knn

August 14, 2020

1 *k*-Nearest Neighbours (*k*-NN)

Welcome! In this Jupyter notebook, you will use scikit-learn to build and predict using the *k*-Nearest Neighbour (or *k*-NN) algorithm. We will also talk distance measures, a bit. But that's enough, let's jump into it.

1.1 Before you start

- In order for the notebooks to function as intended, modify only between lines marked "### begin your code here (__ lines)." and "### end your code here.".
- The line count is a suggestion of how many lines of code you need to accomplish what is asked.
- You should execute the cells (the boxes that a notebook is composed of) in order.
- You can execute a cell by pressing Shift and Enter (or Return) simultaneously.
- You should have completed the *Decision Trees* Jupyter notebook before attempting this one as the concepts covered there are not repeated, for the sake of brevity.

1.2 Loading the appropriate packages

Again, we are loading required packages. From scikit-learn, we will load the packages appropriate for k-NNs this time.

[1]:
```
import numpy as np
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import LabelEncoder
import pandas as pd
import plotly.express as px
import plotly.graph_objects as go
```

Let's turn off the scientific notation for floating point numbers.

[2]:
```
np.set_printoptions(suppress=True)
```

1.3 Loading and examining the data

We will load our data from a CSV file and put it in a pandas an object of the `DataFrame` class.

This is the Iris flower dataset, a very famous dataset which was published in 1936 by studying three different species of the flower Iris: *Iris setosa*, *Iris versicolor* and *Iris virginica*. Originally, the dataset has 150 examples which corresponds to 150 different Iris flowers measured (50 flowers from each species). The dataset has 4 features, the sepal length, sepal width, petal length and petal width of each flower in centimeters. However, for the purpose of ease of illustration, we have chosen only two of the features: the sepal length and sepal width.

- This was taken and modified from the Machine Learning dataset repository of School of Information and Computer Science of University of California Irvine (UCI):

  Dua, D. and Graff, C. (2019). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.

[3]:
```
df = pd.read_csv('data_knn.csv')
```

Let's take a look at our data as a table:

[4]:
```
df
```

[4]:
```
    Sepal Length  Sepal Width      Species
0            5.1          3.5  Iris setosa
1            4.9          3.0  Iris setosa
2            4.7          3.2  Iris setosa
3            4.6          3.1  Iris setosa
4            5.0          3.6  Iris setosa
5            5.4          3.9  Iris setosa
6            4.6          3.4  Iris setosa
7            5.0          3.4  Iris setosa
8            4.4          2.9  Iris setosa
9            4.9          3.1  Iris setosa
10           5.4          3.7  Iris setosa
11           4.8          3.4  Iris setosa
12           4.8          3.0  Iris setosa
13           4.3          3.0  Iris setosa
14           5.8          4.0  Iris setosa
15           5.7          4.4  Iris setosa
16           5.4          3.9  Iris setosa
17           5.1          3.5  Iris setosa
18           5.7          3.8  Iris setosa
19           5.1          3.8  Iris setosa
20           5.4          3.4  Iris setosa
21           5.1          3.7  Iris setosa
22           4.6          3.6  Iris setosa
23           5.1          3.3  Iris setosa
24           4.8          3.4  Iris setosa
25           5.0          3.0  Iris setosa
26           5.0          3.4  Iris setosa
```

```
27            5.2          3.5     Iris setosa
28            5.2          3.4     Iris setosa
29            4.7          3.2     Iris setosa
..            ...          ...             ...
120           6.9          3.2  Iris virginica
121           5.6          2.8  Iris virginica
122           7.7          2.8  Iris virginica
123           6.3          2.7  Iris virginica
124           6.7          3.3  Iris virginica
125           7.2          3.2  Iris virginica
126           6.2          2.8  Iris virginica
127           6.1          3.0  Iris virginica
128           6.4          2.8  Iris virginica
129           7.2          3.0  Iris virginica
130           7.4          2.8  Iris virginica
131           7.9          3.8  Iris virginica
132           6.4          2.8  Iris virginica
133           6.3          2.8  Iris virginica
134           6.1          2.6  Iris virginica
135           7.7          3.0  Iris virginica
136           6.3          3.4  Iris virginica
137           6.4          3.1  Iris virginica
138           6.0          3.0  Iris virginica
139           6.9          3.1  Iris virginica
140           6.7          3.1  Iris virginica
141           6.9          3.1  Iris virginica
142           5.8          2.7  Iris virginica
143           6.8          3.2  Iris virginica
144           6.7          3.3  Iris virginica
145           6.7          3.0  Iris virginica
146           6.3          2.5  Iris virginica
147           6.5          3.0  Iris virginica
148           6.2          3.4  Iris virginica
149           5.9          3.0  Iris virginica

[150 rows x 3 columns]
```

Because now our data has 2 features, we can use a single scatter plot to take a look at our data:

[5]:
```
fig = px.scatter(df, x="Sepal Length", y="Sepal Width", color="Species")
fig.show()
```

Let's extract our data and targets as NumPy arrays `X` and `y`, from pandas data frame `df`. This time, for visualization purposes, we need our targets `y` to be integers instead of text, so we have to extract text labels as `y_text` and then transform them into integer labels `y`. Fortunately, we can do that with the `LabelEncoder` class that comes in scikit-learn:

[6]:
```
X = df.drop('Species', axis=1).to_numpy()
y_text = df['Species'].to_numpy()
y = LabelEncoder().fit_transform(y_text)
```

Let’s see our data `X`:

[7]:
```
X
```

[7]:
```
array([[5.1, 3.5],
       [4.9, 3. ],
       [4.7, 3.2],
       [4.6, 3.1],
       [5. , 3.6],
       [5.4, 3.9],
       [4.6, 3.4],
       [5. , 3.4],
       [4.4, 2.9],
       [4.9, 3.1],
       [5.4, 3.7],
       [4.8, 3.4],
       [4.8, 3. ],
       [4.3, 3. ],
       [5.8, 4. ],
       [5.7, 4.4],
       [5.4, 3.9],
       [5.1, 3.5],
       [5.7, 3.8],
       [5.1, 3.8],
       [5.4, 3.4],
       [5.1, 3.7],
       [4.6, 3.6],
       [5.1, 3.3],
       [4.8, 3.4],
       [5. , 3. ],
       [5. , 3.4],
       [5.2, 3.5],
       [5.2, 3.4],
       [4.7, 3.2],
       [4.8, 3.1],
       [5.4, 3.4],
       [5.2, 4.1],
       [5.5, 4.2],
       [4.9, 3.1],
       [5. , 3.2],
       [5.5, 3.5],
       [4.9, 3.6],
       [4.4, 3. ],
       [5.1, 3.4],
       [5. , 3.5],
       [4.5, 2.3],
       [4.4, 3.2],
       [5. , 3.5],
       [5.1, 3.8],
       [4.8, 3. ],
       [5.1, 3.8],
       [4.6, 3.2],
       [5.3, 3.7],
       [5. , 3.3],
       [7. , 3.2],
       [6.4, 3.2],
       [6.9, 3.1],
       [5.5, 2.3],
       [6.5, 2.8],
       [5.7, 2.8],
       [6.3, 3.3],
       [4.9, 2.4],
       [6.6, 2.9],
       [5.2, 2.7],
       [5. , 2. ],
       [5.9, 3. ],
       [6. , 2.2],
       [6.1, 2.9],
       [5.6, 2.9],
       [6.7, 3.1],
       [5.6, 3. ],
       [5.8, 2.7],
       [6.2, 2.2],
       [5.6, 2.5],
       [5.9, 3.2],
       [6.1, 2.8],
       [6.3, 2.5],
       [6.1, 2.8],
       [6.4, 2.9],
       [6.6, 3. ],
       [6.8, 2.8],
       [6.7, 3. ],
       [6. , 2.9],
       [5.7, 2.6],
       [5.5, 2.4],
       [5.5, 2.4],
       [5.8, 2.7],
       [6. , 2.7],
       [5.4, 3. ],
       [6. , 3.4],
       [6.7, 3.1],
       [6.3, 2.3],
       [5.6, 3. ],
       [5.5, 2.5],
       [5.5, 2.6],
       [6.1, 3. ],
[5.8, 2.6],
[5. , 2.3],
[5.6, 2.7],
[5.7, 3. ],
[5.7, 2.9],
[6.2, 2.9],
[5.1, 2.5],
[5.7, 2.8],
[6.3, 3.3],
[5.8, 2.7],
[7.1, 3. ],
[6.3, 2.9],
[6.5, 3. ],
[7.6, 3. ],
[4.9, 2.5],
[7.3, 2.9],
[6.7, 2.5],
[7.2, 3.6],
[6.5, 3.2],
[6.4, 2.7],
[6.8, 3. ],
[5.7, 2.5],
[5.8, 2.8],
[6.4, 3.2],
[6.5, 3. ],
[7.7, 3.8],
[7.7, 2.6],
[6. , 2.2],
[6.9, 3.2],
[5.6, 2.8],
[7.7, 2.8],
[6.3, 2.7],
[6.7, 3.3],
[7.2, 3.2],
[6.2, 2.8],
[6.1, 3. ],
[6.4, 2.8],
[7.2, 3. ],
[7.4, 2.8],
[7.9, 3.8],
[6.4, 2.8],
[6.3, 2.8],
[6.1, 2.6],
[7.7, 3. ],
[6.3, 3.4],
[6.4, 3.1],
[6. , 3. ],
[6.9, 3.1],
[6.7, 3.1],
[6.9, 3.1],
[5.8, 2.7],
[6.8, 3.2],
[6.7, 3.3],
[6.7, 3. ],
[6.3, 2.5],
[6.5, 3. ],
[6.2, 3.4],
[5.9, 3. ]])
```

Let's check the shape of `X`as well:

[8]:
```
X.shape
```

[8]:
```
(150, 2)
```

Let's do the same checks with the targets vector `y`:

[9]:
```
y
```

[9]:
```
array([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])
```

…and the shape of `y`:

[10]:
```
y.shape
```

[10]:
```
(150,)
```

Everything looks good.

1.4 Splitting data

Again, let's split our data into training, validation and test sets. Let's use 60% (90 examples) for training, 20% for validation (30 examples) and the remaining 20% (30 examples) as test data.

[11]:
```
(X_train, X_vt, y_train, y_vt) = train_test_split(X, y, test_size=0.4,␣
 ↪random_state=0)
(X_validation, X_test, y_validation, y_test) = train_test_split(X_vt, y_vt,␣
 ↪test_size=0.5, random_state=0)
```

1.5 Building and visualizing a *k*-NN model

Now, let's build our *k*-NN model! We will create an object of the class `KNeighborsClassifier` and assign the name `knn` for the resulting object. Remember that we have to specify a number for *k* which is called `n_neighbors` in scikit-learn implementation.

However, before doing that let's have a quick discussion about distance measures which greatly affect how our k-NN classifier performs: Remember that you have to use your knowledge of the data to come up with a distance measure that makes sense for the data you have. You also may have to scale different features by different weights to get a good diatnce measurement out of their combination, before sttempting to train and use a model. For example, if we have integer data, it may make sense to use a Manhattan distance measure. Or use Hamming distance for bit data.

In our data here, all our measure ments are of length and are in centimeters, so no adjustment is needed and a Euclidian distance measure actually makes a lot of sense, since it is measuring some kind of length and the unweighted combination is also sound. For `KNeighborsClassifier`, the default distance measure or `metric` (as the argument is called in scikit-learn) is the L_p measure or the Minkowski distance (`metric=minkowski` in scikit-learn call) with $p = 2$ (p=2 in `KNeighborsClassifier` arguments), which is nothing other than the L_2 distance measure or the Eucliadian distance.

So, other than setting `n_neighbors` to a suitable value, we don't have to specify anything else. Let's start with a value of 1 for `n_neighbors`.

You can see the documentation for `KNeighborsClassifier` here:

https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html

Go ahead and implement that now:

[12]:
```
### begin your code here (1 line).
knn = KNeighborsClassifier(n_neighbors=1)
### end your code here.
```

Next, let's fit our `knn` to our `X_train` and `y_train`. This does nothing but store the training example as k-NN is "lazy": It does all calculations as prediction time and by measuring the distance from the operational datapoints provided (whose labels have to be predicted) to each of the training datapoints, finding the closest training datapoints to the operational points, looking at the labels for those closest training datapoints, and finding the majority class among them.

[13]:
```
### begin your code here (1 line).
knn.fit(X_train, y_train)
### end your code here.
```

[13]:
```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=None, n_neighbors=1, p=2,
                     weights='uniform')
```

Again, you will get a summary for the model:

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

Now, let's visualize our k-NN model. We are using plotly heatmaps (which are a bit more tedious to produce at this moment in time as they have not been ported to plotly express yet) to show regions where points will be predicted in different classes (this will take some time) and we will overlay a scatter plot of training points on top of that:

[14]:
```
detail_steps = 500

(x_vis_0_min, x_vis_0_max) = (X[:, 0].min() - 0.5, X[:, 0].max() + 0.5)
(x_vis_1_min, x_vis_1_max) = (X[:, 1].min() - 0.5, X[:, 1].max() + 0.5)
```

```
x_vis_0_range = np.linspace(x_vis_0_min, x_vis_0_max, detail_steps)
x_vis_1_range = np.linspace(x_vis_1_min, x_vis_1_max, detail_steps)

(XX_vis_0, XX_vis_1) = np.meshgrid(x_vis_0_range, x_vis_1_range)
X_vis = np.c_[XX_vis_0.reshape(-1), XX_vis_1.reshape(-1)]

yhat_vis = knn.predict(X_vis)
YYhat_vis = yhat_vis.reshape(XX_vis_0.shape)

region_colorscale = [
                     [0.0, 'rgb(199, 204, 249)'],
                     [0.5, 'rgb(235, 185, 177)'],
                     [1.0, 'rgb(159, 204, 186)']
                    ]
points_colorscale = [
                     [0.0, 'rgb(99, 110, 250)'],
                     [0.5, 'rgb(239, 85, 59)'],
                     [1.0, 'rgb(66, 204, 150)']
                    ]
fig2 = go.Figure(
               data=[
                     go.Heatmap(x=x_vis_0_range,
                                y=x_vis_1_range,
                                z=YYhat_vis,
                                colorscale=region_colorscale,
                                showscale=False),
                     go.Scatter(x=df['Sepal Length'],
                                y=df['Sepal Width'],
                                mode='markers',
                                text=df['Species'],
                                name='',
#                                showscale=False,
                                marker=dict(
                                            color=y,
                                            colorscale=points_colorscale
                                           )
                               )
                    ],
                    layout=go.Layout(
                                     xaxis=dict(title='Sepal Length'),
                                     yaxis=dict(title='Sepal Width')
                                    )
              )
fig2.show()
```

1.6 Model selection and asessment

Evaluation time! Let's first see how does or k-NN does on training data. We expect to get an accuracy result of near 100% or 1.0. Not exactly 100% because the algorihthm in scikit-learn does not count the same point as a neighbour. Otherwise, we would have had a perfect 100% since the closest single point to each training point in the training points is that point itself and the classes predicted will match that of the training targets. So go ahead and calculate `yhat_train` using `knn` and `X_train`:

[15]:
```
### begin your code here (1 line).
yhat_train = knn.predict(X_train)
### end your code here.
```

Now, to measure the accuracy from `yhat_train` and `y_train`:

[16]:
```
accuracy_score(yhat_train, y_train)
```

[16]:
```
0.9444444444444444
```

It is ~95%. It is as we expected. And we also expect the accuracy on validation points not be that high, since we are probably overfitting by using $k = 1$. Put in the line that generates `yhat_validation`, so we can measure the accuracy:

[17]:
```
### begin your code here (1 line).
yhat_validation = knn.predict(X_validation)
### end your code here.
accuracy_score(yhat_validation, y_validation)
```

[17]:
```
0.7
```

Okay, we got ~70% which has a big gap with the accuracy on training data.

Let's use a higher value of k or `n_neighbors`. Let's use 3 and see what happens. Go ahead a make a new model `knn3` and provide `X_train` and `y_train` to it afterwards using the methd `fit`:

[18]:
```
### begin your code here (2 lines).
knn3 = KNeighborsClassifier(n_neighbors=3)
knn3.fit(X_train, y_train)
### end your code here.
```

[18]:
```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=None, n_neighbors=3, p=2,
                     weights='uniform')
```

Our model summary should indicate `n_neighbors=3` now:

> KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski', metric_params=None, n_jobs=None, **n_neighbors=3**, p=2, weights='uniform')

We can visualize the model with $k = 3$:

[19]:
```
yhat3_vis = knn3.predict(X_vis)
YYhat3_vis = yhat3_vis.reshape(XX_vis_0.shape)

fig3 = fig2
fig3['data'][0]['z'] = YYhat3_vis
```

```
fig3.show()
```

You can see the regions are smoother and less "patchy".

Now, predict `yhat_train3` with this new `knn3`, so we can the accuracy on training data:

[20]:
```
### begin your code here (1 line).
yhat_train3 = knn3.predict(X_train)
### end your code here.
accuracy_score(yhat_train3, y_train)
```

[20]:
```
0.8888888888888888
```

We got ~89% This time we did not predict to near 100% accuracy since the training points surrounding a training point may have different labels. On to validation accuracy then. Let's predict `yhat_validation3` and we can the accuracy on validation data:

[21]:
```
### begin your code here (1 line).
yhat_validation3 = knn3.predict(X_validation)
### end your code here.
accuracy_score(yhat_validation3, y_validation)
```

[21]:
```
0.6333333333333333
```

We get a better accuracy on validation points (~64%), again as expected, since we are probably overfitting less. Let's try a $k = 5$ as well. Make a new model `knn5` with `n_neighbors=5` and provide the data to it via `fit`:

[22]:
```
### begin your code here (2 lines).
knn5 = KNeighborsClassifier(n_neighbors=5)
knn5.fit(X_train, y_train)
### end your code here.
```

[22]:
```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
           metric_params=None, n_jobs=None, n_neighbors=5, p=2,
           weights='uniform')
```

You should get:

KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski', metric_params=None, n_jobs=None, **n_neighbors=5**, p=2, weights='uniform')

We can visualize the model with $k = 5$ as well:

[23]:
```
yhat5_vis = knn5.predict(X_vis)
YYhat5_vis = yhat3_vis.reshape(XX_vis_0.shape)

fig4 = fig2
fig4['data'][0]['z'] = YYhat5_vis

fig4.show()
```

You can see the regions are even more smooth.

Next, let's evaluate our model. First on training data. fill in the part that calculates `yhat_train5` and we can measure the accuracy on training data:

[24]:
```
### begin your code here (1 line).
yhat_train5 = knn5.predict(X_train)
### end your code here.
accuracy_score(yhat_train5, y_train)
```

[24]:
```
0.8666666666666667
```

86.7% Do the same on validation data by predicting `yhat_validation5`:

[25]:
```
### begin your code here (1 line).
yhat_validation5 = knn5.predict(X_validation)
### end your code here.
accuracy_score(yhat_validation5, y_validation)
```

[25]:
```
0.6333333333333333
```

So, we get 63.33% on validation set.

We got our best result with $k = 3$, so let's get a final evaluation on our held-out test set. Predict `yhat_test3`:

[26]:
```
### begin your code here (1 line).
yhat_test3 = knn3.predict(X_test)
### end your code here.
accuracy_score(yhat_test3, y_test)
```

[26]:
```
0.7
```

The accuracy on test data (70%) is both close to performance on validation and training data and is high.

We have a good k-NN model now.