Guía de Laboratorio – Introducción a Python

Sistema de conocimiento:

- Introducción a Python
- Introducción a las bibliotecas y herramientas empleadas en Scikit-learn.

Objetivos:

- Practicar los elementos fundamentales del lenguaje Python a través de la solución de ejercicios.
- Introducir las bibliotecas y herramientas que serán empleadas en conjunto con Scikit-learn para realizar el aprendizaje automático.

Bibliografía:

- McKinney, W. (2022). *Python for data analysis*. O'Reilly Media, Inc.
- Müller, A., & Guido, S. A. (2016). *Introduction to Machine Learning with Python. O'Reilly Media*.

Introducción

Python se ha convertido en la lingua franca de muchas aplicaciones de ciencia de datos. Combina la potencia de los lenguajes de programación de propósito general con la facilidad de uso de lenguajes de programación específicos como MATLAB o R. Python dispone de bibliotecas para la carga de datos, visualización, estadística, procesamiento del lenguaje natural, procesamiento de imágenes y mucho más. Este amplio conjunto de herramientas proporciona a los científicos de datos una gran variedad de funciones generales y especiales. Una de las principales ventajas de utilizar Python es la posibilidad de interactuar directamente con el código, utilizando un terminal u otras herramientas como el Jupyter Notebook, que veremos en breve. El aprendizaje automático y el análisis de datos son procesos fundamentalmente iterativos, en los que los datos dirigen el análisis. Para estos procesos contar con herramientas que permitan una iteración rápida y una interacción sencilla(Müller & Guido, 2016).

scikit-learn es un proyecto de código abierto, lo que significa que es libre de usar y distribuir, y cualquiera puede obtener fácilmente el código fuente para ver lo que ocurre entre bastidores. El proyecto *scikit-learn* se desarrolla y mejora constantemente, y cuenta con una comunidad de usuarios muy activa. Contiene una serie de algoritmos de aprendizaje automático de última generación, así como documentación exhaustiva sobre cada algoritmo(Müller & Guido, 2016).

En el presente laboratorio practicaremos como ejecutar programas sencillos en el lenguaje Python a través de la solución de ejercicios de mediana complejidad. De igual manera realizaremos una introducción a la biblioteca de Scikit-learn y veremos un ejemplo sencillo de su uso.

A continuación debe seguir los pasos que propone la siguiente guía:

Guía

Instalación de Python

Para instalar Python en Ubuntu puede seguir los pasos del siguiente video de Youtube:

- ¿Cómo instalar la última versión de Python en Ubuntu?

Para instalarlo en Python pueden seguir los pasos del siguiente video:

- ¿Cómo Instalar Python en Windows 10 en 2023?

Para usar una versión en línea de un intérprete de Python puede usarse la herramienta Jupyter Notebook de Google Colab.

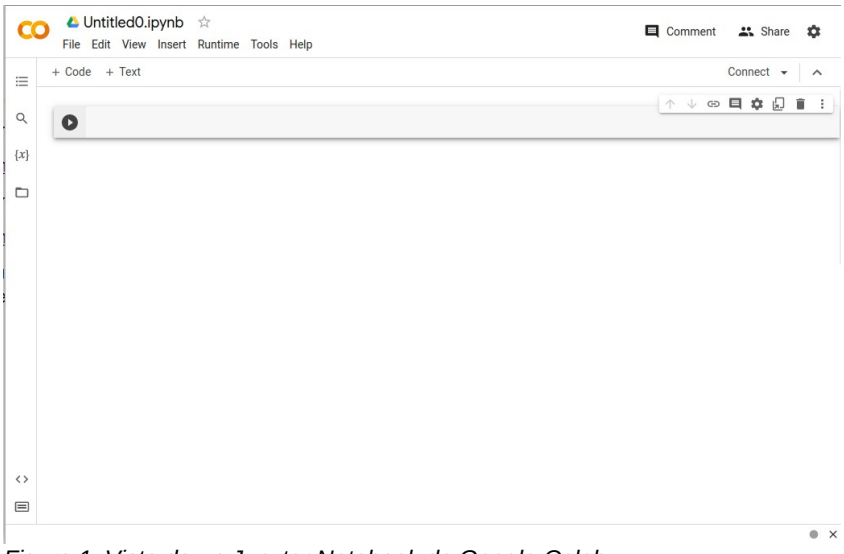

Figura 1: Vista de un Jupyter Notebook de Google Colab.

