Week 7, K-Nearest Neighbors

Author: Jessica Cervi

Expected time = 2 hours

Assignment Overview

In this assignment you will work with k-nearest neighbors to classify wine quality using Pandas, NumPy, and SkLearn python packages. You will use Pandas to import the dataset and convert the data into NumPy arrays to prepare it for classification using sklearn. You will then split the dataset into a training dataset and a test dataset. From there, you will use sklearn to normalize the datasets, train the classifier, and use cross-validation to select the best k. Finally, you will evaluate the selected classifier using the test dataset.

This assignment is designed to help you apply the machine learning algorithms you have learned using packages in Python. Python concepts, instruction, and starter code are embedded within this Jupyter Notebook to help guide you as you progress through the assignment. Remember to run the code of each code cell prior to submitting the assignment. Upon completing the assignment, we encourage you to compare your work against the solution file to perform a self-assessment.

Learning Objectives

- Outline k-nearest neighbours for classification
- Define the concept of proximity for k-nearest neighbours methods
- Convert binary and categorical predictors into numbers
- Explain the relationship between selecting k and the bias-variance trade-off
- Outline k-nearest neighbours for regression
- Discuss real-life applications of k-nearest neighbours

Index:

Week 7: K-Nearest Neighbors

In Week 7, you learned about **k-nearest neighbour** (KNN) for classification.

The KNN algorithm is one of the simplest classification algorithms. KNN is used to predict the classification of a new sample point, based on datasets that are made up of data which are separated into several classes or categories.

The pseudo-algorithm for KNN can be summarized as follows:

1. Load the training and test data
2. Choose the value of K
3. For each point in test data:
    - find the Euclidean distance to all training data points
    - store the Euclidean distances in a list and sort it
    - choose the first k points
    - assign a class to the test point based on the majority of classes present in the chosen points
4. Review Ouput

Predicting Wine Quality with k-Nearest Neighbours

For this exercise we will use the dataset “sparklingwine.csv” to predict wine quality and we will build a KNN classifier in Python for the dataset going through the following steps:

1. Load the data file;
2. Construct a new binary column “good wine” that indicates whether the wine is good (which we define as having a quality of 6 or higher) or not;
3. Move the data to NumPy arrays and split the data set into a training data set (first 400 samples), a validation data set (next 200 samples) and a test data set (last 200 samples)
4. Normalise the data according to the Z-score transform;
5. Load and train the k-Nearest Neighbours classifiers for k = 1,2, ...,100;
6. Evaluate each classifier using the validation data set and select the best classifier;
7. Predict the generalisation error using the test data set.

Back to top

Part 1 - Importing the Dataset and Exploratory Data Analysis (EDA)

We begin by using `pandas` to import the dataset. To do so, we import `pandas` first and we read the file using the `.read_csv()` function by passing the name of the dataset we want to read as a string.

Notice that, because the columns in the dataset are separated using a `;` , we specified the type of delimiter in the `.read_csv()` function (the default value is `,`).

Complete the code cell below adding the name of the dataset inside `.read_csv()` .

In [1]:
```
import pandas as pd

df = pd.read_csv("sparklingwine.csv", sep = ";")
```

Before performing any algorithm on the dataframe, it is always good practice to perform exploratory data analysis.

We begin by visualizing the first ten rows of the DataFrame `df` using the function `.head()` . By default, `.head()` displays the first five rows of a DataFrame.

Complete the code cell below by passing the desired number of rows to the function `.head()` as an integer.

In [2]:
```
df.head(10)
```

Out[2]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcoho
0	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4
1	7.8	0.88	0.00	2.6	0.098	25.0	67.0	0.9968	3.20	0.68	9.8
2	7.8	0.76	0.04	2.3	0.092	15.0	54.0	0.9970	3.26	0.65	9.8
3	11.2	0.28	0.56	1.9	0.075	17.0	60.0	0.9980	3.16	0.58	9.8
4	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4
5	7.4	0.66	0.00	1.8	0.075	13.0	40.0	0.9978	3.51	0.56	9.4
6	7.9	0.60	0.06	1.6	0.069	15.0	59.0	0.9964	3.30	0.46	9.4
7	7.3	0.65	0.00	1.2	0.065	15.0	21.0	0.9946	3.39	0.47	10.0
8	7.8	0.58	0.02	2.0	0.073	9.0	18.0	0.9968	3.36	0.57	9.5
9	7.5	0.50	0.36	6.1	0.071	17.0	102.0	0.9978	3.35	0.80	10.5

Next, we retrieve some more information about our DataFrame by using the properties `.shape` and `columns` and the function `.describe()` .

Here's a brief description of what each of the above functions does:

- `.shape` : Returns a tuple representing the dimensionality of the DataFrame.
- `.columns` : Returns the column labels of the DataFrame.
- `.describe()` : Returns summary statistics of the columns in the Dataframe provided such as mean, count, standad deviation and so on.

Run the cells below to get information aboout the DataFrame.

In [3]:
```
df.shape
```

Out[3]:
```
(800, 12)
```

In [4]:
```
df.columns
```

Out[4]:
```
Index(['fixed acidity', 'volatile acidity', 'citric acid', 'residual sugar',
       'chlorides', 'free sulfur dioxide', 'total sulfur dioxide', 'density',
       'pH', 'sulphates', 'alcohol', 'quality'],
      dtype='object')
```

In [5]:
```
df.describe()
```

Out[5]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	
count	800.000000	800.000000	800.000000	800.000000	800.000000	800.000000	800.000000	800.
mean	8.860750	0.539938	0.303875	2.617938	0.093080	14.946250	50.710000	0.
std	1.890583	0.180819	0.206469	1.262319	0.054436	9.725725	34.295205	0.
min	4.600000	0.180000	0.000000	1.200000	0.034000	1.000000	8.000000	0.
25%	7.500000	0.407500	0.120000	2.000000	0.073000	7.000000	24.000000	0.
50%	8.400000	0.530000	0.280000	2.300000	0.082000	12.000000	41.500000	0.
75%	10.000000	0.645000	0.490000	2.800000	0.094000	20.000000	66.000000	0.
max	15.900000	1.330000	1.000000	15.500000	0.611000	68.000000	165.000000	1.

Back to top

Part 2 - Creating a New Binary Column for Good Wines

For the second step of this exercise, we will construct a new binary column `good_wine` that indicates whether a wine is good or not.

Because the values in the new column `good_wine` will be based on the values in the column `quality`, we will use the function `.apply()` which allows us to pass a user-defined function and apply it to every single value of the Pandas series (`quality` in this case).

`.apply()` takes at least the Python function or the `NumPy` function to apply. Additional arguments can be passed as well, a detailed description can be found here (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.apply.html).

In the code cell below, complete the definition of the function `goodwine`. Your function should take, as input, the column `quality`. Based on whether the values in `quality` are greater than or equal to six your function should return 1 or 0 otherwise. Remember, your function should look something like the below and you should fill in the function-name, function-input, and integer values.

```
def function-name (function-input):
    if function-input >= integer :
        return integer
```

In [6]:
```
def goodwine(quality):
    if quality >= 6:
        return 1
    return 0
```

Next, we use the `.apply()` function to create the new column.

Complete the code in the cell below by creating a new column in the DataFrame `df` called `good_wine` and by passing the function `goodwine` as an argument to the function `.apply()` .

Hint: New columns need to be passed to the DataFrame as strings. As you may remember, a string is enclosed in quotation marks like "string."

In [7]:
```
df["good_wine"] = df.quality.apply(goodwine)
```

Run the code cell below to visualize the DataFrame again. Observe that now the DataFrame has a new column `good_wine` .

In [8]:
```
df
```

Out[8]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alco
0	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.99780	3.51	0.56	
1	7.8	0.88	0.00	2.6	0.098	25.0	67.0	0.99680	3.20	0.68	
2	7.8	0.76	0.04	2.3	0.092	15.0	54.0	0.99700	3.26	0.65	
3	11.2	0.28	0.56	1.9	0.075	17.0	60.0	0.99800	3.16	0.58	
4	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.99780	3.51	0.56	
...	...	...	...	...	...	...	...	...	...	...	
795	10.8	0.89	0.30	2.6	0.132	7.0	60.0	0.99786	2.99	1.18	1
796	8.7	0.46	0.31	2.5	0.126	24.0	64.0	0.99746	3.10	0.74	
797	9.3	0.37	0.44	1.6	0.038	21.0	42.0	0.99526	3.24	0.81	1
798	9.4	0.50	0.34	3.6	0.082	5.0	14.0	0.99870	3.29	0.52	1
799	9.4	0.50	0.34	3.6	0.082	5.0	14.0	0.99870	3.29	0.52	1

800 rows × 13 columns

Alternatively, we could have created the new column "good wine" by passing a lambda function to `.apply()`

In [9]:
```
df["good wine"] = df.quality.apply (lambda x : 1 if x>=6 else 0)
```

Back to top

Part 3 - Moving the Data to NumPy Arrays and Splitting the Dataset

The KNN algorithm is implemented in the SciKit-Learn package which takes, as inputs, `NumPy` arrays.

Now, we want to predict wine quality to complete column 13: `good_wine` . To do this, we will use all features (or columns in our dataframe) except `quality` (column 12) and `good_wine` (column 13). Hence, we will take the first 11 columns in our DataFrame as features to predict the classification labels in the 13th column, `good_wine` .

- To begin, we import the `Numpy` package.
- We define `X` and `y` as `Numpy` arrays. So we will start with `X = np.array` for example.
- `X` should be a two-dimensional `Numpy` array of predictors that contains the values of the first eleven columns in `df` .
- `y` should be a one-dimensional `NumPy` array with the response variable: entries of the column `good_wine` .

```
In [10]: import numpy as np
         X = np.array (df[df.columns[:11]])
         y = np.array (df.good_wine)
```

Again, a sanity check, it is good practice to check the dimensions of `X` and `y` .

Run the code cells below to obtain the shapes of `X` and `y` .

```
In [11]: X.shape
Out[11]: (800, 11)
```

```
In [12]: y.shape
Out[12]: (800,)
```

Next, according to the instructions, we split `X` into a training data set (first 400 samples), a validation data set (next 200 samples) and a test data set (last 200 samples). Run the code cell below.

```
In [13]: X_train_unproc = X[:400]
         X_val_unproc = X[400:600]
         X_test_unproc = X[600:]
```

Complete the code cell below to split `y` in the sets `y_train` , `y_set` , and `y_test` in a simiar way as we split `X` .

In [14]:
```
y_train = y[:400]
y_val=y[400:600]
y_test = y[600:]
```

Back to top

Part 4 - Data Normalisation

In this part, we are going to load and train the k-Nearest Neighbours classifiers for k = 1,2, ...,100.

To do so, we are going to normalise the data as the Nearest Neighbours classifier is sensitive to scaling. Standardization of a dataset is a common requirement for many machine learning estimators: the main idea is to normalise (mean = 0 and standard deviation = 1) your features `X` before applying machine learning techniques.

Although here the training and the test sets are available in advance, we are going to take a general approach: we normalise according to the training data and apply the same transformation to the test set in a consistent manner.

In the code cell below, import `.StandardScaler()` from the library `sklearn.preprocessing`. Next use the `.StandardScaler()` function `.fit()` to compute the mean and standard deviation to be used for later scaling on the unprocessed training set `X_train_unproc`. Assign this to the variable `scaler`.

In [15]:
```
#Importing the scaler from scikit-learn
from sklearn.preprocessing import StandardScaler


#Compute the mean and the std on the training set
scaler = StandardScaler().fit(X_train_unproc)
```

Next, we perform center and scale the Data by using the `.transform()` function on each unproccesed feature set.

Run the cell below to obain the normalised `X` sets.

In [16]:
```
X_train = scaler.transform (X_train_unproc)
X_val = scaler.transform (X_val_unproc)
X_test = scaler.transform (X_test_unproc)
```

Back to top

Part 5- Loading and Training the Classifier

To load and train the classifier, we first need to import the Nearest Neighbours package. Complete the cell below by importing `KNeighborsClassifier` from the `sklearn.neighbors` package.

Next use `KNeighborsClassifier` to initialize the classifier `clf`, setting k = 3.

In [17]:
```
from sklearn.neighbors import KNeighborsClassifier

clf = KNeighborsClassifier(3)
```

Run the code cell below to train the classifier `clf` using `X_train` an `y_train` .

In [18]:
```
#train using the training sets
clf.fit (X_train , y_train)
```

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

For our initial choice of k = 3, we evaluate the performance of the classifier on the training data, and we now estimate the performance on new data using the validation set, also known as the test set.

This can be achieved by using the `SciKit-learn` function `.score()`

Compute the score on both the training and validation `X` and `y` sets.

In [19]:
```
#score on the training set
clf.score(X_train, y_train)

#score on the validation set
clf.score(X_val, y_val)
```

Out[19]:
```
0.85
```

Back to top

Part 6 - Evaluate and Select the Best Classifier

To evaluate and select the best classifier, we are going to use cross-validation to choose the value of `k` that has the most promising performance on future data.

We do so by evaluating each classifier and computing the scores using the training and the validation sets.

Define the classifier range `ks` as a sequence of integers from 1 to 100 with step 1, and define two empty lists `inSampleScores` and `valScores` .

In [20]:
```
#defining the range for classifiers
ks=range (1 ,100 ,1)

inSampleScores =[]
valScores =[]
```

Run the cell below to evaluate each classifier and compute the scores.

In [21]:
```
for k in ks:
    clf = KNeighborsClassifier(k).fit(X_train,y_train)
    inSampleScores.append(clf.score(X_train, y_train))
    valScores.append(clf.score(X_val,y_val))
```

Finally, we select the best classifier by plotting the scores for the training (in) and the validation (out) sets.

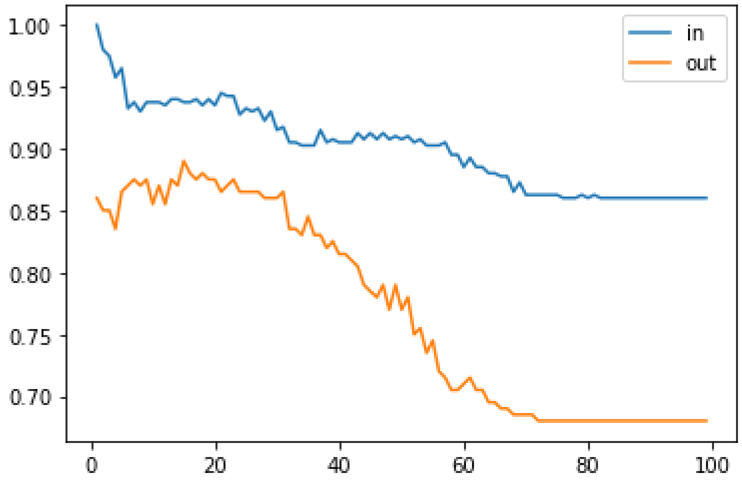
To do so, import `plt` from the `matplotlib.pyplot` library.

In [22]:
```
import matplotlib.pyplot as plt
#This command is necessary to display plots in Jupyter Notebooks
%matplotlib inline
```

Run the cell below to visualize a plot with the scores for the training and validation sets.

In [23]:
```
#Plot
p1 = plt.plot(ks, inSampleScores)
p2 = plt.plot(ks , valScores )
plt.legend(['in', 'out'], loc = 'upper right')
```

Out[23]:
```
<matplotlib.legend.Legend at 0x11e645b90>
```



In analyzing this plot, we see that for small values of k the Nearest Neighbours classifier suffers from overfitting, whereas for large values of k, the classifer suffers from underfitting. Our plot indicates that a k around 15 is a reasonable choice. We will use this value when evaluating the selected classifier on the test set.

Part 7 - Predict the Generalisation Error Using the Test Data Set.

In the last part of this exercise, we will evaluate the classifier with k = 15 on the test set.

Run the cell below to compute the classifier for k=15 and to compute the predicted value of `y` .

In [24]:
```
clf = KNeighborsClassifier(15).fit(X_train , y_train)

y_test_pred = clf.predict(X_test)
```

Finally, in the cell below compute the new score obtained by uning `X_test` and `y_test` . Assign the result to the variable `score_test` .

In [25]:
```
score_test = clf.score(X_test, y_test)
```