A First Application: Classifying Iris Species

APT 3025: APPLIED MACHINE LEARNING

Notes taken from Introduction to Machine Learning with Python by Andreas C. Müller & Sarah Guido, O'Reilley Media, 2016

Lecture Overview

- Introduction to machine learning tools
- The iris classification problem
  - Load and inspect the data
  - Choose a learning algorithm
  - Train the classifier
  - Evaluate it

Why Python?

- Python has become the language of choice for many data science applications.
- It combines the power of general-purpose programming languages with the ease of use of domain-specific scripting languages like MATLAB or R.
- Python has libraries for data loading, visualization, statistics, natural language processing, image processing, and more.
- This vast toolbox provides data scientists with a large array of general- and special-purpose functionality.

Notes taken from Introduction to Machine Learning with Python by Andreas C. Müller & Sarah Guido, O'Reilley Media, 2016

Why Python?

- One of the main advantages of using Python is the ability to interact directly with the code, using a terminal or other tools like the Jupyter Notebook, which we'll look at shortly.
- Machine learning and data analysis are fundamentally iterative processes.
- It is essential for these processes to have tools that allow quick iteration and easy interaction.

What is scikit-learn?

- scikit-learn is an open source project, meaning that it is free to use and distribute.
- The scikit-learn project is constantly being developed and improved, and it has a very active user community. It contains a number of state-of-the-art machine learning algorithms, as well as comprehensive documentation about each algorithm.
- scikit-learn is a very popular tool, and the most prominent Python library for machine learning. It is widely used in industry and academia, and a wealth of tutorials and code snippets are available online.

Other Libraries

- In addition to scikit-learn, we will need the following libraries:
  - numpy
  - scipy
  - matplotlib
  - jupyter
  - pandas
  - pillow
- The easiest way to get all these libraries (and many more) is to install Anaconda (www.anaconda.com), a fully featured scientific computing platform.

Jupyter Notebook

- The Jupyter Notebook is an interactive environment for running code in the browser.
- It is a great tool for exploratory data analysis and is widely used by data scientists.
- While the Jupyter Notebook supports many programming languages, we only need the Python support.
- The Jupyter Notebook makes it easy to incorporate code, text, and images

Jupyter Lab

- Jupyterlab is the next generation Jupyter Notebook platform. It adds many improvements and new features, including:
  - Ability to generate a table of contents for ease of navigating the notebook
  - A visual debugger
- To lauch jupyterlab, at the command prompt type
  - jupyter lab

Numpy

- NumPy is one of the fundamental packages for scientific computing in Python.
- It contains functionality for multidimensional arrays, high-level mathematical functions such as linear algebra operations and the Fourier transform, and pseudorandom number generators.

Numpy and scikit-learn

- In scikit-learn, the NumPy array is the fundamental data structure.
- scikit-learn takes in data in the form of NumPy arrays.
- Any data you’re using will have to be converted to a NumPy array.
- The core functionality of NumPy is the ndarray class, a multidimensional (n-dimensional) array.
- All elements of the array must be of the same type.

Creating a Numpy Array

In[1]:

```
import numpy as np

x = np.array([[1, 2, 3], [4, 5, 6]])
print("x:\n{}".format(x))
```

Out[1]:

```
x:
[[1 2 3]
 [4 5 6]]
```

SciPy

- SciPy is a collection of functions for scientific computing in Python. It provides, among other functionality, advanced linear algebra routines, mathematical function optimization, signal processing, special mathematical functions, and statistical distributions.
- scikit-learn draws from SciPy's collection of functions for implementing its algorithms.