Para realizar un estudio de los elementos fundamentales del lenguaje puede usar la documentación oficial del lenguaje en la siguiente dirección: https://docs.python.org/es/3/

Instalación de Scikit-learn

scikit-learn depende de otros dos paquetes de Python, NumPy y SciPy. Para el trazado y el desarrollo interactivo, también debe instalar matplotlib, IPython y Jupyter Notebook. Si ya tienes instalado Python, puedes usar pip para instalar todos estos paquetes:

```
pip install numpy scipy matplotlib ipython scikit-learn pandas
```

Bibliotecas y herramientas esenciales

Entender qué es scikit-learn y cómo usarlo es importante, pero hay algunas otras bibliotecas que mejorarán su experiencia. scikit-learn está construido sobre las bibliotecas científicas de Python NumPy y SciPy. Además de NumPy y SciPy, utilizaremos pandas y matplotlib. También presentaremos Jupyter Notebook, un entorno de programación interactivo basado en navegador. Brevemente, esto es lo que debe saber acerca de estas herramientas con el fin de obtener el máximo provecho de scikit-learn(Müller & Guido, 2016).

Jupyter Notebook

Jupyter Notebook es un entorno interactivo para ejecutar código en el navegador. Es una gran herramienta para el análisis exploratorio de datos y es ampliamente utilizado por los científicos de datos. Aunque el Jupyter Notebook soporta muchos lenguajes de programación, sólo necesitamos Python. El Jupyter Notebook facilita la incorporación de código, texto e imágenes, y todo este libro fue escrito como un Jupyter Noteboo(Müller & Guido, 2016).

NumPy

NumPy es uno de los paquetes fundamentales para la computación científica en Python. Contiene funciones para matrices multidimensionales, funciones matemáticas de alto nivel, como operaciones de álgebra lineal y la transformada de Fourier, y funciones pseudoaleatorias.

En *scikit-learn*, el array NumPy es la estructura de datos fundamental. *scikit-learn* toma los datos en forma de arrays NumPy. Cualquier dato que esté utilizando tendrá que ser convertido a un array NumPy. La funcionalidad central de NumPy es la clase ndarray, un array multidimensional (n-dimensional). Todos los elementos del array deben ser del mismo tipo. Un array NumPy tiene este aspecto:

In[2]:

```
import numpy as np

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

Out[2]:

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

SciPy

SciPy es una colección de funciones para la computación científica en Python. Proporciona, entre otras funcionalidades, rutinas avanzadas de álgebra lineal, optimización de funciones matemáticas, procesamiento de señales, funciones matemáticas especiales y distribuciones estadísticas. *scikit-learn* utiliza la colección de funciones de SciPy para implementar sus algoritmos. La parte más importante de SciPy es scipy.sparse: proporciona matrices dispersas, que son otra representación que se utiliza para los datos en scikitlearn. Las matrices dispersas se utilizan cuando queremos almacenar una matriz 2D que contiene en su mayoría ceros:

In[3]:

```
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))
```

Out[3]:

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

In[4]:

```
# 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[4]:

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

matplotlib

matplotlib es la principal biblioteca de gráficos científicos de Python. Proporciona funciones para realizar visualizaciones con calidad de publicación, como gráficos de líneas, histogramas, diagramas de dispersión, etc. La visualización de los datos y los diferentes aspectos de su análisis puede darle una visión importante, y vamos a utilizar matplotlib para todas nuestras visualizaciones. Cuando trabajes dentro del Jupyter Notebook, puedes mostrar figuras directamente en el navegador usando los comandos %matplotlib notebook y %matplotlib inline. Se recomienda usar %matplotlib notebook, que proporciona un entorno interactivo (aunque nosotros usamos %matplotlib notebook). Por ejemplo el siguiente código genera la Figura 2.

```
In[6]:

    %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")
```

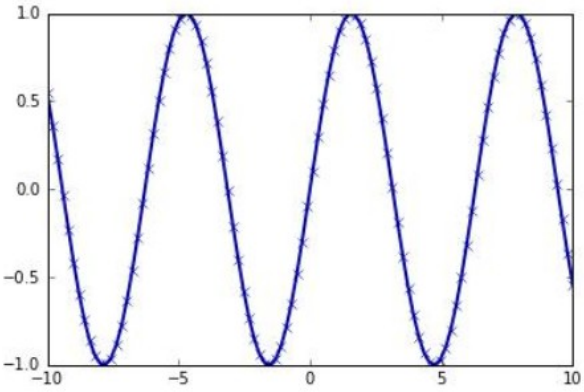

Figura 2: Gráfico simple de la función seno creada con matplotlib.

