Programming for Data Science and Artificial Intelligence

Supervised Learning - Classification - K-Nearest Neighbors

Readings:

- [VANDER] Ch5
- [HASTIE] Ch9, 13

In [1]:

```
Name = "Muhammad Omer Farooq Bhatti"
Id = "st122498"
```

In [9]:

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.datasets import make_blobs
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import average_precision_score, classification_report
from sklearn.preprocessing import label_binarize
```

K-Nearest Neighbors

The intuition behind the KNN algorithm is one of the simplest of all the supervised machine learning algorithms. It simply calculates the distance of a new data point to all other training data points. The distance can be of any type e.g Euclidean or Manhattan etc. It then selects the K-nearest data points, where K can be any integer. Finally it assigns the data point to the class to which the majority of the K data points belong.

For example, given the red cross X, it simply get the majority class of neighbors, and assign to its own.

Scratch

Implementation steps:

1. Prepare your data
   - $\mathbf{X}$ and $\mathbf{y}$ in the right shape
     - $\mathbf{X}$ -> (m, n)
     - $\mathbf{y}$ -> $(m,)$
     - Why no w?
   - train-test split
   - feature scale
   - clean out any missing data
   - (optional) feature engineering
2. Write a function for computing pairwise distance between every points
3. Then, given set of X_test data, compute their distance to all other points, then argsort the distance matrix, and get the k-nearest indices
4. Get the majority class

1. Prepare your data

2. Function for pairwise distance

I have written three different ways in numpy assignment answer question 11

3. Argsort the pairwise distance matrix

4. Get the majority class

When to use KNN

I guess the only good thing about it is that KNN is super easy to implement, and generally work quite well on simple classification problems. However, it also comes with a price:

- Computational expense as feature grows, since it requires computing the distance for each feature, where for each feature, we have to compute the input points with every single points, then perform sort (which can be expensive), and then get the majority class from the nearest nth-neighbors. Very expensive!
- Can't work with categorical features since it is difficult to formulate distance formulas for categorial features
- Of course, it takes even more time to find the right n_neighbors (or commonly known as k)

===Task===

Your work: Let's modify the above scratch code to

- If the majority class of the first place is equal to the second place, then ask the algorithm to pick the next nearest neighbors as the decider

- Modify the code so it outputs the probability of the decision, where the probability is simply the class probability based on all the nearest neighbors
- Write a function which allows the program to receive a range of k, and output the cross validation score. Last, it shall inform us which k is the best to use from a predefined range
- Put everything into a class `KNN(k=3)` . It should have at least one method, `predict(X_train, X_test, y_train)`

In [4]:

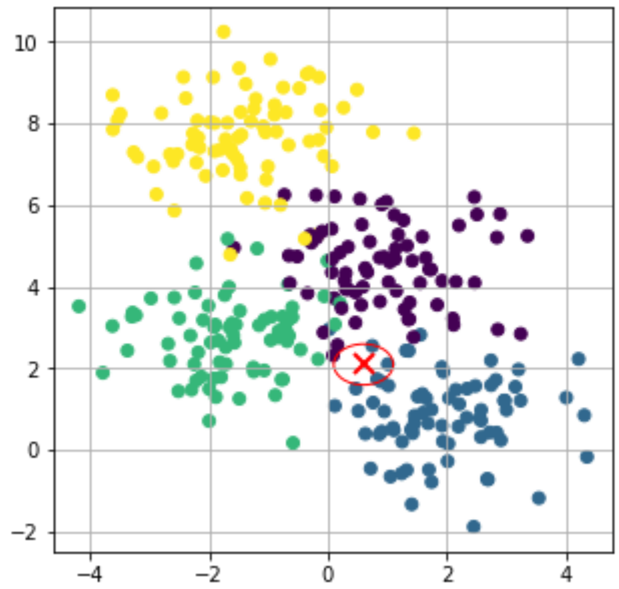
```
class KNN():
    def __init__(self, k=3):
        self.k=k
        self.X_train=None
        self.X_test=None
        self.y_train=None

