, 0, 1, 0]])
```

Custom Transformers

- Create a class and implement three methods: fit() (returning self), transform(), and fit_transform()
- Transformer class adds the combined attributes

```
from sklearn.base import BaseEstimator, TransformerMixin

rooms_ix, bedrooms_ix, population_ix, household_ix = 3, 4, 5, 6

class CombinedAttributesAdder(BaseEstimator, TransformerMixin):
    def __init__(self, add_bedrooms_per_room = True): # no *args or **kargs
        self.add_bedrooms_per_room = add_bedrooms_per_room
    def fit(self, X, y=None):
        return self  # nothing else to do
    def transform(self, X, y=None):
        rooms_per_household = X[:, rooms_ix] / X[:, household_ix]
        population_per_household = X[:, population_ix] / X[:, household_ix]
        if self.add_bedrooms_per_room:
            bedrooms_per_room = X[:, bedrooms_ix] / X[:, rooms_ix]
            return np.c_[X, rooms_per_household, population_per_household,
                         bedrooms_per_room]
        else:
            return np.c_[X, rooms_per_household, population_per_household]

attr_adder = CombinedAttributesAdder(add_bedrooms_per_room=False)
housing_extra_attribs = attr_adder.transform(housing.values)
```

Feature Scaling

- Feature scaling is a step of Data Pre Processing which is applied to independent variables or features of data.
- It basically helps to normalize the data within a particular range. Sometimes, it also helps in speeding up the calculations in an algorithm.
- It has two ways:
  - Min-max scaling
  - Standardization

Feature Scaling

- In Min-max scalingvalues are shifted and rescaled so that they end up ranging from 0 to 1
- This is done by subtracting the min value and dividing by the max minus the min
- Scikit-Learn provides a transformer called MinMaxScaler which has a feature_range hyperparameter that let us to change the range if we don't want 0–1 for some reason

Feature Scaling

- Standardization subtracts the mean value (so standardized values always have a zero mean), and then it divides by the variance so that the resulting distribution has unit variance
- Standardization does not bound values to a specific range, which may be a problem for some algorithms (e.g., neural networks often expect an input value ranging from 0 to 1)but standardization is much less affected by outliers
- Scikit-Learn provides a transformer called StandardScaler for standardization

Transformation Pipelines

- Scikit-Learn provides the Pipeline class to help with sequences of transformations

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

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

housing_num_tr = num_pipeline.fit_transform(housing_num)
```

Transformation Pipelines

- To join two pipelines,Scikit-Learn provides FeatureUnion class
- We give it a list of transformers (which can be entire transformer pipelines).
- When its transform() method is called, it runs each transformer's transform() method in parallel, waits for their output, and then concatenates them and returns the result (and of course calling its fit() method calls each transformer's fit() method)

Transformation Pipelines

- A full pipeline handling both numerical and categorical attributes may look like this

```
from sklearn.pipeline import FeatureUnion

full_pipeline = FeatureUnion(transformer_list=[
        ("num_pipeline", num_pipeline),
        ("cat_pipeline", cat_pipeline),
    ])
```

- We can run the whole pipeline simply

```
>>> housing_prepared = full_pipeline.fit_transform(housing)
>>> housing_prepared
array([[-1.15604281,  0.77194962,  0.74333089, ...,  0.        ,
         0.        ,  0.        ],
       [-1.17602483,  0.6596948 , -1.1653172 , ...,  0.        ,
```

5.Select and Train a Model

- Now we will select and train a Machine Learning model
- First we'll train a Linear Regression model

```
from sklearn.linear_model import LinearRegression

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

Training and Evaluating on the Training Set

- Measure this regression model's RMSE on the whole training set using Scikit-Learn's mean_squared_error function

```
>>> from sklearn.metrics import mean_squared_error
>>> housing_predictions = lin_reg.predict(housing_prepared)
>>> lin_mse = mean_squared_error(housing_labels, housing_predictions)
>>> lin_rmse = np.sqrt(lin_mse)
>>> lin_rmse
68628.198198489219
```

- Most districts' median_housing_values range between $120,000 and 265,000$, so a typical prediction error of \$68,628 is not very satisfying
- This is an example of a model underfitting the training data

Select and Train a Model

- Now we will train a DecisionTreeRegressor.This is a powerful model, capable of finding complex nonlinear relationships in the data

```
from sklearn.tree import DecisionTreeRegressor

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

- We have trained the model so will evaluate it on training set

```
>>> housing_predictions = tree_reg.predict(housing_prepared)
>>> tree_mse = mean_squared_error(housing_labels, housing_predictions)
>>> tree_rmse = np.sqrt(tree_mse)
>>> tree_rmse
0.0
```

Better Evaluation Using Cross-Validation

- Scikit-Learn's cross-validation feature performs K-fold cross-validation
- It randomly splits the training set into 10 distinct subsets called folds, then it trains and evaluates the Decision Tree model 10 times, picking a different fold for evaluation every time and training on the other 9 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)
```

Better Evaluation Using Cross-Validation

- Decision Tree doesn't look as good as it did earlier. In fact, it seems to perform worse than the Linear Regression model
- Cross-validation allows us to get not only an estimate of the performance of our model, but also a measure of how precise this estimate is (i.e., its standard deviation)

Better Evaluation Using Cross-Validation

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

```
>>> lin_scores = cross_val_score(lin_reg, housing_prepared, housing_labels,
...                              scoring="neg_mean_squared_error", cv=10)
...
>>> lin_rmse_scores = np.sqrt(-lin_scores)
>>> display_scores(lin_rmse_scores)
Scores: [ 66760.97371572  66962.61914244  70349.94853401  74757.02629506
  68031.13388938  71193.84183426  64968.13706527  68261.95557897
  71527.64217874  67665.10082067]
Mean: 69047.8379055
Standard deviation: 2735.51074287
```

Better Evaluation Using Cross-Validation

- Now we 'll train the RandomForestRegressor
- Random Forests work by training many Decision Trees on random subsets of the features, then averaging out their predictions.Building a model on top of many other models is called Ensemble Learning

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

6.Fine-Tune Your Model

- Tuning is the process of maximizing a model's performance.Tuning allows you to customize your models so they generate the most accurate outcomes.
- By tuning you change some hyperparameters (for example, the number of trees in a tree-based algorithm or the value of alpha in a linear algorithm), run the algorithm on the data again, then compare its performance on your validation set in order to determine which set of hyperparameters results in the most accurate model
- There are two ways to tune a model
  1. Grid search
  2. Randomized search

Grid search

- Scikit-Learn's GridSearchCV needs hyperparameters that we want to experiment with, and what values to try out, and it will evaluate all the possible combinations of hyperparameter values, using cross-validation
- This approach is fine when we are exploring relatively few combinations

```
from sklearn.model_selection import GridSearchCV

param_grid = [
    {'n_estimators': [3, 10, 30], 'max_features': [2, 4, 6, 8]},
    {'bootstrap': [False], 'n_estimators': [3, 10], 'max_features': [2, 3, 4]},
  ]

forest_reg = RandomForestRegressor()

grid_search = GridSearchCV(forest_reg, param_grid, cv=5,
                           scoring='neg_mean_squared_error')

grid_search.fit(housing_prepared, housing_labels)
```

Randomized Search

- When the hyperparameter search space is large, it is often preferable to use RandomizedSearchCV instead
- It evaluates a given number of random combinations by selecting a random value for each hyperparameter at every iteration

Ensembles Method

- It is another method to tune a model
- Try to combine the models that perform best. The group (or "ensemble") will often perform better than the best individual model

Analyze the Best Models and Their Errors

- Gain good insights on the problem by inspecting the best models
- RandomForestRegressor can indicate the relative importance of each attribute for making accurate predictions
- Display these importance scores next to their corresponding attribute names

```
>>> extra_attribs = ["rooms_per_hhold", "pop_per_hhold", "bedrooms_per_room"]
>>> cat_one_hot_attribs = list(encoder.classes_)
>>> attributes = num_attribs + extra_attribs + cat_one_hot_attribs
>>> sorted(zip(feature_importances, attributes), reverse=True)
[(0.36615898061813418, 'median_income'),
 (0.16478099356159051, 'INLAND'),
 (0.10879295677551573, 'pop_per_hhold'),
 (0.073344235516012421, 'longitude'),
 (0.062909070482620302, 'latitude'),
 (0.056419179181954007, 'rooms_per_hhold'),
 (0.053351077347675809, 'bedrooms_per_room'),
 (0.041143798478729635, 'housing_median_age'),
 (0.014874280890402767, 'population'),
```

Evaluate Your System on the Test Set

- Evaluate the final model on the test set
- Get the predictors and the labels from your test set, run your full_pipeline to transform the data (call transform(), not fit_transform()!), and evaluate the final model on the test set

```
final_model = grid_search.best_estimator_

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

X_test_prepared = full_pipeline.transform(X_test)

final_predictions = final_model.predict(X_test_prepared)

final_mse = mean_squared_error(y_test, final_predictions)
final_rmse = np.sqrt(final_mse)   # => evaluates to 47,766.0
```

7.Launch, Monitor, and Maintain Your System

- Get your solution ready for production (plug into production data inputs, write unit tests, etc.)
- Write monitoring code to check your system's live performance at regular intervals and trigger alerts when it drops
- Retrain your models on a regular basis on fresh data (automate as much as possible)