Puede encontrar toda una biblioteca de ejemplos de construcción de gráficos con matplotlib en la siguiente dirección: https://matplotlib.org/stable/gallery/index.html

Pandas

pandas es una biblioteca de Python para la gestión y el análisis de datos. Está construida en torno a una estructura de datos llamada DataFrame que sigue el modelo de R DataFrame. En pocas palabras, un DataFrame de *pandas* es una tabla, similar a una hoja de cálculo de Excel. *pandas*

proporciona una gran variedad de métodos para modificar y operar sobre esta tabla; en particular, permite consultas y uniones de tablas tipo SQL. A diferencia de NumPy, que requiere que todas las entradas de un array sean del mismo tipo, *pandas* permite que cada columna tenga un tipo distinto (por ejemplo, enteros, fechas, flotantes, etc.). Otra herramienta valiosa que proporciona *pandas* es su capacidad de ingesta desde una gran variedad de formatos de archivo y bases de datos, como SQL, archivos Excel y archivos de valores separados por comas (CSV)(Müller & Guido, 2016).

Para profundizar en el estudio de esta biblioteca puede leer el capítulo 5 del libro Python for Data Analysis (McKinney, 2022).

Aquí va un ejemplo de como crear un DataFrame de *pandas* a partir de un diccionario:

In[7]:

```
import pandas as pd

# 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)
```

Esto produce la siguiente salida:

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

Luego hay muchas maneras de realizar consultas sobre esta tabla. Por ejemplo:

In[8]:

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

produce la siguiente salida:

	Age	Location	Name
2	53	Berlin	Peter
3	33	London	Linda

Scikit-learn

A continuación veremos un ejemplo de un proceso de aprendizaje automático usando scikit-learn, y, por tanto, crearemos nuestro primer modelo. En el proceso introduciremos algunos términos y conceptos básicos. Utilizaremos un conjunto de datos muy famoso, denominado Iris Dataset; este dataset consiste en un conjunto de 150 instancias de orquídeas de las que se conoce la longitud y la anchura de los pétalos y la longitud y la anchura de los sépalos, todo ello medidos en centímetros, y además estas están clasificadas en 3 tipos fundamentales dependiendo de la especia a la que pertenece: *setosa*, *versicolor*, o *virginica.*

Nuestro objetivo es construir un modelo de aprendizaje automático que pueda aprender de las mediciones de estos iris cuya especie se conoce, de modo que podamos predecir la especie para un nuevo iris.

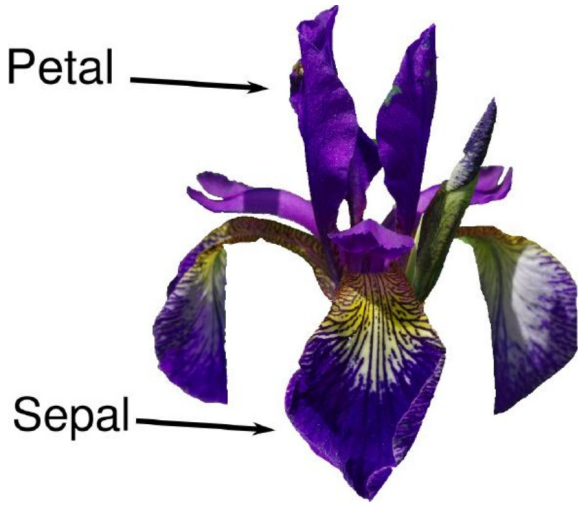

Figura 3: Partes de una flor de iris. Fuente:(Müller & Guido, 2016)

Debido a que poseemos mediciones para la que está establecida la especie de la flor, este es un problema de *aprendizaje supervisado*. Además, en este problema queremos predecir una de varias opciones de especies de iris, por lo que es un problema de *clasificación*. También conocemos que cada individuo pertenece solamente a una de las especies, así que este es un *problema de clasificación de tres clases*.

Conociendo los datos

Este conjunto de datos clásico está incluido en el módulo de datasets de scikit-learn. Puede ser cargado mediante la llamada a la función *load_iris*:

In[10]:

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

El objeto iris que es retornado por la función llamada es un objeto Bunch, que es muy similar a un diccionario. Este contiene llaves y valores:

In[11]:

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

Out[11]:

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

El valor de la llave DESCR es una descripción breve del conjunto de datos.

In[12]:

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

Out[12]:

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

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

El valor de la llave target_names es un arreglo de cadenas, conteniendo las especies de flores que queremos predecir.

In[13]:

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

Out[13]:

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

El valor de feature_names es una lista de cadenas, dando la descripción de cada característica.

In[14]:

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

Out[14]:

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

Los datos en si están contenidos en los campos target y data. Data contiene las medidas numéricas de la longitud de sepal, ancho de sepal, longitud de pétalo y ancho de pétalo en un arreglo de Numpy:

In[15]:

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

Out[15]:

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

Las filas en el arreglo de datos se corresponden con flores, mientras que las columnas representan las medidas tomadas para cada flor.

In[16]:

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

Out[16]:

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

Podemos ver que el arreglo contiene mediciones para 150 flores diferentes. Recuerda que cada individuo es llamado muestra (*sample*) en el aprendizaje automático, y sus propiedades son llamados *features*. La forma del arreglo de datos es el numero de muestras multiplicado por el número de características. Esta es una convención en *scikit-learn*, y tus datos deben siempre asumir que se encuentran en esta forma. Aquí está el valor de las características de las primeras 5 muestras:

In[17]:

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

Out[17]:

```
First five columns 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]]
```

El arreglo target contiene las especies de cada flor que fueron medidas, igualmente como un arreglo Numpy:

In[18]:

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

Out[18]:

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

El target es un arreglo de una dimensión, una entrada por cada flor.

In[19]:

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

Out[19]:

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

Las especies están codificadas como un entero de 0 a 2.

In[20]:

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

Out[20]:

```
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 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]
```

El significado de los números está dado por el arreglo iris['target_names']:0 significa setosa, 1 significa versicolor y 2 significa virginica.

Midiendo el éxito: Datos de prueba y entrenamiento

Queremos construir un modelo de aprendizaje automático a partir de estos datos que pueda predecir las especies de iris para un nuevo conjunto de mediciones.Pero antes de poder aplicar nuestro modelo a nuevas mediciones, necesitamos saber si realmente funciona, es decir, si debemos confiar en sus predicciones.

Lamentablemente, no podemos utilizar los datos con los que construimos el modelo para evaluarlo. Esto se debe a que nuestro modelo siempre puede simplemente recordar todo el conjunto de entrenamiento y, por lo tanto, siempre predecirá la etiqueta correcta para cualquier punto del conjunto de entrenamiento. Este "recordar" no nos indica si nuestro modelo generalizará bien (en otras palabras, si también funcionará bien con datos nuevos).

Para evaluar el rendimiento del modelo, le mostramos datos nuevos (datos que no ha visto antes) para los que tenemos etiquetas). Esto suele hacerse dividiendo en dos partes los datos (en este caso, nuestras 150 mediciones de flores). Una parte de los datos se utiliza para construir nuestro modelo de aprendizaje automático y se denomina conjunto de entrenamiento. El resto de los datos se utilizará para evaluar el funcionamiento del modelo; se denominan datos de prueba, conjunto de prueba o conjunto de espera.

scikit-learn contiene una función que baraja el conjunto de datos y lo divide por usted: la función train_test_split. Esta función extrae el 75% de las filas de los datos como el conjunto de entrenamiento, junto con las etiquetas correspondientes a estos datos. El 25% restante de los datos, junto con las etiquetas restantes, se declara conjunto de prueba. Decidir cuántos datos se van a

incluir en el conjunto de entrenamiento y en el de prueba es en cierto modo arbitrario, pero utilizar un conjunto de prueba no es una decisión fácil. pero una buena regla general es utilizar un conjunto de prueba que contenga el 25% de los datos.

En *scikit-learn*, los datos se denotan generalmente con una X mayúscula, mientras que las etiquetas se denotan una y minúscula. Esto se inspira en la formulación estándar f(x)=y en matemáticas, donde x es la entrada de una función e y es la salida. Siguiendo más convenciones matemáticas, utilizamos una X mayúscula porque los datos son una matriz bidimensional (una y minúscula porque el objetivo es una matriz unidimensional (un vector).

Llamemos a train_test_split en nuestros datos y asignemos las salidas usando esta nomenclatura:

In[21]:

```
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)
```

Antes de realizar la división, la función train_test_split baraja el conjunto de datos utilizando un generador de números pseudoaleatorios. Si sólo tomáramos el último 25% de los datos como conjunto de prueba, todos los puntos de datos tendrían la etiqueta 2, ya que los puntos de datos están ordenados por la etiqueta (véase la salida para iris['target'] mostrada anteriormente). El uso de un conjunto de prueba que sólo contenga una de las tres clases no nos diría mucho sobre lo bien que generaliza nuestro modelo, así que barajamos nuestros datos para asegurarnos de que los datos de prueba contienen datos de todas las clases.

Para asegurarnos de que obtendremos el mismo resultado si ejecutamos la misma función varias veces, proporcionamos al generador de números pseudoaleatorios una semilla fija mediante el parámetro random_state. Esto hará que el resultado determinista, por lo que esta línea siempre tendrá el mismo resultado. Siempre fijaremos el random_state de esta manera cuando procedimientos aleatorios en este curso.

La salida de la función train_test_split es X_train, X_test, y_train, y y_test, que son matrices NumPy. X_train contiene el 75% de las filas del conjunto de datos, y X_test contiene el 25% restante:

In[22]:

```
print("X_train shape: {}".format(X_train.shape))
print("y_train shape: {}".format(y_train.shape))
```

Out[22]:

```
X_train shape: (112, 4)
y_train shape: (112,)
```

In[23]:

```
print("X_test shape: {}".format(X_test.shape))
print("y_test shape: {}".format(y_test.shape))
```

Out[23]:

```
X_test shape: (38, 4)
y_test shape: (38,)
```

Primero miremos nuestros datos

Antes de construir un modelo de aprendizaje automático, suele ser buena idea inspeccionar los datos, para ver si la tarea puede resolverse fácilmente sin aprendizaje automático, o si la información deseada podría no estar contenida en los datos. Además, inspeccionar los datos es una buena forma de encontrar anormalidades y peculiaridades. Tal vez algunos de sus iris se midieron en pulgadas y no en centímetros, por ejemplo. En el mundo real, las incoherencias en los datos y las mediciones inesperadas son muy comunes.

Una de las mejores formas de inspeccionar los datos es visualizarlos. Una forma de hacerlo es mediante un gráfico de dispersión. Un gráfico de dispersión de los datos coloca una característica a lo largo del eje x y otra a lo largo del eje y, y dibuja un punto para cada punto de datos. Por desgracia, las pantallas de ordenador sólo tienen dos dimensiones, lo que nos permite representar sólo dos (o quizá tres) características a la vez. De este modo, es difícil representar conjuntos de datos con más de tres características. Una forma de evitar este problema es realizar un gráfico de pares, que examina todos los pares posibles de características. Si tiene un número pequeño de características, como las cuatro que tenemos aquí, esto es bastante razonable. Sin embargo, hay que tener en cuenta que un gráfico de pares no muestra la interacción de todas las características a la vez, por lo que algunos aspectos interesantes de los datos pueden no revelarse al visualizarlos de esta forma. algunos aspectos interesantes de los datos.

La Figura 4 es un gráfico de pares de las características del conjunto de entrenamiento. Los puntos de datos están coloreados según la especie a la que pertenece el iris. Para crear el gráfico, primero convertimos la matriz NumPy en un DataFrame de pandas. *pandas* tiene una función para crear gráficos de pares llamada scatter_matrix. La diagonal de esta matriz se rellena con histogramas de cada característica:

In[24]:

```
# 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
grr = pd.scatter_matrix(iris_dataframe, c=y_train, figsize=(15, 15), marker='o',
                        hist_kwds={'bins': 20}, s=60, alpha=.8, cmap=mglearn.cm3)
```

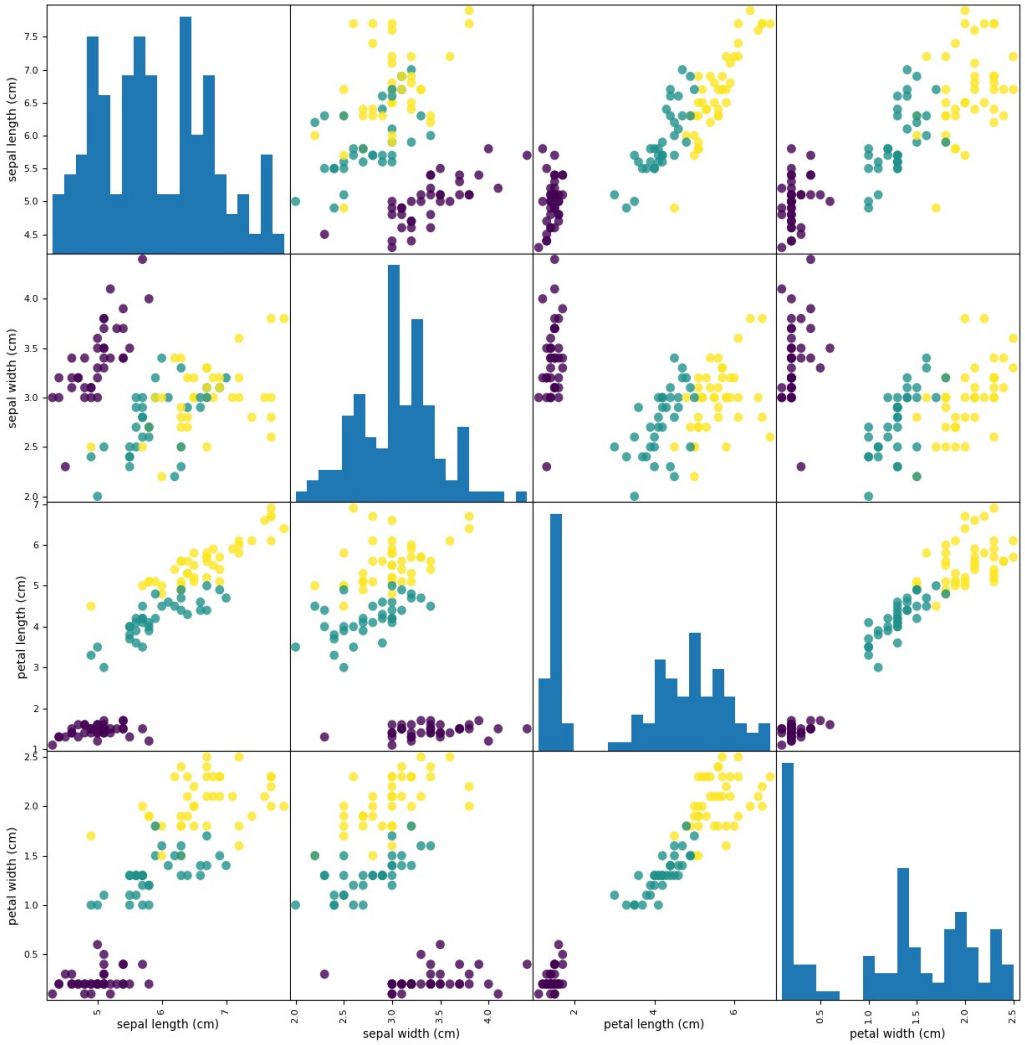

Figura 4: Gráfico de pares del conjunto de datos Iris, coloreado por etiqueta de clase.

A partir de los gráficos, podemos ver que las tres clases parecen estar relativamente bien separadas utilizando las medidas de los sépalos y los pétalos. Esto significa que un modelo de aprendizaje automático será capaz de aprender a separarlas.

Construyendo tu primer modelo: K-Vecinos más cercanos

Ahora podemos empezar a construir el modelo de aprendizaje automático real. Hay muchos algoritmos de clasificación en scikit-learn que podríamos utilizar. Aquí vamos a utilizar un clasificador k-vecinos más cercanos, que es fácil de entender. Construir este modelo sólo consiste en almacenar el conjunto de entrenamiento. Para hacer una predicción para un nuevo punto de datos, el algoritmo

encuentra el punto en el conjunto de entrenamiento que está más cerca del nuevo punto. A continuación, asigna la etiqueta de este punto de entrenamiento al nuevo punto de datos.

La k en k-vecinos más cercanos significa que, en lugar de utilizar sólo el vecino más cercano al nuevo punto de datos, podemos considerar cualquier número fijo k de vecinos en el entrenamiento (por ejemplo, los tres o cinco vecinos más cercanos). A continuación, podemos hacer una predicción utilizando la clase mayoritaria entre estos vecinos.

Todos los modelos de aprendizaje automático en scikit-learn se implementan en sus propias clases, que se llaman clases Estimator. El algoritmo de clasificación k-nearest neighbors se implementa en la clase KNeighborsClassifier en el módulo neighbors. Antes de poder utilizar el modelo, necesitamos instanciar la clase en un objeto. Es entonces cuando estableceremos los parámetros del modelo. El parámetro más importante del clasificador K-neighbors es el número de vecinos, que estableceremos en 1:

```
In[25]:

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

El objeto knn encapsula el algoritmo que se utilizará para construir el modelo a partir de los datos de entrenamiento, así como el algoritmo para realizar predicciones sobre nuevos puntos de datos. También contendrá la información que el algoritmo haya extraído de los datos de entrenamiento. En el caso de KNeighborsClassifier, sólo almacenará el conjunto de entrenamiento.

Para construir el modelo sobre el conjunto de entrenamiento, llamamos al método fit del objeto knn, que toma como argumentos la matriz NumPy X_train que contiene los datos de entrenamiento y la matriz NumPy y_train de las etiquetas de entrenamiento correspondientes:

```
In[26]:

    knn.fit(X_train, y_train)

Out[26]:

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

El método fit devuelve el propio objeto knn (y lo modifica en su lugar), por lo que obtenemos una representación en forma de cadena de nuestro clasificador. La representación nos muestra qué parámetros se utilizaron para crear el modelo. Casi todos ellos son los valores por defecto, pero también se puede encontrar n_neighbors=1, que es el parámetro que pasamos. La mayoría de los modelos en *scikit-learn* tienen muchos parámetros, pero la mayoría de ellos son optimizaciones de velocidad o para casos de uso muy especiales. Usted no tiene que preocuparse por los otros parámetros que se muestran en esta representación. La impresión de un modelo scikit-learn puede producir cadenas muy largas, pero no se deje intimidar por ellas.

Realizando predicciones

Ahora podemos hacer predicciones con este modelo sobre nuevos datos de los que quizá no conozcamos las etiquetas correctas. Imaginemos que encontramos un iris en la naturaleza con una longitud de sépalo de 5 cm, una anchura del sépalo de 2,9 cm, una longitud del pétalo de 1 cm y una anchura del pétalo de 0,2 cm. ¿De qué especie de iris se trataría? Podemos poner estos datos en un array NumPy, de nuevo calculando la forma, es decir, el número de muestras (1) multiplicado por el número de características (4):

In[27]:

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

Out[27]:

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

Tenga en cuenta que hicimos las mediciones de esta única flor en una fila en una matriz bidimensional NumPy, como scikit-learn siempre espera matrices bidimensionales para los datos.

Para hacer una predicción, llamamos al método predict del objeto knn:

In[28]:

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

Out[28]:

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

Nuestro modelo predice que este nuevo iris pertenece a la clase 0, lo que significa que su especie es setosa. Pero, ¿cómo sabemos si podemos fiarnos de nuestro modelo? No sabemos cuál es la especie correcta de esta muestra, ¡que es para lo que hemos construido el modelo!

Evaluando el modelo

Aquí es donde entra en juego el conjunto de pruebas que creamos anteriormente. Estos datos no se usaron para construir el modelo, pero sabemos cuál es la especie correcta para cada iris en el conjunto de prueba.

Por tanto, podemos hacer una predicción para cada iris de los datos de prueba y compararla con su etiqueta (la especie conocida). Podemos medir la eficacia del modelo calculando la precisión, que es la fracción de flores para las que se predijo la especie correcta:

In[29]:

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

Out[29]:

```
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[30]:

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

Out[30]:

```
Test set score: 0.97
```

También podemos utilizar el método de puntuación del objeto knn, que calculará por nosotros la precisión del conjunto de prueba:

In[31]:

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

Out[31]:

```
Test set score: 0.97
```

Para este modelo, la precisión del conjunto de pruebas es de aproximadamente 0,97, lo que significa que hicimos la predicción correcta para el 97% de los iris del conjunto de pruebas. Esto significa que podemos esperar que nuestro modelo sea correcto el 97% de las veces para los nuevos iris. Para nuestra aplicación de botánico aficionado, este alto nivel de precisión significa que nuestro modelo puede ser lo suficientemente fiable como para ser utilizado.

Resumiendo

Para construir nuestro modelo, utilizamos un conjunto de datos de mediciones anotadas por un experto con la especie correcta, lo que hace que esta tarea sea de aprendizaje supervisado. Había tres especies posibles, setosa, versicolor o virginica, lo que convertía la tarea en un problema de clasificación de tres clases. Las especies posibles se denominan clases en el problema de clasificación, y la especie de un iris se llama etiqueta.

El conjunto de datos Iris consta de dos matrices NumPy: una que contiene los datos, que se denomina X en scikit-learn, y otra que contiene los resultados correctos o deseados, que se denomina y. La matriz X es una matriz bidimensional de características, con una fila por punto de datos y una columna por característica. La matriz y es una matriz unidimensional, que aquí contiene una etiqueta de clase, un número entero que va de 0 a 2, para cada una de las muestras.

Dividimos nuestro conjunto de datos en un conjunto de entrenamiento, para construir nuestro modelo, y un conjunto de prueba, para evaluar lo bien que nuestro modelo generalizará a nuevos datos no vistos previamente.

Elegimos el algoritmo de clasificación k-nearest neighbors, que realiza predicciones para un nuevo punto de datos teniendo en cuenta su(s) vecino(s) más cercano(s) en el conjunto de entrenamiento. Esto se implementa en la clase KNeighborsClassifier, que contiene el algoritmo que construye el modelo, así como el algoritmo que hace una predicción utilizando el modelo. Instanciamos la clase y establecemos los parámetros. Después construimos el modelo llamando al método fit, pasando los datos de entrenamiento (X_train) y los resultados del entrenamiento (y_train) como parámetros. Evaluamos el modelo utilizando el método de puntuación, que calcula la precisión del modelo. Aplicamos el método de puntuación a los datos del conjunto de pruebas y a las etiquetas del conjunto de pruebas y comprobamos que nuestro modelo tiene una precisión del 97%, lo que significa que acierta el 97% de las veces en el conjunto de pruebas.

Esto nos dio confianza para aplicar el modelo a nuevos datos (en nuestro ejemplo, nuevas mediciones de flores) y confiar en que el modelo sería correcto aproximadamente el 97% de las veces.

He aquí un resumen del código necesario para todo el procedimiento de entrenamiento y evaluación:

In[32]:

```
X_train, X_test, y_train, y_test = train_test_split(
    iris_dataset['data'], iris_dataset['target'], random_state=0)

knn = KNeighborsClassifier(n_neighbors=1)
knn.fit(X_train, y_train)

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

Out[32]:

```
Test set score: 0.97
```

Este fragmento contiene el código central para aplicar cualquier algoritmo de aprendizaje automático utilizando *scikit-learn*. Los métodos fit, predict y score son la interfaz común a los modelos supervisados en *scikit-learn*, y con los conceptos introducidos en este capítulo, puede aplicar estos modelos a muchas tareas de aprendizaje automático.

Ejercicios

Inicialmente, ejercitaremos los conceptos fundamentales de Python vistos en la conferencia y la actividad no presencial anterior. Para ello debe resolver los problemas siguientes en la plataforma vjudge.net:

1. https://vjudge.net/problem/HackerRank-python-lists
2. https://vjudge.net/problem/HackerRank-python-string-split-and-join

3. https://vjudge.net/problem/HackerRank-python-integers-come-in-all-sizes
4. https://vjudge.net/problem/HackerRank-python-quest-1
5. https://vjudge.net/problem/HackerRank-py-set-discard-remove-pop

Luego para ejercitar las herramientas introducidas en la guía realice los siguientes ejercicios:

1. Usando Numpy cree una matriz de dos dimensiones de 3x3 los sus valores de 1 a 9.
2. Usando Numpy cree una matríz de 5x3 formada unicamente por unos (utilice las funciones de esa biblioteca).
3. Cree un diccionario en Python con varias series de valores, por ejemplo: una llave ciudad que tenga como valor una lista de 5 ciudades, otro campo “población” que tenga una lista con las poblaciones de dichas ciudades, y por último otro campo “país” que tenga una lista de los países a los que pertenece dichas ciudades. Luego convierta ese diccionario a un DataFrame de *pandas*.
   a) Filtre el DataFrame creado para que muestre solamente las ciudades pertenecientes a un país en concreto.
   b) Imprima cuantas ciudades tienen más de 500 000 habitantes.
4. Usando el DataFrame creado anteriormente, cree un gráfico de barra usando matplotlib donde se muestre por cada ciudad cual es su población.
5. En *matplotlib* cree un gráfico que represente la función tangente.
6. Reproduzca el ejemplo presentado para la creación de un modelo para el dataset iris, con el algoritmo KNN.

Elaborado por: M.Sc. Angel Alberto Vazquez Sánchez
Profesor Auxiliar del Departamento de Inteligencia Computacional.