    def predict(self, X_train, X_test, y_train):
        self.X_train=X_train
        self.X_test=X_test
        self.y_train=y_train
        num_class=len(set(self.y_train))
        neighbours_idx = self.find_all_neighbours()
        prediction=np.zeros(X_test.shape[0])
        probability=np.zeros(X_test.shape[0])
        for idx, y in enumerate(self.y_train[neighbours_idx]):
            prediction[idx], probability[idx] = self.get_most_common(y, num_class)
        return prediction, probability

    def find_k_neighbours(self):
        squared_distance = np.square(self.X_test[:,np.newaxis,:]-self.X_train[np.newaxis,:,:])
        euclidian_distance = np.sqrt(np.sum(squared_distance, axis=2))
        neighbours_idx = np.argsort(euclidian_distance)
        return neighbours_idx[:,:self.k]

    def find_all_neighbours(self):
        squared_distance = np.square(self.X_test[:,np.newaxis,:]-self.X_train[np.newaxis,:,:])
        euclidian_distance = np.sqrt(np.sum(squared_distance, axis=2))
        neighbours_idx = np.argsort(euclidian_distance)
        return neighbours_idx

    def get_most_common(self, y, num_class):
        k_nearest_y = y[:self.k]
        y_count = np.bincount(k_nearest_y, minlength=num_class)
        most_common_y = y_count.argmax()
        second_most_common_y = y_count.argsort()[-2]
        #print(f"k_nearest_y: {k_nearest_y}")
        #print(f"bincount: {y_count}")
        #print(f"num_classes: {num_class}")
        #print(f"most_common_y: {most_common_y}")
        #print(f"second_most_common_y: {second_most_common_y}")
        #print(f"y_count[most_common_y]: {y_count[most_common_y]}")
        #print(f"y_count[second_most_common_y]: {y_count[second_most_common_y]}")

        if (y_count[most_common_y] == y_count[second_most_common_y]):
            k_nearest_y = y[:self.k+1] ## add another y element/neighbour

        y_count = np.bincount(k_nearest_y, minlength=num_class)
        yhat = y_count.argsort()[-1]
        p_yhat = y_count[yhat]/np.sum(y_count)
        return yhat, p_yhat

    def k_fold_cross_validation(self, X_train, y_train, k_range, k=10 ):
        #implementing k fold cross validation for determining the best value of hyperparameter k=no. of neig
        #We divide the training set into k parts. We withhold one part for testing and use the rest for trai
        #Loop through all the folds, by keeping each one separate as a testing set and using the rest for tr

        fold_size = int(X_train.shape[0]/k)
        accuracy = {}
        for k in k_range:
            self.k = k
            accuracy[self.k]=[]
            for i in range(0,X_train.shape[0],fold_size):
                xtest = X_train[i:i+fold_size]
                ytest = y_train[i:i+fold_size]
                xtrain = np.concatenate((X_train[:i], X_train[i+fold_size:]), axis=0)
                ytrain = np.concatenate((y_train[:i], y_train[i+fold_size:]), axis=0)
                yhat, p_yhat = self.predict(xtrain, xtest, ytrain)
                accuracy[self.k].append( np.sum(yhat==ytest)/ytest.shape[0] )
        return accuracy
```

In [14]:

```
X, y = make_blobs(n_samples=300, centers=4,
                  random_state=0, cluster_std=1.0)

xfit = np.linspace(-1, 3.5)

figure = plt.figure(figsize=(5, 5))
ax = plt.axes()  #get the instance of axes from plt

ax.grid()
ax.scatter(X[:, 0], X[:, 1], c=y)

#where should this value be classified as?
ax.plot([0.6], [2.1], 'x', color='red', markeredgewidth=2, markersize=10)

#let's say roughly 5 neighbors
circle = plt.Circle((0.6, 2.1), 0.5, color='red', fill=False)
ax.add_artist(circle)
```

Out[14]:

```
<matplotlib.patches.Circle at 0x1c6c17b19a0>
```



In [15]:

```
#standardize
scaler = StandardScaler()
X = scaler.fit_transform(X)

#do train test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
```

In [16]:

```
model = KNN()
yhat, yp = model.predict(X_train, X_test, y_train)
print(yhat)
print(f"Probability of yhat: {yp}")
```

```
[2. 3. 2. 1. 2. 2. 2. 3. 0. 3. 0. 0. 0. 1. 2. 0. 2. 3. 3. 2. 0. 0. 2. 1.
 3. 0. 3. 1. 1. 3. 1. 3. 1. 0. 1. 2. 3. 1. 2. 2. 1. 3. 1. 0. 0. 2. 2. 3.
 2. 1. 1. 3. 3. 2. 0. 1. 3. 2. 1. 0. 3. 3. 3. 0. 1. 0. 0. 0. 1. 3. 2. 1.
 0. 0. 2. 3. 1. 1. 0. 3. 2. 1. 1. 0. 3. 1. 3. 0. 0. 0.]
[1.         0.66666667 1.         1.         1.         1.
 1.         1.         1.         1.         1.         0.66666667
 1.         1.         1.         1.         1.         0.66666667
 1.         0.66666667 1.         1.         1.         1.
 1.         1.         0.66666667 0.66666667 1.         1.
 1.         1.         1.         1.         1.         1.
 1.         1.         1.         1.         1.         1.
 1.         1.         1.         1.         1.         1.
 0.5        1.         1.         1.         1.         1.
 1.         1.         1.         1.         1.         1.
 1.         1.         1.         1.         1.         1.
 0.66666667 1.         1.         1.         1.         1.
 1.         1.         1.         1.         1.         1.
 1.         1.         1.         1.         0.66666667 1.
 0.66666667 1.         1.         1.         1.         0.66666667]
```

In [17]:

```
n_classes = len(np.unique(y_test))

print("Accuracy: ", np.sum(yhat == y_test)/len(y_test))

print("=========Average precision score=======")
y_test_binarized = label_binarize(y_test, classes=[0, 1, 2, 3])
yhat_binarized = label_binarize(yhat, classes=[0, 1, 2, 3])

for i in range(n_classes):
    class_score = average_precision_score(y_test_binarized[:, i], yhat_binarized[:, i])
    print(f"Class {i} score: ", class_score)

print("=========Classification report=======")
print("Report: ", classification_report(y_test, yhat))
```

```
Accuracy:  0.8888888888888888
=========Average precision score=======
Class 0 score:  0.6385620915032679
Class 1 score:  0.9422222222222223
Class 2 score:  0.8076388888888888
Class 3 score:  0.8990338164251208
=========Classification report=======
Report:                precision    recall  f1-score   support

           0       0.67      0.94      0.78        17
           1       1.00      0.92      0.96        25
           2       0.95      0.79      0.86        24
           3       0.96      0.92      0.94        24

    accuracy                           0.89        90
   macro avg       0.89      0.89      0.88        90
weighted avg       0.91      0.89      0.89        90
```

In [18]:

```
k_range = [3, 4, 5, 6, 7, 8, 9, 10]
accuracy = model.k_fold_cross_validation(X_train, y_train, k_range, k=5 )
acc=np.zeros((len(k_range),2))
for idx, key in enumerate(k_range):
    accuracy[key] = np.mean(accuracy[key])
    print(f"Mean accuracy for k={key}: {accuracy[key]}")
    acc[idx,0] = key
    acc[idx,1] = accuracy[key]
#print(acc)
print(f"The max accuracy achieved was {acc[acc.argmax(axis=0)[1],1]} for k = {acc[acc.argmax(axis=0)[1],0]}"
```

```
Mean accuracy for k=3: 0.9523809523809523
Mean accuracy for k=4: 0.9380952380952381
Mean accuracy for k=5: 0.9380952380952381
Mean accuracy for k=6: 0.9523809523809523
Mean accuracy for k=7: 0.9523809523809523
Mean accuracy for k=8: 0.9428571428571428
Mean accuracy for k=9: 0.9428571428571428
Mean accuracy for k=10: 0.9428571428571428
The max accuracy achieved was 0.9523809523809523 for k = 3.0
```

In [ ]: