Machine Learning Assignment 1 2021

February 14, 2021

1 Importing the Dataset

If you are using Jupyter Notebook, you have to navigate to the folder in which you have stored the data first. There are some 'magic' commands for the Jupyter environment that are not Python com- mands. Search online for 'jupyter magic commands' to learn more about them! We will use Pandas to import the data set. By manually exploring the data file first, we can see that columns are separated by a semicolon.

[1]:
```
import pandas as pd

#load the data file as follows (use only the first 800 instances as instructed)
df = pd.read_csv("winequality-white.csv", delimiter=";", nrows=800)
```

This is a clean data set without any missing values etc. It is always recommended to explore the data set. For example, you may do:

[2]:
```
#df.head()
```

[3]:
```
#df.columns
```

[4]:
```
#df.describe()
```

[5]:
```
#df.corr()
```

2 Creating a New Binary Column for Good Wines

We aim to classify wines in a binary fashion. The "quality" column is a number between 1-10. We create the new binary column by first defining a function (this is good coding practice), and then applying the function to the dataframe.

[6]:
```
def goodwine (quality):
    if quality>= 7:
        return 1
    return 0
```

[7]:
```
df['goodwine'] = df.quality.apply(goodwine)
```

Alternatively, you may use the lambda syntax as folows:

[8]:
```
#df['goodwine'] = df.quality.apply(lambda x: 1 if x >= 7 else 0)
```

3 Splitting the Data Set

SciKit-Learn takes its input in the form of NumPy arrays.columns, and our classification label is the 13th column: Our features are the first 11

[9]:
```
import numpy as np
X = np.array (df[df.columns[:11]])
y = np.array(df.goodwine)


#check that the resulting arrays have the correct dimensions
print (X.shape)
print (y.shape)
```

```
(800, 11)
(800,)
```

The data may then be split as follows, keeping 400 instances for training, 200 instances for validation and 200 instances for testing. Note that the data should not be shuffled.

[10]:
```
x_train, x_validate, x_test = X[:400], X[400:600], X[600:]
y_train, y_validate, y_test = y[:400], y[400:600], y[600:]  # Split data
```

4 Normalise the data using a Z-score transform

We are going to normalise the data as the Nearest Neighbours classifier is sensitive to scal- ing. While here the training set and the test set 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.

[11]:
```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler().fit(x_train)

x_train_norm = scaler.transform(x_train)
x_validate_norm = scaler.transform(x_validate)
x_test_norm = scaler.transform(x_test)
```

The last command can be run after training, should the test data be initially unavailable.

5 Loading and training the kNN classifier

First we need to import the Nearest Neighbours package:

[12]:
```
from sklearn.neighbors import KNeighborsClassifier
```

as an example, we may train the classifier with a value of k, say k = 3

[13]:
```
clf = KNeighborsClassifier(3) #initialise
clf.fit(x_train_norm, y_train) #train the classifier using the training set
```

[13]:
```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=None, n_neighbors=3, p=2,
                     weights='uniform')
```

Note: Nearest Neighbours is a 'lazy' classifier. At training stage, the classifier only stores the data as well as the parameter k. To evaluate the performance of your classifier on the training data, we type:

[14]:
```
clf.score(x_train_norm, y_train)
```

[14]:
```
0.935
```

For our initial choice of k = 3, we now estimate the performance on new data using the validation set. Unless you have made up your mind that k = 3 is the final classifier that you are going to use (for example for your coursework), you should not use the test set for performance evaluation.

[15]:
```
clf.score(x_validate_norm, y_validate)
```

[15]:
```
0.77
```

We are going to use cross-validation to choose the value of k that has the most promising performance on future data:

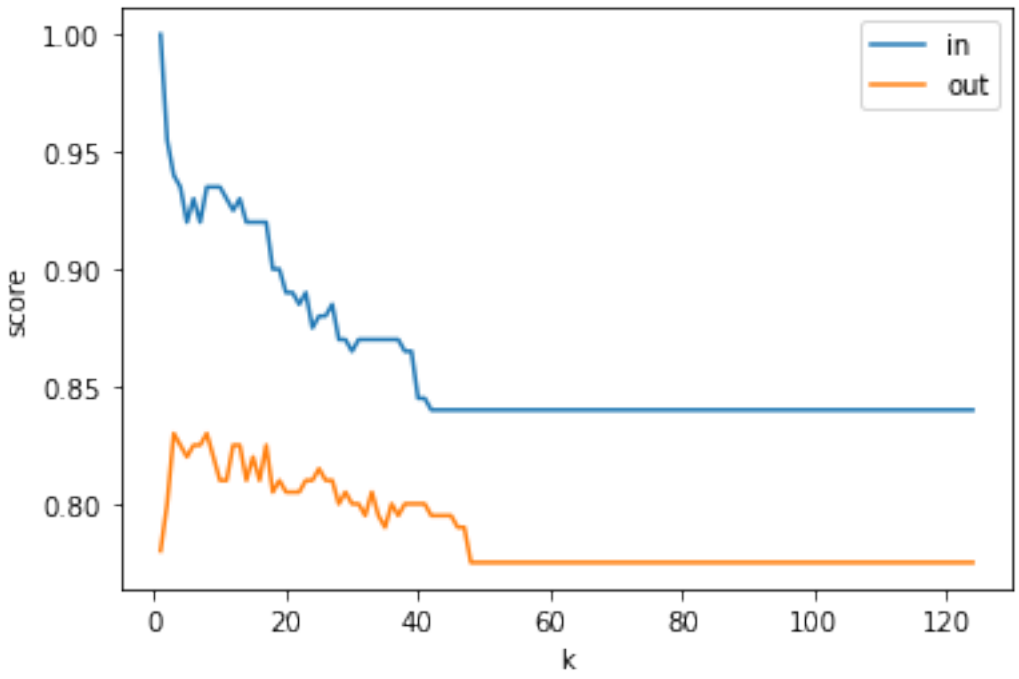
[16]:
```
ks = range(1, 125,1)
inSampleScores =[]
valScores = []

for k in ks:
    clf = KNeighborsClassifier(k).fit(x_train_norm, y_train)
    inSampleScores.append(clf.score(x_train_norm, y_train))
    valScores.append(clf.score(x_validate_norm, y_validate))
```

6 Selecting the best k

It is best to visualise the performace through plots of the recorded scores.

[22]:
```
import matplotlib.pyplot as plt

p1 = plt.plot(ks, inSampleScores)
p2 = plt.plot(ks, valScores)
plt.legend(['in','out'])
plt.xlabel('k')
plt.ylabel('score')
```

[22]:
```
Text(0, 0.5, 'score')
```



As we have seen in class, for small (large) values of k the Nearest Neighbours classifier suffers from overfitting (underfitting). Our plot indicates that a k around 8 is a reasonable choice.

7 Evaluating the Selected Classifier on the Test Set

We can now evaluate the classifier with k = 8 on the test set:

[18]:
```
clf = KNeighborsClassifier(8).fit(x_train_norm, y_train)
y_test_pred = clf.predict(x_test_norm)
```

As we also have the actual outputs y test, we can calculate the performance:

[19]:
```
clf.score(x_test_norm, y_test)
```

[19]: 0.785

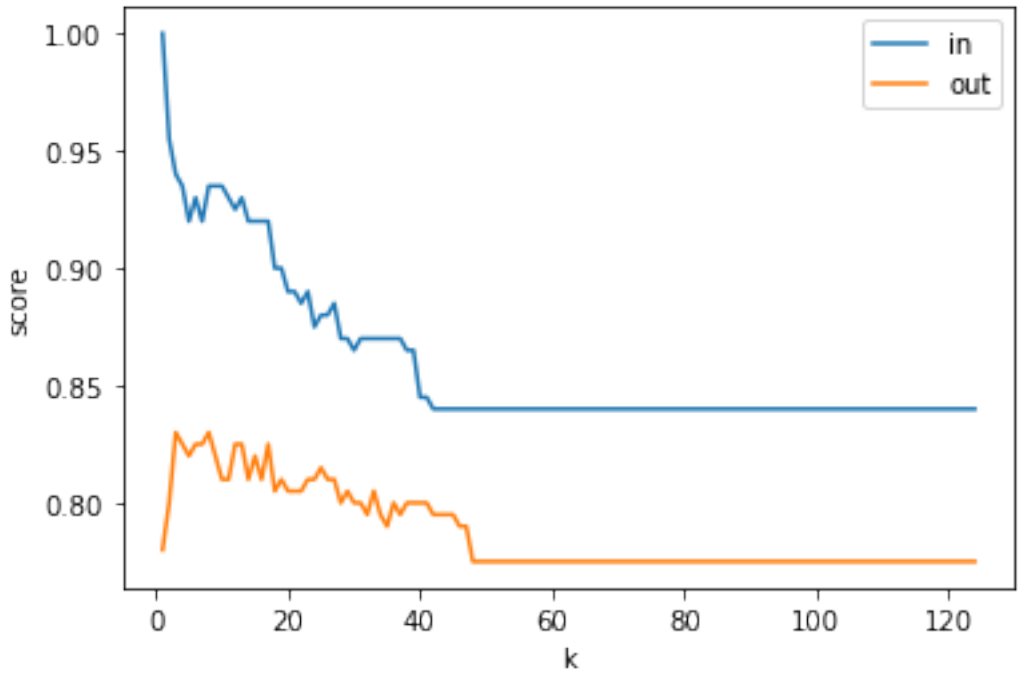
8 Trying a different split

[20]:
```
X = np.array (df[df.columns[:11]])
y = np.array(df.goodwine)
#check that the resulting arrays have the correct dimensions
print (X.shape)
print (y.shape)

x_train, x_validate, x_test = X[:200], X[200:400], X[600:]
```

```
y_train, y_validate, y_test = y[:200], y[200:400], y[600:]  # Split data

scaler = StandardScaler().fit(x_train)
x_train_norm = scaler.transform(x_train)
x_validate_norm = scaler.transform(x_validate)
x_test_norm = scaler.transform(x_test)

ks = range(1, 125,1)
inSampleScores =[]
valScores = []
for k in ks:
    clf = KNeighborsClassifier(k).fit(x_train_norm, y_train)
    inSampleScores.append(clf.score(x_train_norm, y_train))
    valScores.append(clf.score(x_validate_norm, y_validate))

p1 = plt.plot(ks, inSampleScores)
p2 = plt.plot(ks, valScores)
plt.legend(['in','out'])
plt.xlabel('k')
plt.ylabel('score')
```

```
(800, 11)
(800,)
```

[20]: `Text(0, 0.5, 'score')`



[21]:
```
#k = 8 seems ok again
clf = KNeighborsClassifier(8).fit(x_train_norm, y_train)
y_test_pred = clf.predict(x_test_norm)
clf.score(x_test_norm, y_test)
```

[21]: 0.805

Comments should address: 1. how the size of the train and validation sets affects the training 2. how the size of the test set affects evaluation 3. compare and explain overfitting in the case of each of the splits 4. answers should not try to justify the scores obtained even if they are counter-intuitive

9 How to judge if the classifier is suited to the dataset

Comments should address overfittng, evaluation metrics and the curse of dimensionality: 1. What constitutes the majority class? Is the dataset balanced? 2. In relation to the above, is accuracy is good judge of the performace? 3. What other metrics can be used to evaluate performance? precision/recall? - which should be used and in which context? 4. What level of generalisation error is acceptable? 5. How many dimensions are involved in this dataset? What is the level of complexity? Is kNN suited for this level of dimensionality? 6. What was the training time? Was the training computationally expensive?