Creating Sparse Matrices Using SciPy

- The most important part of SciPy for us is scipy.sparse: this provides sparse matrices, which are another representation that is used for data in scikit-learn.
- Sparse matrices are used whenever we want to store a 2D array that contains mostly zeros:

In[2]:

```
from scipy import sparse

# Create a 2D NumPy array with a diagonal of ones, and zeros everywhere else
eye = np.eye(4)
print("NumPy array:\n{}".format(eye))
```

The Compressed Sparse Row (CSR) Format

Out[2]:

```
NumPy array:
[[ 1.  0.  0.  0.]
 [ 0.  1.  0.  0.]
 [ 0.  0.  1.  0.]
 [ 0.  0.  0.  1.]]
```

In[3]:

```
# Convert the NumPy array to a SciPy sparse matrix in CSR format
# Only the nonzero entries are stored
sparse_matrix = sparse.csr_matrix(eye)
print("\nSciPy sparse CSR matrix:\n{}".format(sparse_matrix))
```

Out[3]:

```
SciPy sparse CSR matrix:
  (0, 0)    1.0
  (1, 1)    1.0
  (2, 2)    1.0
  (3, 3)    1.0
```

The Coordinate (COO) Format

- Usually it is not possible to create dense representations of sparse data (as they would not fit into memory), so we need to create sparse representations directly.
- Here is a way to create the same sparse matrix as before, using the COO format:

In[4]:

```
data = np.ones(4)
row_indices = np.arange(4)
col_indices = np.arange(4)
eye_coo = sparse.coo_matrix((data, (row_indices, col_indices)))
print("COO representation:\n{}".format(eye_coo))
```

Out[4]:

```
COO representation:
  (0, 0)    1.0
  (1, 1)    1.0
  (2, 2)    1.0
  (3, 3)    1.0
```

matplotlib

- matplotlib is the primary scientific plotting library in Python.
- It provides functions for making publication-quality visualizations such as line charts, histograms, scatter plots, and so on.
- Visualizing your data and different aspects of your analysis can give you important insights and provide guidance on needed adjustments

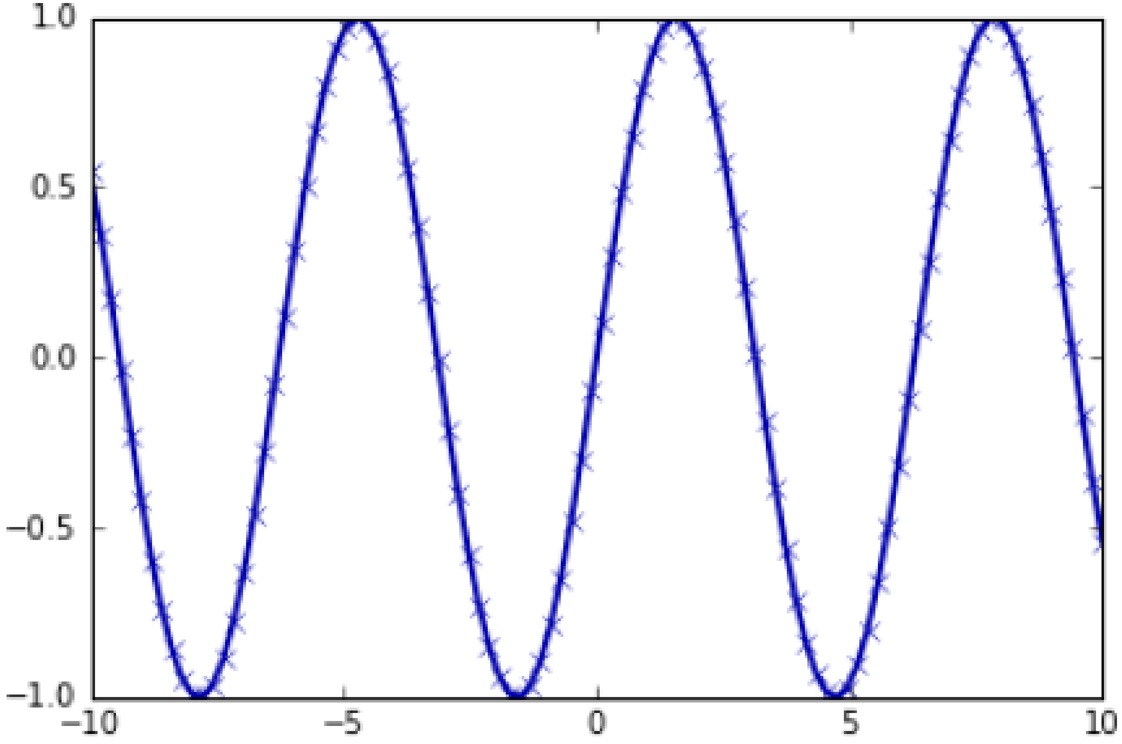
Using matplotlib

In[5]:

```
%matplotlib inline
import matplotlib.pyplot as plt

# Generate a sequence of numbers from -10 to 10 with 100 steps in between
x = np.linspace(-10, 10, 100)
# Create a second array using sine
y = np.sin(x)
# The plot function makes a line chart of one array against another
plt.plot(x, y, marker="x")
```

Output



Notes taken from Introduction to Machine Learning with Python by Andreas C. Müller & Sarah Guido, O'Reilley Media, 2016

pandas

- pandas is a Python library for data wrangling and analysis.
- It is built around a data structure called the DataFrame, which a is a table, similar to an Excel spreadsheet.
- pandas provides methods to modify and operate on this table; in particular, it allows SQL-like queries and joins of tables.

pandas

- In contrast to NumPy, pandas allows each column to have a separate type (for example, integers, dates, floating-point numbers, and strings).
- pandas can ingest from a great variety of file formats and databases, like SQL, Excel files, and comma-separated values (CSV) files.

Using pandas

In[6]:

```
import pandas as pd
from IPython.display import display

# create a simple dataset of people
data = {'Name': ["John", "Anna", "Peter", "Linda"],
        'Location' : ["New York", "Paris", "Berlin", "London"],
        'Age' : [24, 13, 53, 33]
       }

data_pandas = pd.DataFrame(data)
# IPython.display allows "pretty printing" of dataframes
# in the Jupyter notebook
display(data_pandas)
```

	Age	Location	Name
0	24	New York	John
1	13	Paris	Anna
2	53	Berlin	Peter
3	33	London	Linda

Notes taken from Introduction to Machine Learning with Python by Andreas C. Müller & Sarah Guido, O'Reilley Media, 2016

Querying the Table

- There are several possible ways to query the table, for example:

In[7]:

```
# Select all rows that have an age column greater than 30
display(data_pandas[data_pandas.Age > 30])
```

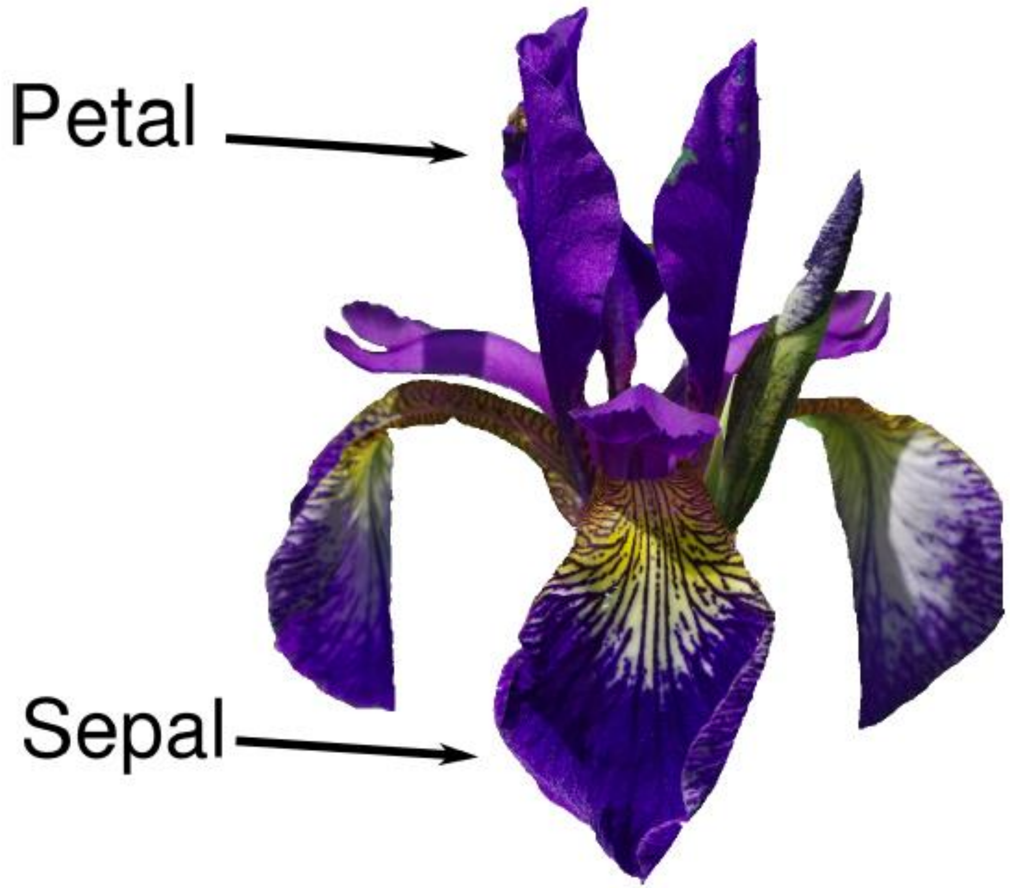
This produces the following result:

	Age	Location	Name
2	53	Berlin	Peter
3	33	London	Linda

The Iris Classification Problem

- We will go through a simple machine learning application and create our first model.
- Let's assume that a hobby botanist is interested in distinguishing the species of some iris flowers that she has found.
- She has collected some measurements associated with each iris: the length and width of the petals and the length and width of the sepals, all measured in centimeters

Parts of the Iris Flower



Notes taken from Introduction to Machine Learning with Python by Andreas C. Müller & Sarah Guido, O'Reilley Media, 2016

A Supervised Learning Problem

- Because we have measurements for which we know the correct species of iris, this is a supervised learning problem.
- In this problem, we want to predict one of several options (the species of iris).
- This is an example of a classification problem.
- The possible outputs (different species of irises) are called classes.

Label

- Every iris in the dataset belongs to one of three classes, so this problem is a three-class classification problem.
- The desired output for a single data point (an iris) is the species of this flower.
- For a particular data point, the species it belongs to is called its label or its class.
- A data point is also called an instance or an example.

The Data

- The data we will use for this example is the Iris dataset, a well-known dataset in machine learning and statistics.
- It is included in scikit-learn in the datasets module.
- We can load it by calling the load_iris function:

In[9]:

```
from sklearn.datasets import load_iris
iris_dataset = load_iris()
```

The Bunch Object

- The iris object that is returned by load_iris is a Bunch object, which is very similar to a dictionary. It contains keys and values:

In[10]:

```
print("Keys of iris_dataset: \n{}".format(iris_dataset.keys()))
```

Out[10]:

```
Keys of iris_dataset:
dict_keys(['target_names', 'feature_names', 'DESCR', 'data', 'target'])
```

Description of the Dataset

- The value of the key DESCR is a short description of the dataset. We show the beginning of the description here:

In[11]:

```
print(iris_dataset['DESCR'][:193] + "\n...")
```

Out[11]:

```
Iris Plants Database
====================

Notes
----
Data Set Characteristics:
    :Number of Instances: 150 (50 in each of three classes)
    :Number of Attributes: 4 numeric, predictive att
...
----
```

Target Names

- The value of the key target_names is an array of strings, containing the species of flower that we want to predict:

In[12]:

```
print("Target names: {}".format(iris_dataset['target_names']))
```

Out[12]:

```
Target names: ['setosa' 'versicolor' 'virginica']
```

Feature Names

- The value of feature_names is a list of strings, giving the description of each feature:

In[13]:

```
print("Feature names: \n{}".format(iris_dataset['feature_names']))
```

Out[13]:

```
Feature names:
['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)',
 'petal width (cm)']
```

Data and Target

- The data itself is contained in the target and data fields. data contains the numeric measurements of sepal length, sepal width, petal length, and petal width in a NumPy array:

In[14]:

```
print("Type of data: {}".format(type(iris_dataset['data'])))
```

Out[14]:

```
Type of data: <class 'numpy.ndarray'>
```

Rows and Columns

- The rows in the data array correspond to flowers, while the columns represent the four measurements that were taken for each flower.
- There are 150 samples (flowers) and each sample is described by four features (measurements).

In[15]:

```
print("Shape of data: {}".format(iris_dataset['data'].shape))
```

Out[15]:

```
Shape of data: (150, 4)
```

Displaying the Data

- Here are the feature values for the first five samples:

In[16]:

```
print("First five rows of data:\n{}".format(iris_dataset['data'][:5]))
```

Out[16]:

```
First five rows of data:
[[ 5.1  3.5  1.4  0.2]
 [ 4.9  3.   1.4  0.2]
 [ 4.7  3.2  1.3  0.2]
 [ 4.6  3.1  1.5  0.2]
 [ 5.   3.6  1.4  0.2]]
```

The Target Array

- The target array contains the species of each of the flowers that were measured, also as a NumPy array:

In[17]:

```
print("Type of target: {}".format(type(iris_dataset['target'])))
```

Out[17]:

```
Type of target: <class 'numpy.ndarray'>
```

Shape of Target Array

- target is a one-dimensional array with one entry per flower.

In[18]:

```
print("Shape of target: {}".format(iris_dataset['target'].shape))
```

Out[18]:

```
Shape of target: (150,)
```

Encoding of Classes

- The species (classes) are encoded as integers from 0 to 2.
- The meanings of the numbers are given by the iris['target_names'] array: 0 means setosa, 1 means versicolor, and 2 means virginica.

In[19]:

```
print("Target:\n{}".format(iris_dataset['target']))
```

Out[19]:

```
Target:
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]
```

Measuring Success

- We want to build a machine learning model from this data that can predict the species of iris for a new set of measurements.
- But before we can apply our model to new measurements, we need to know whether we should trust its predictions.

Generalising

- We cannot use the data we used to build the model to evaluate it.
- This is because our model can always simply remember the whole training set, and will therefore always predict the correct label for any point in the training set.
- This “remembering” does not indicate to us whether our model will generalize well (in other words, whether it will also perform well on new data)

Training and Testing Data

- To assess the model's performance, we show it new data (data that it hasn't seen before) for which we have labels.
- This is usually done by splitting the labeled data we have collected (here, our 150 flower measurements) into two parts.
- One part of the data is used to build our machine learning model, and is called the training data or training set.
- The rest of the data will be used to assess how well the model works; this is called the test data, test set, or hold-out set.

Splitting the Dataset

- scikit-learn contains a function that shuffles the dataset and splits it for you: the train_test_split function.
- This function extracts 75% of the rows in the data as the training set, together with the corresponding labels for this data.
- The remaining 25% of the data, together with the remaining labels, is declared as the test set.

Splitting the Dataset

- In scikit-learn, data is usually denoted with a capital X, while labels are denoted by a lowercase y.
- Setting the value of random_state makes it possible to repeat the experiment exactly.

