The Iris Dataset in Python - K-Nearest Neighbours

The Iris dataset available under the `sklearn` module in Python is known for its simplicity. It contains the measurements of 150 iris flowers from three different species: *Iris setosa*, *Iris versicolor*, and *Iris virginica*.

The `sklearn` module has a very straightforward set of data on these iris species, consisting of the following:

- Features in the Iris dataset:
    1. sepal length (in cm)
    2. sepal width (in cm)
    3. petal length (in cm)
    4. petal width (in cm)
- Target classes to predict:
    1. *Iris setosa*
    2. *Iris versicolor*
    3. *Iris virginica*

1. Loading the Iris dataset with Scikit-learn

`scikit-learn` embeds a copy of the Iris CSV file along with a helper function to load it into `numpy` arrays:

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

The resulting dataset is a `Bunch` object.

```
type(iris)
```

```
sklearn.utils._bunch.Bunch
def __init__(**kwargs)

/usr/local/lib/python3.10/dist-packages/sklearn/utils/_bunch.py
Container object exposing keys as attributes.

Bunch objects are sometimes used as an output for functions and methods.
They extend dictionaries by enabling values to be accessed by key,
```

The features of each sample flower are stored in the `data` attribute of the dataset. Let us find out the number of samples and the number of features in the dataset, as well as the data for the first sample flower:

```
n_samples, n_features = iris.data.shape
print('Number of samples in the dataset: ', n_samples)
print('Number of features in the dataset: ', n_features)
print('Data for the first sample flower: ', iris.data[0])
```

```
Number of samples in the dataset:  150
Number of features in the dataset:  4
Data for the first sample flower:  [5.1 3.5 1.4 0.2]
```

The information about the class of each sample, i.e., the labels, is stored in the `target` attribute of the dataset:

```
print(iris.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]
```

Using the `bincount()` function of NumPy, we can see that the classes in this dataset are evenly distributed: there are 50 flowers of each species, with:

- Class 0: *Iris setosa*
- Class 1: *Iris versicolor*
- Class 2: *Iris virginica*

```
import numpy as np
np.bincount(iris.target)
```

```
array([50, 50, 50])
```

These class names are stored in the last attribute, i.e., `target_names`:

```
print(iris.target_names)
```

```
['setosa' 'versicolor' 'virginica']
```

2. Preparing the dataset

Let us look at the data for four sample flowers of our choice:

```
data = iris.data
labels = iris.target

for i in [0, 49, 99, 149]:
  print(f"index: {i:3}, features: {data[i]}, label: {labels[i]}")
```

```
index:   0, features: [5.1 3.5 1.4 0.2], label: 0
index:  49, features: [5.  3.3 1.4 0.2], label: 0
index:  99, features: [5.7 2.8 4.1 1.3], label: 1
index: 149, features: [5.9 3.  5.1 1.8], label: 2
```

We now create learning and testing datasets from the given dataset, using `permutation` from `np.random` to split the data randomly.

```
import numpy as np
np.random.seed(42)
ind = np.random.permutation(len(data))

n_training_samples = 12
learn_data = data[ind[:-n_training_samples]]
learn_labels = labels[ind[:-n_training_samples]]
test_data = data[ind[-n_training_samples:]]
test_labels = labels[ind[-n_training_samples:]]

print("The first samples of our learning dataset are as follows:")
print(f"{'index':7s}{'data':20s}{'label':3s}")
for i in range(5):
 print(f"{i:4d}  {learn_data[i]}  {learn_labels[i]:3}")

print("The first samples of our testing dataset are as follows:")
print(f"{'index':7s}{'data':20s}{'label':3s}")
for i in range(5):
 print(f"{i:4d}  {learn_data[i]}  {learn_labels[i]:3}")
```

```
The first samples of our learning dataset are as follows:
index  data                label
   0  [6.1 2.8 4.7 1.2]    1
   1  [5.7 3.8 1.7 0.3]    0
   2  [7.7 2.6 6.9 2.3]    2
   3  [6.  2.9 4.5 1.5]    1
   4  [6.8 2.8 4.8 1.4]    1
The first samples of our testing dataset are as follows:
index  data                label
   0  [6.1 2.8 4.7 1.2]    1
   1  [5.7 3.8 1.7 0.3]    0
   2  [7.7 2.6 6.9 2.3]    2
   3  [6.  2.9 4.5 1.5]    1
   4  [6.8 2.8 4.8 1.4]    1
```

3. Determining the neighbours

We will use the Euclidean distance to determine the similarity between two instances. The Euclidean distance can be calculated with the `norm` function of the `np.linalg` module.

```
def distance(inst1, inst2):
  """This function calculates the Euclidean distance between two instances."""
  return np.linalg.norm(np.subtract(inst1, inst2))

print(distance(learn_data[8], learn_data[36]))
```

```
1.019803902718557
```

Let us now define a function `get_neighbours`, which will return a list with `k` neighbours closest to the instance `test_inst`:

```
def get_neighbours(training_set, labels, test_inst, k, distance):
 """
 This function calculates a list of the k nearest neighbors of an instance 'test_inst'. It returns a list of k 3-tuples, and
 each 3-tuple consists of (index, dist, label), where:
 index: the index from the training_set,
 dist: the distance between test_inst and the instance training_set[index]
 distance: a reference to the function used to calculate the Euclidean distance
 """
 distances = []
 for index in range(len(training_set)):
  dist = distance(test_inst, training_set[index])
  distances.append((training_set[index], dist, labels[index]))
 distances.sort(key=lambda x: x[1])
 neighbours = distances[:k]
 return neighbours
```

We test the function on our Iris samples:

```
for i in range(5):
 neighbours = get_neighbours(learn_data, learn_labels, test_data[i], 3, distance = distance)
 print("Index: ",i,'\n',
       "Testset Data: ",test_data[i],'\n',
       "Testset Label: ",test_labels[i],'\n',
       "Neighbours: ",neighbours,'\n')
```

```
Index:  0
 Testset Data:  [5.7 2.8 4.1 1.3]
 Testset Label:  1
 Neighbours:  [(array([5.7, 2.9, 4.2, 1.3]), 0.14142135623730995, 1), (array([5.6, 2.7, 4.2, 1.3]), 0.17320508075688815, 1), (array

Index:  1
 Testset Data:  [6.5 3.  5.5 1.8]
 Testset Label:  2
 Neighbours:  [(array([6.4, 3.1, 5.5, 1.8]), 0.1414213562373093, 2), (array([6.3, 2.9, 5.6, 1.8]), 0.24494897427831783, 2), (array(

Index:  2
 Testset Data:  [6.3 2.3 4.4 1.3]
 Testset Label:  1
 Neighbours:  [(array([6.2, 2.2, 4.5, 1.5]), 0.26457513110645864, 1), (array([6.3, 2.5, 4.9, 1.5]), 0.574456264653803, 1), (array([6

Index:  3
 Testset Data:  [6.4 2.9 4.3 1.3]
 Testset Label:  1
 Neighbours:  [(array([6.2, 2.9, 4.3, 1.3]), 0.20000000000000018, 1), (array([6.6, 3. , 4.4, 1.4]), 0.2645751311064587, 1), (array(

Index:  4
 Testset Data:  [5.6 2.8 4.9 2. ]
 Testset Label:  2
 Neighbours:  [(array([5.8, 2.7, 5.1, 1.9]), 0.31622776601683755, 2), (array([5.8, 2.7, 5.1, 1.9]), 0.31622776601683755, 2), (array
```

4. Voting to get a single result

We define a `vote` function, which will use the class `Counter` from `collections` to count the number of the classes inside an instance list (here, the neighbours), and return the most common class.

```
from collections import Counter

def vote(neighbours):
 class_counter = Counter()
 for neighbour in neighbours:
  class_counter[neighbour[2]] += 1
 return class_counter.most_common(1)[0][0]
```

We test the function on our training samples:

```
for i in range(n_training_samples):
 neighbours = get_neighbours(learn_data, learn_labels, test_data[i], 3, distance = distance)
 print("index: ", i,
       " result of vote: ", vote(neighbours),
       " label: ", test_labels[i],
       " data: ", test_data[i])
```

```
index:  0  result of vote:  1  label:  1  data:  [5.7 2.8 4.1 1.3]
index:  1  result of vote:  2  label:  2  data:  [6.5 3.  5.5 1.8]
index:  2  result of vote:  1  label:  1  data:  [6.3 2.3 4.4 1.3]
index:  3  result of vote:  1  label:  1  data:  [6.4 2.9 4.3 1.3]
index:  4  result of vote:  2  label:  2  data:  [5.6 2.8 4.9 2. ]
index:  5  result of vote:  2  label:  2  data:  [5.9 3.  5.1 1.8]
index:  6  result of vote:  0  label:  0  data:  [5.4 3.4 1.7 0.2]
index:  7  result of vote:  1  label:  1  data:  [6.1 2.8 4.  1.3]
index:  8  result of vote:  1  label:  2  data:  [4.9 2.5 4.5 1.7]
index:  9  result of vote:  0  label:  0  data:  [5.8 4.  1.2 0.2]
index:  10  result of vote:  1  label:  1  data:  [5.8 2.6 4.  1.2]
index:  11  result of vote:  2  label:  2  data:  [7.1 3.  5.9 2.1]
```

Observation: The predictions correspond to the actual labeled results, except in the case of the item with index 8.

Let us now define a function `vote_with_prob` similar to `vote`, but this one will return the class name and probability for this class.

```
def vote_prob(neighbours):
 class_counter = Counter()
 for neighbour in neighbours:
  class_counter[neighbour[2]] += 1
 labels, votes = zip(*class_counter.most_common())
 winner = class_counter.most_common(1)[0][0]
 votes_for_winner = class_counter.most_common(1)[0][1]
 return winner, votes_for_winner/sum(votes)

for i in range(n_training_samples):
 neighbours = get_neighbours(learn_data, learn_labels, test_data[i], 5, distance = distance)
 print("index: ", i,
       " vote_prob: ", vote_prob(neighbours),
       " label: ", test_labels[i],
       " data: ", test_data[i])
```

```
index:  0  vote_prob:  (1, 1.0)  label:  1  data:  [5.7 2.8 4.1 1.3]
index:  1  vote_prob:  (2, 1.0)  label:  2  data:  [6.5 3.  5.5 1.8]
index:  2  vote_prob:  (1, 1.0)  label:  1  data:  [6.3 2.3 4.4 1.3]
index:  3  vote_prob:  (1, 1.0)  label:  1  data:  [6.4 2.9 4.3 1.3]
index:  4  vote_prob:  (2, 1.0)  label:  2  data:  [5.6 2.8 4.9 2. ]
index:  5  vote_prob:  (2, 0.8)  label:  2  data:  [5.9 3.  5.1 1.8]
index:  6  vote_prob:  (0, 1.0)  label:  0  data:  [5.4 3.4 1.7 0.2]
index:  7  vote_prob:  (1, 1.0)  label:  1  data:  [6.1 2.8 4.  1.3]
index:  8  vote_prob:  (1, 1.0)  label:  2  data:  [4.9 2.5 4.5 1.7]
index:  9  vote_prob:  (0, 1.0)  label:  0  data:  [5.8 4.  1.2 0.2]
index:  10  vote_prob:  (1, 1.0)  label:  1  data:  [5.8 2.6 4.  1.2]
index:  11  vote_prob:  (2, 1.0)  label:  2  data:  [7.1 3.  5.9 2.1]
```

Start coding or generate with AI.