In[20]:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    iris_dataset['data'], iris_dataset['target'], random_state=0)
```

The Shapes of the Resulting Datasets

- The output of the train_test_split function is X_train, X_test, y_train, and y_test, which are all NumPy arrays.
- X_train contains 75% of the rows of the dataset, and X_test contains the remaining 25%.
- The shape of X_train is (112, 4)
- The shape of y_train is (112,)
- The shape of X_test is (38, 4)
- The shape of y_test is (38,)

Inspecting the Data

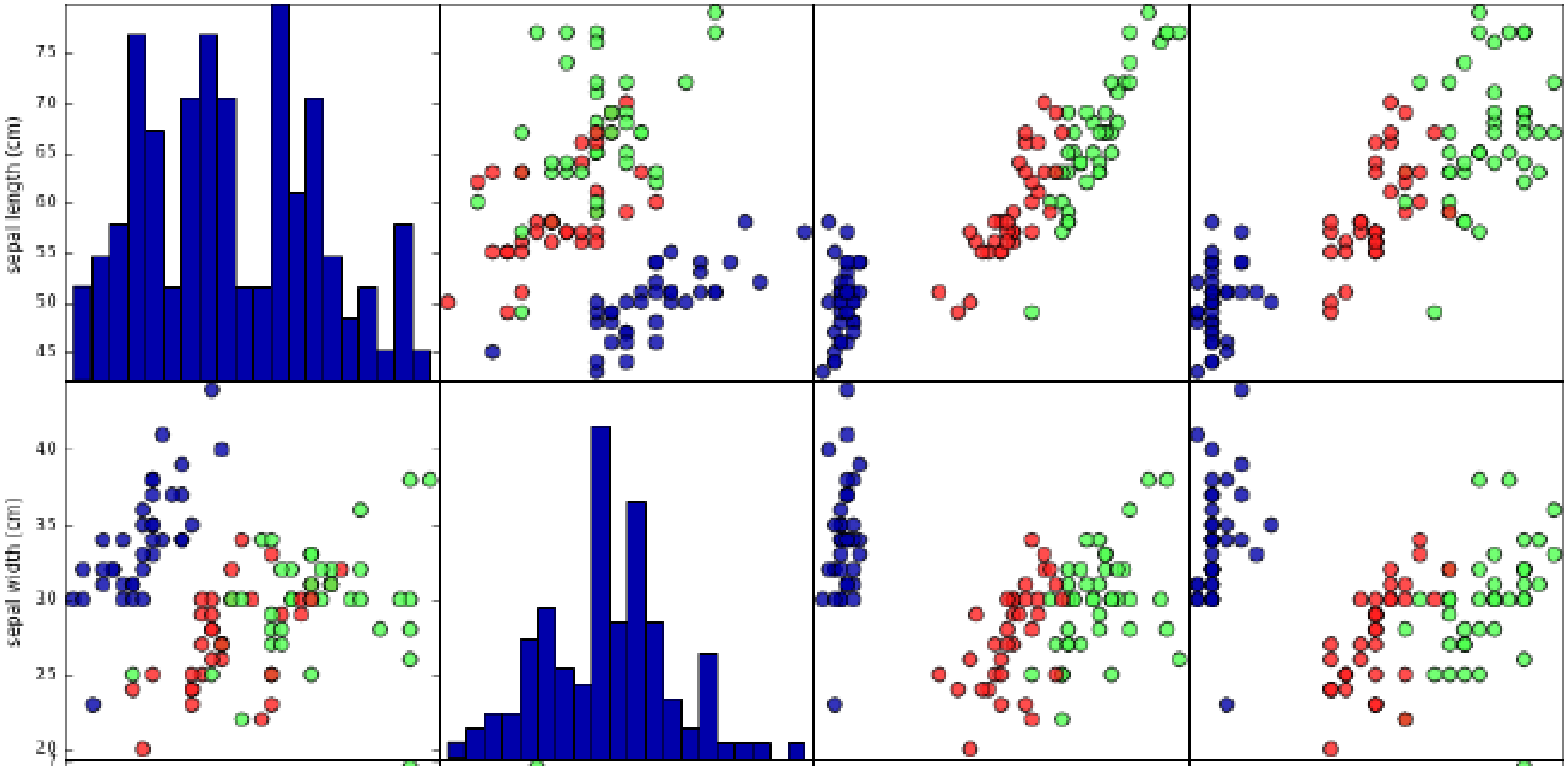
- Before building a machine learning model it is often a good idea to inspect the data, to see if the task is easily solvable without machine learning, or if the desired information might not be contained in the data.
- We can inspect the data by visualizing it.
- One form of visualization is a scatter plot.
- Since a scatter plot is only possible for two variables, we will generate several pair plots.

Using pandas to Create Visualizations

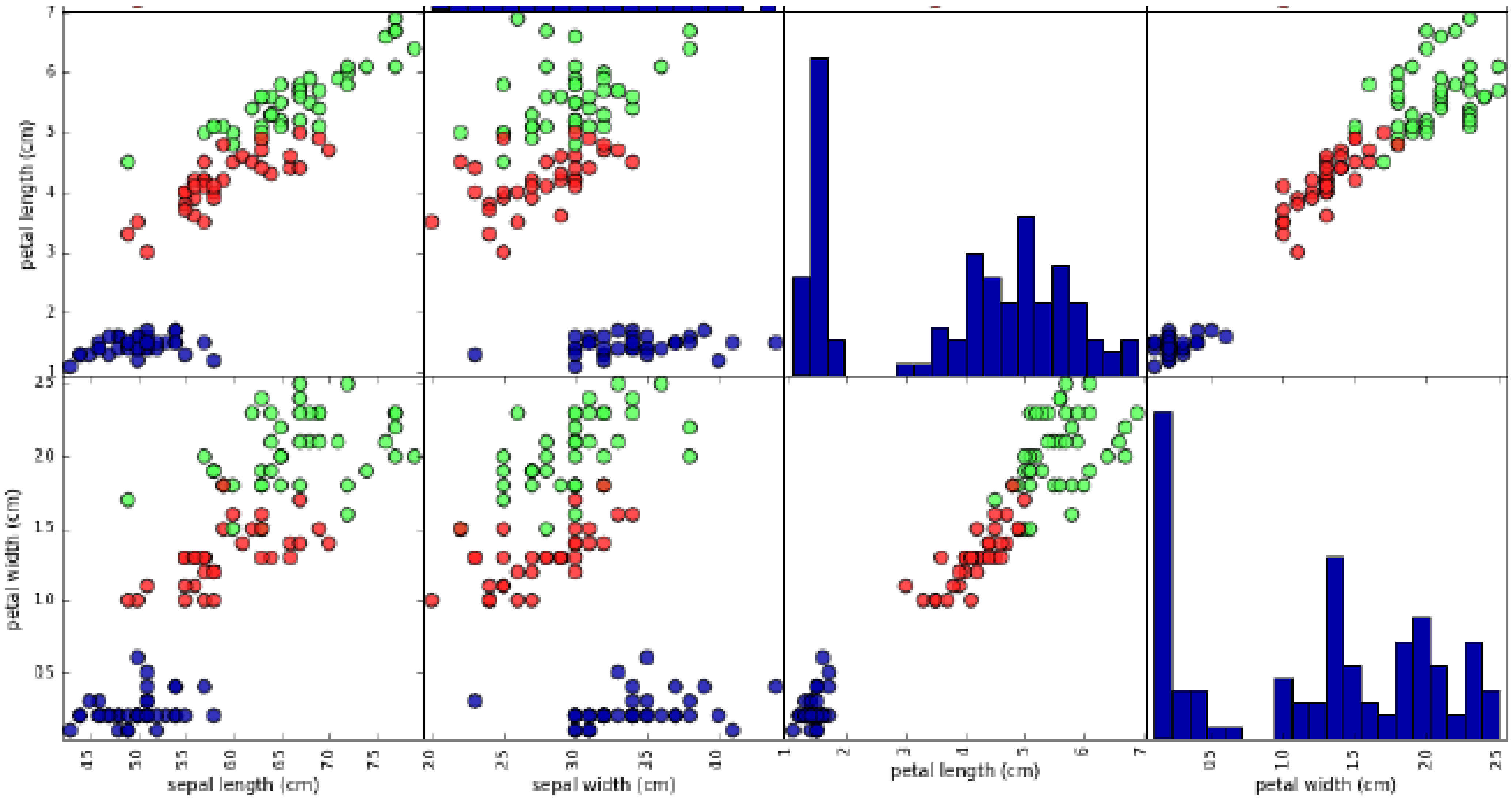
- To create the plot, we first convert the NumPy array into a pandas DataFrame. pandas has a function to create pair plots called scatter_matrix. The diagonal of this matrix is filled with histograms of each feature:

In[23]:

```
# create dataframe from data in X_train
# label the columns using the strings in iris_dataset.feature_names
iris_dataframe = pd.DataFrame(X_train, columns=iris_dataset.feature_names)
# create a scatter matrix from the dataframe, color by y_train
pd.plotting.scatter_matrix(iris_dataframe, c=y_train, figsize=(15, 15),
                           marker='o', hist_kwds={'bins': 20}, s=60,
                           alpha=.8, cmap=mglearn.cm3)
```



Notes taken from Introduction to Machine Learning with Python by Andreas C. Müller & Sarah Guido, O'Reilley Media, 2016



Notes taken from Introduction to Machine Learning with Python by Andreas C. Müller & Sarah Guido, O'Reilley Media, 2016

Interpreting the Plots

- The above figure is a pair plot of the features in the training set.
- The data points are colored according to the species the iris belongs to (i.e., the class).
- To create the plot, we first convert the NumPy array into a pandas DataFrame. pandas has a function to create pair plots called scatter_matrix.
- The diagonal of this matrix is filled with histograms of each feature.
- We can see that the three classes seem to be relatively well separated using the sepal and petal measurements. This means that a machine learning model will likely be able to learn to separate them.

Building a k-Nearest Neighbors (kNN) Model

- There are many classification algorithms in scikit-learn that we could use.
- Here we will use a k-nearest neighbors classifier, which is easy to understand.
- Building this model only consists of storing the training set.
- To make a prediction for a new data point, the algorithm finds the point in the training set that is closes to the new point.
- Then it assigns the label of this training point to the new data point.

The k in kNN

- The k in k-nearest neighbors signifies that instead of using only the closest neighbor to the new data point, we can consider any fixed number k of neighbors in the training (for example, the closest three or five neighbors).
- Then, we can make a prediction using the majority class among these neighbors.
- For now we will use a single neighbour.

Implementing kNN with scikit-learn

- All machine learning models in scikit-learn are implemented in their own classes which are called Estimator classes.
- The k-nearest neighbors classification algorithm is implemented in the kNeighborsClassifier class in the neighbors module.

Implementing kNN with scikit-learn

- Before we can use the model, we need to instantiate the class into an object.
- This is when we will set any parameters of the model. The most important parameter of KNeighborsClassifier is the number of neighbors, which we will set to 1.

In[24]:

```
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=1)
```

The Classifier Object

- The knn object encapsulates the algorithm that will be used to build the model from the training data, as well as the algorithm to make predictions on new data points.
- It will also hold the information that the algorithm has extracted from the training data.
- In the case of kNeighborsClassifier, it will just store the training set.

Building the Model

- To build the model on the training set, we call the fit method of the knn object, which takes as arguments the NumPy array X_train containing the training data and the NumPy array y_train of the corresponding training labels:

In[25]:

```
knn.fit(X_train, y_train)
```

Out[25]:

```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
           metric_params=None, n_jobs=1, n_neighbors=1, p=2,
           weights='uniform')
```

Parameters used in building the model

- The fit method returns the knn object itself (and modifies it in place), so we get a string representation of our classifier.
- The representation shows us which parameters were used in creating the model, including the one we set, n_neighbors=1.

Making Predictions

- We can now make predictions using this model on new data for which we might not know the correct labels.
- Imagine we found an iris in the wild with a sepal length of 5 cm, a sepal width of 2.9 cm, a petal length of 1 cm, and a petal width of 0.2 cm.
- What species of iris would this be?
- We can put this data into a NumPy array, again by calculating the shape--that is, the number of samples (1) by the number of features (4).

New Sample for Prediction

- Note that we made the measurements of this single flower into a row in a two dimensional NumPy array, as scikit-learn always expects two-dimensional arrays for the data.

In[26]:

```
X_new = np.array([[5, 2.9, 1, 0.2]])
print("X_new.shape: {}".format(X_new.shape))
```

Out[26]:

```
X_new.shape: (1, 4)
```

Making a Prediction

- To make a prediction, we can call the predict method of the knn object:

In[27]:

```
prediction = knn.predict(X_new)
print("Prediction: {}".format(prediction))
print("Predicted target name: {}".format(
       iris_dataset['target_names'][prediction]))
```

Out[27]:

```
Prediction: [0]
Predicted target name: ['setosa']
```

Evaluating the Model

- This is where the test set that we created earlier comes in.
- This data was not used to build the model, but we know what the correct species is for each iris in the test set.
- Therefore, we can make a prediction for each iris in the test data and compare it against its label (the known species).
- We can measure how well the model works by computing the accuracy, which is the fraction of flowers for which the right species was predicted.

Evaluating the Model

In[28]:

```
y_pred = knn.predict(X_test)
print("Test set predictions:\n {}".format(y_pred))
```

Out[28]:

```
Test set predictions:
 [2 1 0 2 0 2 0 1 1 1 2 1 1 1 1 0 1 1 0 0 2 1 0 0 2 0 0 1 1 0 2 1 0 2 2 1 0 2]
```

In[29]:

```
print("Test set score: {:.2f}".format(np.mean(y_pred == y_test)))
```

Out[29]:

```
Test set score: 0.97
```

Evaluating the Model

- We can also use the score method of the knn object, which will compute the test set accuracy for us:

In[30]:

```
print("Test set score: {:.2f}".format(knn.score(X_test, y_test)))
```

Out[30]:

```
Test set score: 0.97
```